Vector search & vector database

What is vector search?

Nov 17th 2022

AI

Vector search is a way to find related objects that have similar characteristics using machine learning models that detect semantic relationships between objects in an index.

Solutions for vector search and recommendation are becoming more and more common. If you want to add a natural language text search on your site, create image search, or build a powerful recommendation system, you'll want to look into using vectors.

SEARCH

The research behind it has been decades in the making, but up until now building and scaling vector search has only been available to the largest of companies like Google, Amazon, and Netflix. These companies have hired thousands of engineers and data scientists, and some have even developed their own computer chips to offer faster machine learning.

Today, just about any company can deploy vector-powered search and recommendations in a fraction of the time and price. Vector technologies unleash a whole new era for developers to build solutions that enable better search, recommendation, and prediction solutions.

This blog offers an introduction to vector search and some of the technology behind it such as vector embeddings and neural networks. And, I'll briefly describe neural hashing, a new technique that enables vectors to be delivered even faster and more efficiently.

The problem with language

Language is often ambiguous and fuzzy. Two words can mean the same thing (synonyms) or the same word can have multiple meanings (polysems). In English for example, "fantastic" and "awesome" can sometimes be synonymous, but "awesome" can also mean many different things — inspiring, daunting, divine, or even plentiful.

Vector embeddings (also known as word embeddings, or just vectors) along with different machine learning techniques such as spelling correction, language processing, category

matching, and more can be used to structure and make sense of language.

What are vector embeddings?

Vectorization is the process of converting words into vectors (numbers) which allows their meaning to be encoded and processed mathematically. You can think of vectors as groups of numbers that represent something. In practice, vectors are used for automating synonyms, clustering documents, detecting specific meanings and intents in queries, and ranking results. Embeddings are very versatile and other objects — like entire documents, images, video, audio, and more — can be embedded too.

We can visualize vectors using a simple 3-dimensional diagram:

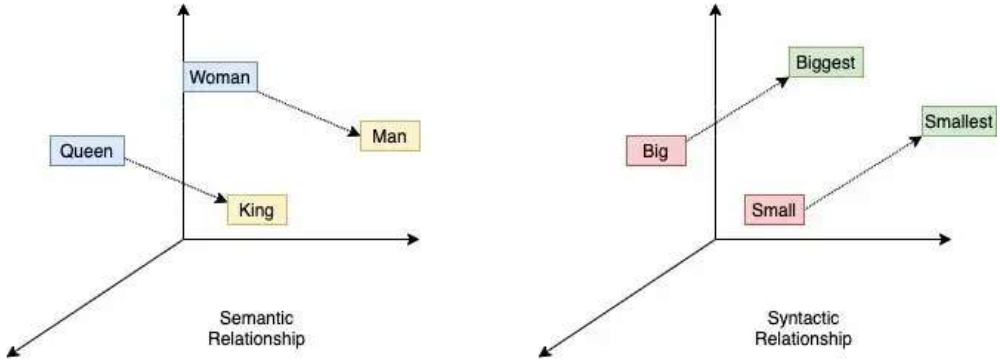


Image via Medium showing vector space dimensions. Similarity is often measured using Euclidean distance or cosine similarity.

You and I can understand the meaning and relationship of terms such as “king,” “queen,” “ruler,” “monarchy,” and “royalty.” With vectors, computers can make sense of these terms by clustering them together in *n*-dimensional space. In the 3-dimensional examples above, each term can be located with coordinates (x, y, z), and similarity can be calculated using distance and angles.

In practice, there can be billions of points and thousands of dimensions. Machine learning models can then be applied to understand that words which are close together in vector

space — like “king” and “queen” — are related, and words that are even closer — “queen” and “ruler” — may be synonymous.

Vectors can also be added, subtracted, or multiplied to find meaning and build relationships. One of the most popular examples is **king – man + woman = queen**. Machines might use this kind of relationship to determine gender or understand gender relationship. Search engines could use this capability to determine the largest mountain ranges in an area, find “the best” vacation itinerary, or identify diet cola alternatives. Those are just three examples, but there are thousands more!

How vector embeddings are created

Some of the earliest models and attempts to represent words as vectors go back to the 1950s with roots in computational linguistics. In the 1960s, research on semantic differentials attempted to measure the semantics, or meaning, of words. Natural language processing (NLP), a way to analyze text to infer meaning and structure, began with complex sets of handwritten rules, but turned to new machine learning models in the 1980s. NLP is still used today in search engines to help structure queries.

It was in the late 1980s that a new statistical model, latent semantic analysis (LSA), also called latent semantic indexing (LSI), was developed for creating vectors and performing information retrieval. LSA is very good at understanding document relatedness by analyzing what terms are frequently used together to build a model of semantic relatedness (e.g., “royalty” and “queen”).

It is a good approach for handling certain kinds of problems — such as synonyms and polysems, and measuring distance (or similarity) between objects — however, it has difficulty scaling. LSA can be computationally expensive especially as the number of vectors increases or as the underlying data changes — for example, every time you update your catalog.

In 2013, Word2Vec was introduced as a new model to understand word similarity using neural networks. Like LSA, Word2Vec can be used to create the word embeddings and then be trained to find text that is semantically similar.

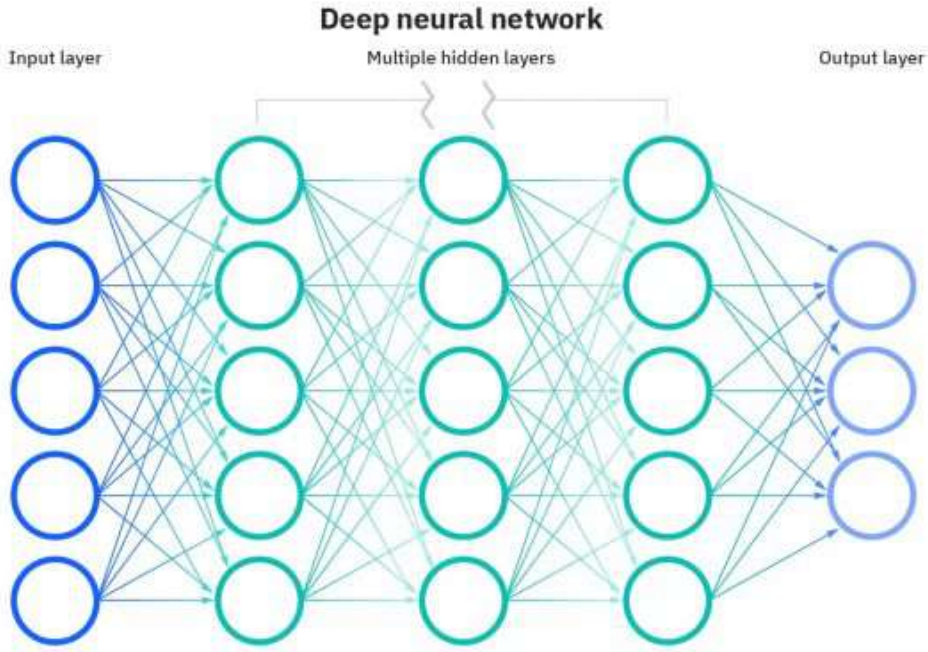


Image via IBM

As the name suggests, neural networks are machine learning networks that resemble the neurons in a brain. Underlying neural networks is a type of machine learning known as deep learning. Every "neuron" in a neural network is essentially just a mathematical function. The weighted total of each neuron's inputs is calculated; the more significant an input's weight, the more it influences the neuron's output.

You can find deep learning used in voice assistants, facial recognition, self-driving cars, and many other applications. Deep learning can be trained on enormous datasets and is able to recognize a large number of complex patterns.

Examples of vector search results

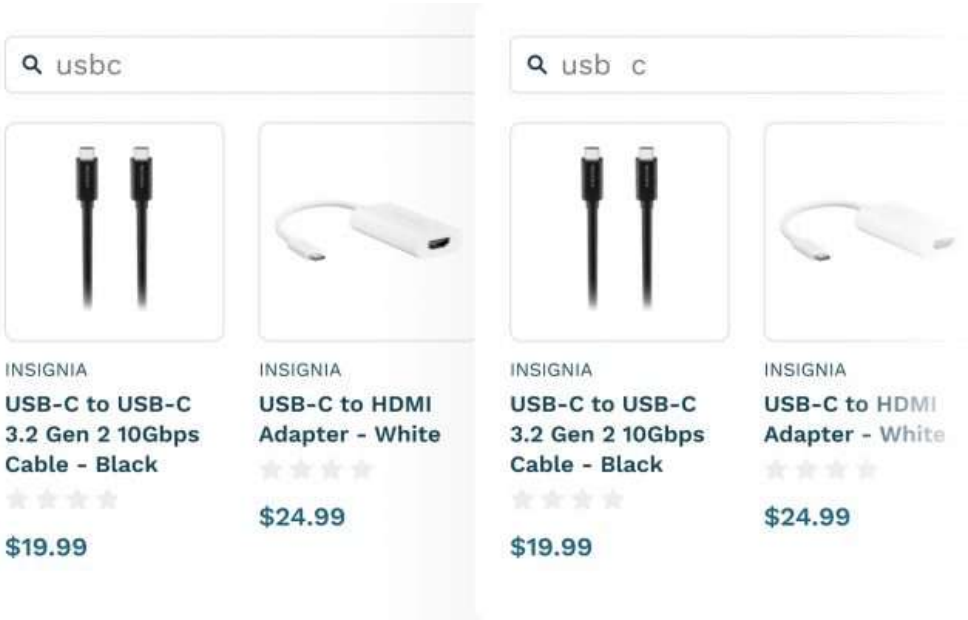
Nowadays, there is a wide diversity of vector embedding models to process different data such as images, videos, and audio. There are also many freely available vector databases

with vector embeddings and distance metrics that represent nearness or similarity between vectors.

There are also various algorithms which can be used to search a vector database to find similarity. These include:

- ANN (approximate nearest neighbor): an algorithm that uses distance algorithms to locate nearby vectors.
- kNN: an algorithm that uses proximity to make predictions about grouping.
- (SPTAG) Space partition tree and graph: a library for large scale approximate nearest neighbors.
- Faiss: Facebook's similarity search algorithm.
- HNSW (hierarchical navigable small world): a multilayered graph approach for determining similarity.

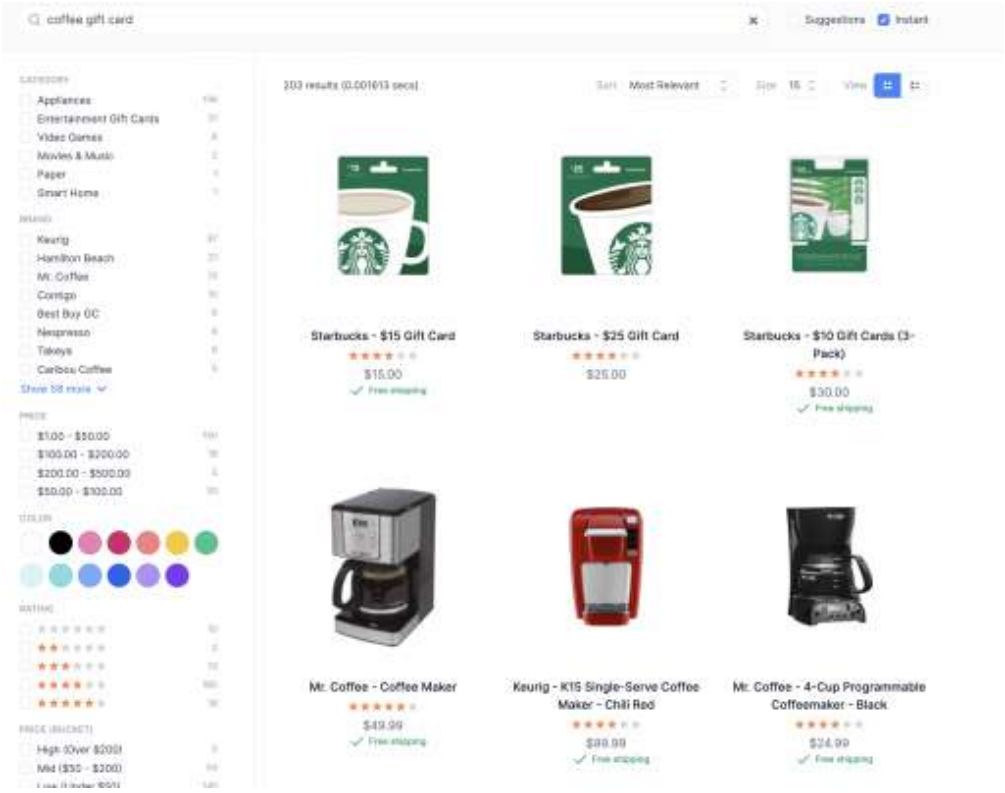
There are tradeoffs between these different techniques and often you'll see multiple techniques being used to deliver results faster and with greater accuracy. These various techniques will deliver better results even for hard-to-process queries. We will write a future blog about these different techniques and tradeoffs.



For example, when searching an electronics catalog, people sometimes type "usbc", "usb-c", or "usb c". Do these mean the same thing, or is it for three different items? Keyword engines can struggle with this kind of formatting, and typically you

might need to create if/then rules to teach the search engine how to manage this query. However, with vector search, this isn't a problem. Vector search engines will know to deliver similar results.

Here's a more interesting example:



In our test database with more than 20,000 products — which includes only product titles and brand names — we performed a search for "coffee gift card" (above). The term "coffee" is not in Starbucks gift card description, however, the vector engine can make the connection between "coffee" and "starbucks" to return good results!

Vector search challenges

Vector embeddings help us to find similarity between documents. When it comes to relevance, vector search is superior to keyword search for many types of queries. If they're so great, why don't we use vector search for everything? In fact, for many query types, keyword search still provides better relevance. Additionally, vector search is not very efficient and historically can't scale without a significant

investment in computer processing. With new, recently introduced neural hashing capabilities, vector search is finally able to scale. More on this below.

Accuracy vs keyword search

Vector search is terrific for fuzzy or broad searches, but keyword search still rules the roost for precise queries. As the name suggests, keyword search tries to match exact keywords. Other features such as autocomplete, instant search, and filters have also made keyword search popular.

For example, when you query for “Adidas” on a keyword engine, by default you will only see the Adidas brand. The default behavior in a vector engine is to return *similar* results — Nike, Puma, Adidas, etc.. They are all in the same conceptual space. Keyword search still provides better results for short queries with specific intention.

Speed and scale

Bottlenecks are more likely with vector search because queries must do complex vector calculations to predict relationships as opposed to just reading column based indexes. Machines divide CPU time between various inbound processes. In fact, much of the embedding requires GPU inference also, which includes the queries, so this is even more complicated in some ways.

To cope, search engines either need more compute power or must instead process the same queries faster. Vector search companies have been pushing the benefits of vector AI for years, but the cost and performance issues have impeded its progress and engendered concerns about its viability.

Some companies that offer vector search module add-ons will attempt to skirt the problem by only running the vector search if the keyword search result is poor. The message is that you

can have one or the other — keywords or vectors, speed or quality — but not both running at the same time.

Some have suggested that caching is a good way around this problem. The argument goes that by caching results you can virtually eliminate costs and provide results instantly. In practice, search queries vary considerably and the cost benefit for caches is often questionable. The cache rate of search can be extremely low, especially for sites with massive longtail content (using our own customer data we have seen, on average, 50% of the traffic are longtail queries that are not frequent enough to be cached).

One fix to all of these problems — accuracy, speed, scalability, and cost — is called neural hashing. We'll explain briefly how it works.

Binary vectors

Vectors work, but as mentioned above, have speed and scale limitations that affect performance and cost. We took a different approach, called neural hashing, that leverages vectors without tradeoffs.

Vector search engines use neural networks and deep learning models to deliver semantic search capabilities.

Neural hashing makes vector-based search as fast as keyword search and this is done without the need for GPUs or specialized hardware. Neural hashing uses neural networks to

hash vectors — compressing the vectors into binary hashes (or binary vectors). You may have heard of hashes; cryptographic hashing is a commonly used technique in security for producing a tiny, unique output for protected password comparisons.

Performance-wise, these hashed vectors can be run on commodity hardware, retain 96% (or more!) of the vector information, and can be calculated potentially hundreds of times faster than vectors alone.

Now, if there was only some way to get keyword search and neural hashing into the same query....

Hybrid search

Hybrid search is a new method to combine a full-text keyword search engine and a vector search engine into a single API to get the best of both worlds.

There is tremendous complexity in running both keyword and vector engines at the same time for the same query. Some companies have opted to go around the complexity by running these processes sequentially — they run a keyword search and then, if a certain relevance threshold isn't met, run a vector search. There are many poor tradeoffs for this such as speed, accuracy, filtering, and heap sorting. These so-called dual systems suffer because the vector databases often don't have the same (or any) filtering capabilities so they return massive amounts of information that's unnecessary.

True hybrid search is different. By combining full-text keyword search and vector search into a single query, customers can get more accurate results fast. For Algolia, we've combined neural hashes with our world-class and blazingly fast keyword search technology into a single API call. It scales to meet the needs of any size dataset — even for indexes that have a lot of

changes with frequent updates and deletions — without any additional overhead.

Hopefully this has provided you with a good overview of vector search and how it can radically improve your site's search results!

Text similarity

→ Choose a ml model for embeddings

→ Create database of embeddings from dataset

→ Now, before proceeding with database creation choose which approach you want to retrieve results.

Non-indexing such as

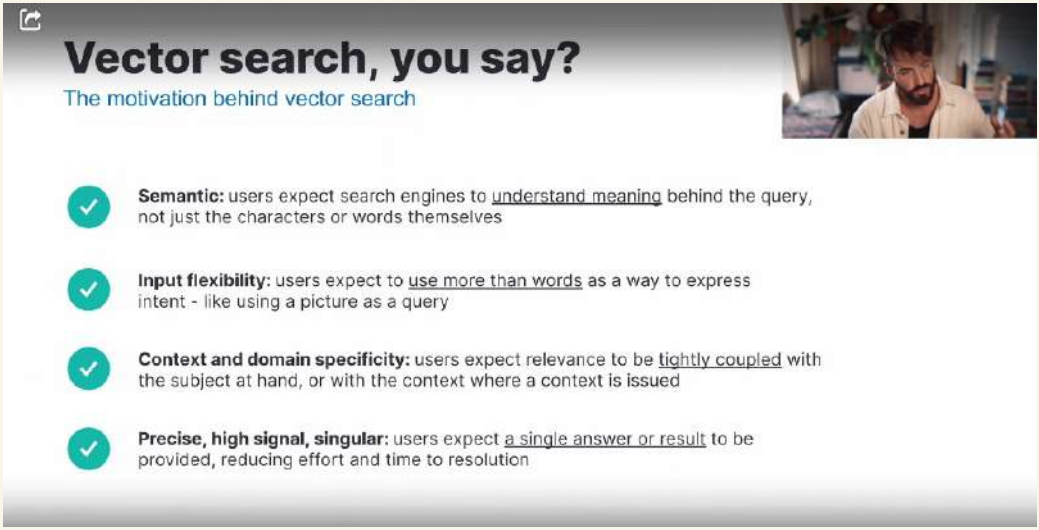
ANN [Approximate nearest neighbour]

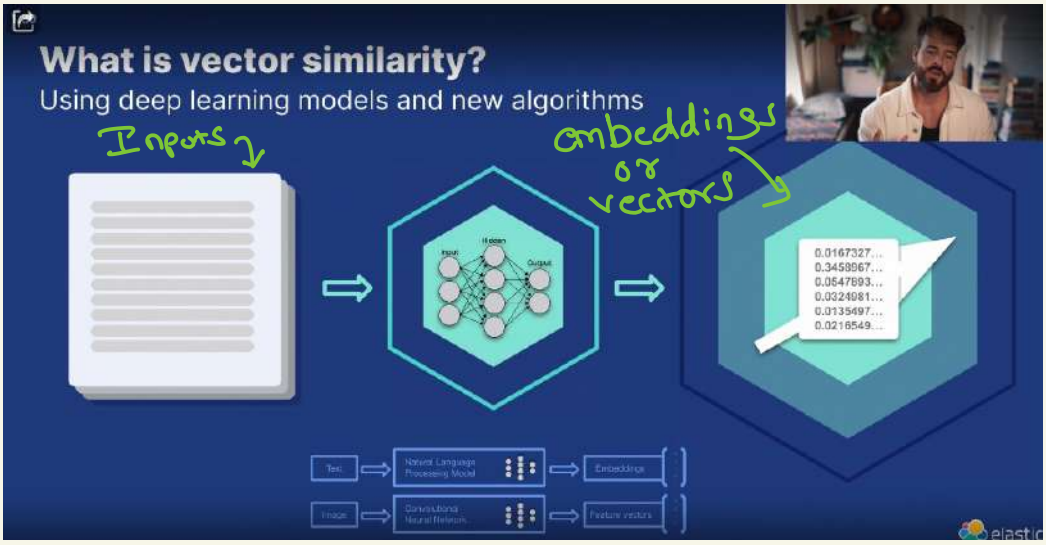
OR

Vector database

→ choose appropriate & create it.

→ Now put query and observe results.

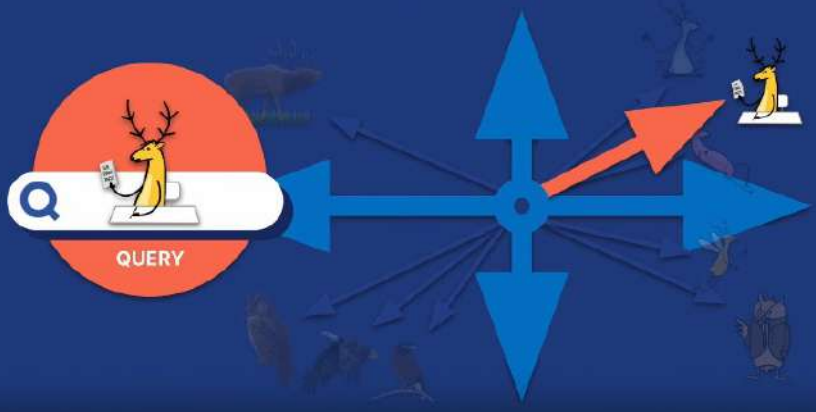




Inputs converted to embeddings

What is vector similarity?

Query is also vectorized (embeddings)



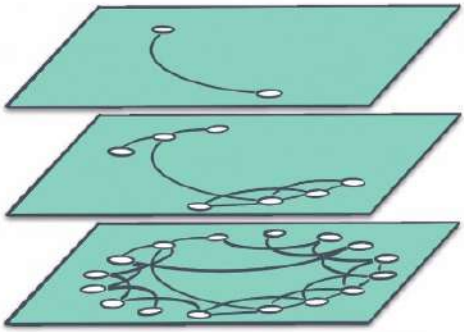
Relevance	Result
Query	
1	
2	
3	
4	
5	

EXTRA

HNSW: performance at scale

Elastic uses the latest in production-grade algorithms

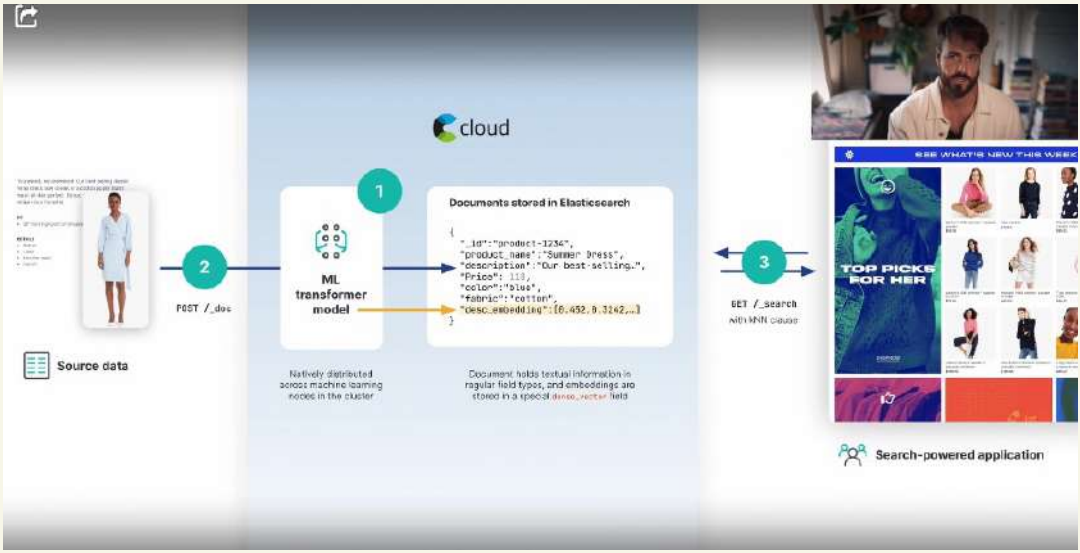
- **Hierarchical Navigable Small Worlds:** a layered approach that simplifies access to the nearest neighbor
- **Tiered:** from coarse to fine approximation over a few steps
- **Balance:** Bartering a little accuracy for a lot of scalability
- **Speed:** Excellent query latency on large scale indices



Vector search, the Elastic way

What you'll need to get searching

- **A use case:** Embedding generation occurs at ingest time, so understanding how the data will be queried is important in selecting the right machine learning model
- **Elastic Cloud:** All of the tools you'll need all ready for you, without hassle - including Elasticsearch, Kibana and machine learning-optimized nodes
- **A machine learning model:** a supported transformer (BERT or similar) PyTorch model, trained by you, or found on popular model repositories such as Hugging Face
- **A search-powered application:** the experience that delivers result(s) to an end user, from natural language querying to question answering



- Here, description field is long text. So this field is converted to vector & stored in 'desc_embedding'

Vector search, the Elastic way

Step 1: Setting up the machine learning model


```
$ eland_import_hub_model
--url https://cluster_URL
--hub-model-id BERT-MiniLM-L6
--task-type text_embedding
--start
```

PyTorch → eland →

Select the appropriate model for the task at hand

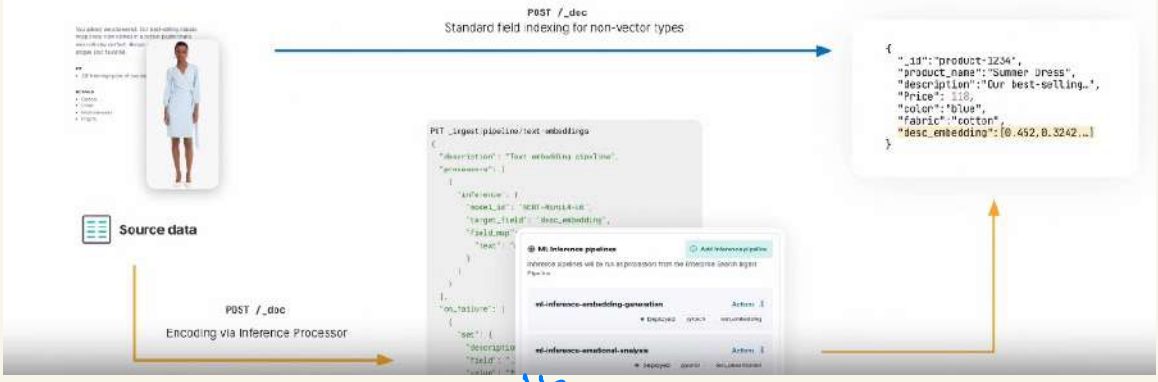
Load the model to the cluster

Manage all loaded models from the Trained Models area

Share Video

Vector search, the Elastic way

Step 2: Data ingestion and embedding generation



↳ This can be used as General approach

Take a ML model

↓

Input the field to ML
(Here description)

↓

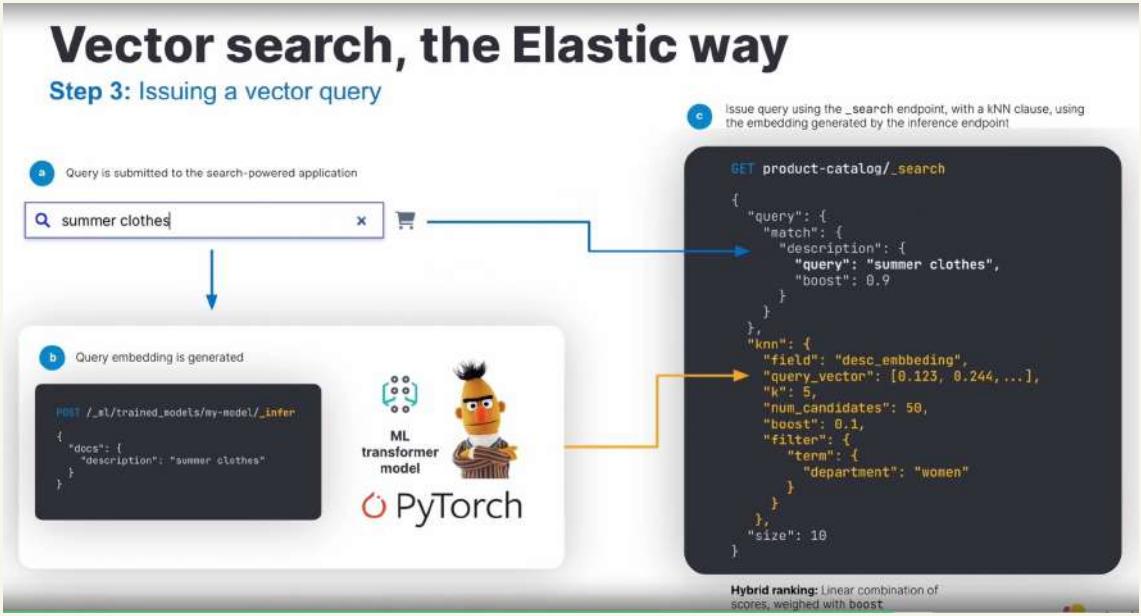
Generate embeddings

↓

Output the embedings to a
field (Here desc_embedings)

↓

Like this create for all rows
in database



User type in search → it process by 2 ways

(Blue arrow)

one by traditional way. Query is matched with words containing

yellow arrow

one by vector search. Query is converted to embedings.

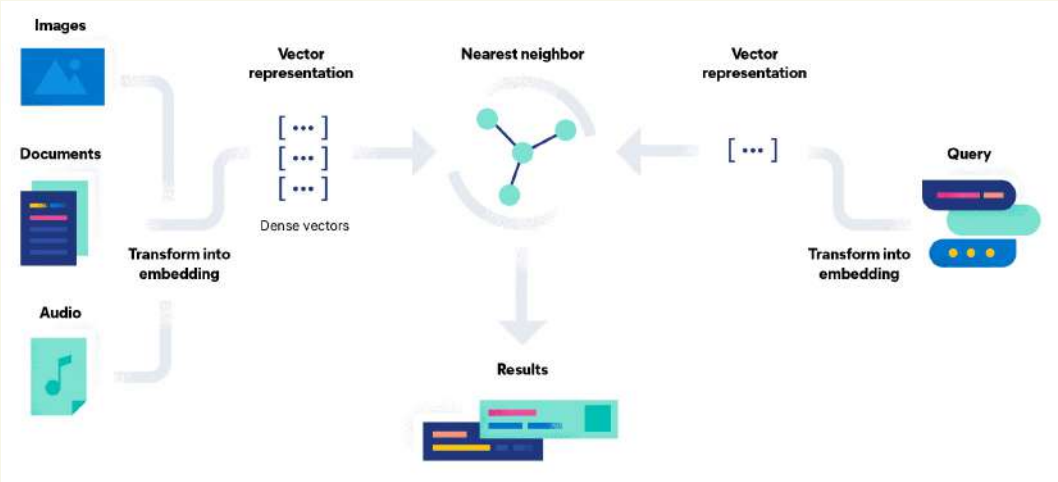
→ We can do hybrid ranking obtained from traditional & vector search

→ That means you can apply your own logic to rank the results obtained from both

How does a vector search engine work?

Vector search engines — known as vector databases, semantic, or cosine search — find the nearest neighbors to a given (vectorized) query.

Where traditional search relies on mentions of keywords, lexical similarity, and the frequency of word occurrences, vector search engines use distances in the embedding space to represent similarity. Finding related data becomes searching for nearest neighbors of your query.



Semantic Search

Semantic search

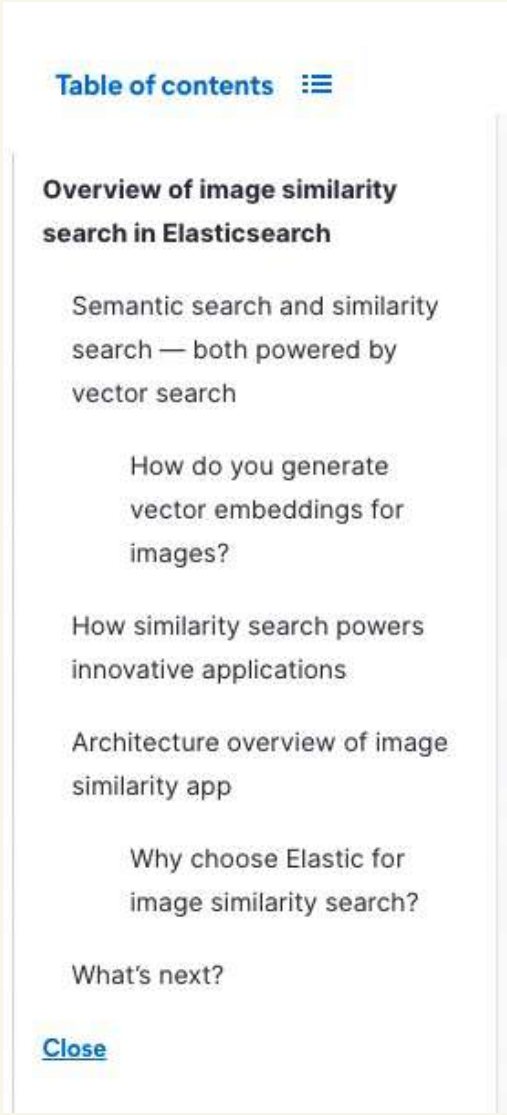
Vector search powers semantic or similarity search. Since the meaning and context is captured in the embedding, vector search finds what users mean, *without requiring an exact keyword match*. It works with textual data (documents), images, and audio. Easily and quickly find products that are similar or related to their query.

Browse unstructured data

Search any unstructured data. You can create embeddings for text, images, audio, or sensor measurements.

How to implement image similarity search →

Overview of image similarity search in Elasticsearch



Semantic search and similarity search — both powered by vector search

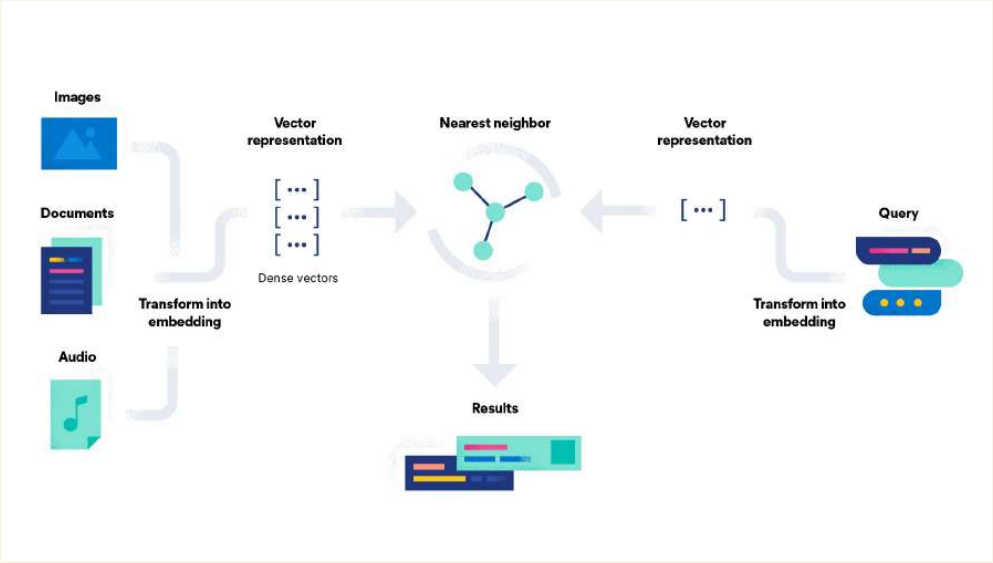
Vector search leverages **machine learning** (ML) to capture the meaning and context of unstructured data. Vector search finds similar data using **approximate nearing neighbor (ANN)** algorithms.

This approach works not only with text data but also images and other types of unstructured data for which generic embedding models are available. For text data, it is commonly referred to as **semantic search**, while **similarity search** is frequently used in the context of images and audio.

How do you generate vector embeddings for images?

Vector embeddings are the numeric representation of data and related context stored in high dimensional (dense) vectors. Models that generate embeddings are typically trained on millions of examples to deliver more relevant and accurate results.

For text data, BERT-like transformers are popular to generate embeddings that work with many types of text, and they are available on public repositories like Hugging Face. Embedding models, which work well on any type of image, are a subject of ongoing research. The **CLIP model** — used by our teams to prototype the image similarity app — is distributed by OpenAI and provides a good starting point. For specialized use cases and advanced users, you may need to train a custom embedding model to achieve desired performance. Next, you need the ability to search efficiently. Elastic supports the widely adopted HNSW-based approximate nearest neighbor search.



How do you generate vector embeddings for images?

Vector embeddings are the numeric representation of data and related context stored in high dimensional (dense) vectors. Models that generate embeddings are typically trained on millions of examples to deliver more relevant and accurate results.

For text data, BERT-like transformers are popular to generate embeddings that work with many types of text, and they are available on public repositories like Hugging Face. Embedding models, which work well on any type of image, are a subject of ongoing research. The **CLIP model** — used by our teams to prototype the image similarity app — is distributed by OpenAI and provides a good starting point. For specialized use cases and advanced users, you may need to train a custom embedding model to achieve desired performance. Next, you need the ability to search efficiently. Elastic supports the widely adopted HNSW-based approximate nearest neighbor search.

Architecture overview of image similarity app

Making this kind of interactive application can seem complex. That's especially true if you have been considering implementation within a traditional architecture as shown below. But in the second diagram, it shows how Elastic significantly simplifies this architecture...

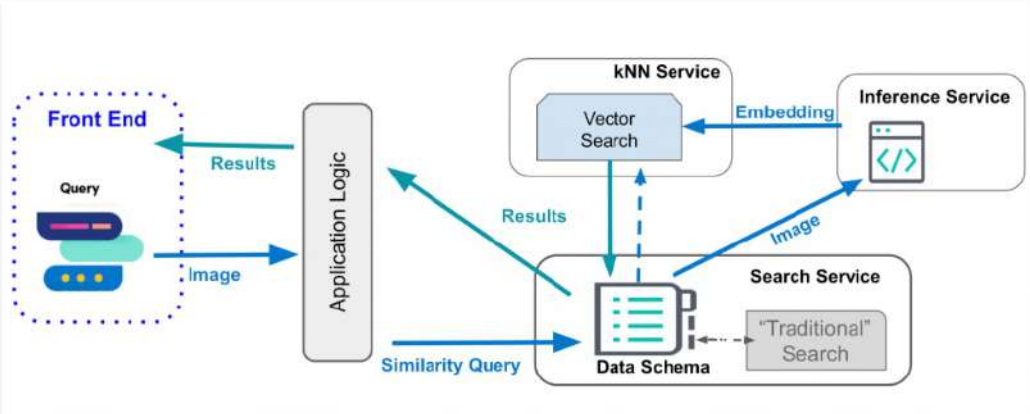
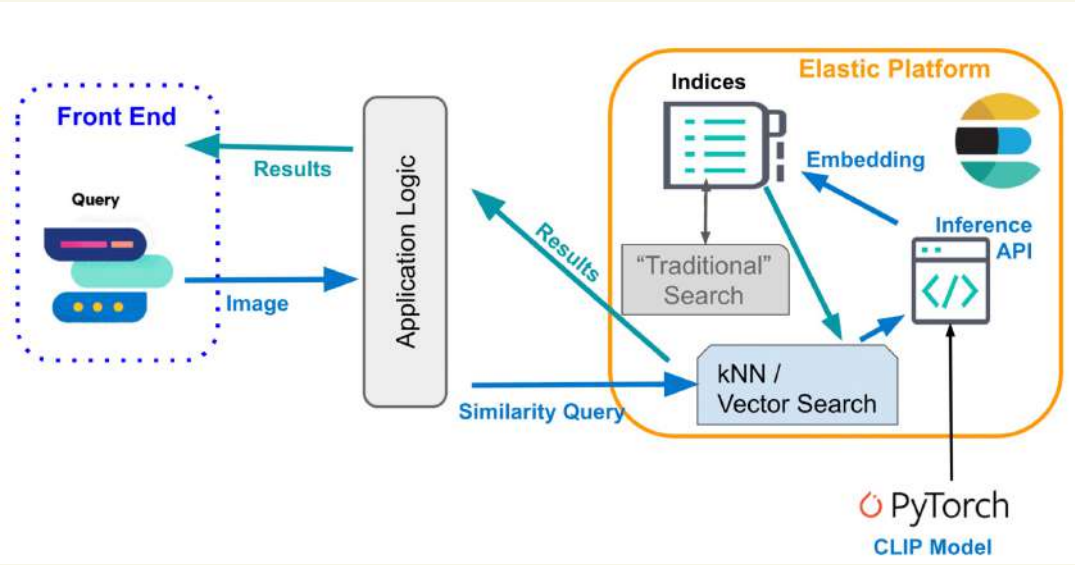


Figure 1: Typical implementation of image similarity search (requires external services)


Front End
Query
Image
Results
Application Logic
Results
Similarity Query
Elastic Platform
Indices
Embedding
Inference API
"Traditional" Search
kNN / Vector Search
PyTorch
CLIP Model

Why choose Elastic for image similarity search?

Implementing image similarity search in Elastic provides you with istinct advantages. With Elastic, you can...

Reduce application complexity. With Elastic you don't need separate services for running kNN search and vectorizing your search input. Vector search and NLP inference endpoints are integrated within a scalable search platform. In other popular frameworks, applying deep neural networks and NLP models occurs separately from scaling searches on large data sets. This means you need to hire experts, add development time to your project, and set aside resources to manage it over time.

Scale with speed. In Elastic, you get scale and speed. Models live alongside nodes running search in the same cluster, which applies to on-premise clusters, and even more so if you deploy to the **cloud**. Elastic Cloud allows you to easily scale up and down, depending on your current search workload.

Reducing the number of services needed by an application has benefits beyond scaling. You can experience simplified performance monitoring, a smaller maintenance footprint, and fewer security vulnerabilities — to name a few. Future **serverless architectures** will take application simplicity to a whole new level.

Introducing approximate nearest neighbor search in Elasticsearch 8.0

k-nearest neighbor (kNN) search algorithms find the vectors in a dataset that are most similar to a query vector. Paired with these vector representations, kNN search opens up exciting possibilities for retrieval:

- Finding passages likely to contain the answer to a question
- Detecting near-duplicate images in a large dataset → Here image dedupe comes
- Finding songs that sound similar to a given song

ANN in Elasticsearch

What is approximate nearest neighbor search?

Explained in next page

There are well-established data structures for kNN on low-dimensional vectors, like KD-trees. In fact, **Elasticsearch incorporates KD-trees** to support searches on geospatial and numeric data. But modern embedding models for text and images typically produce high-dimensional vectors of 100 - 1000 elements, or even more. These vector representations present a unique challenge, as it's very difficult to efficiently find nearest neighbors in high dimensions.

Faced with this difficulty, nearest neighbor algorithms usually sacrifice perfect accuracy to improve their speed. These approximate nearest neighbor (ANN) algorithms may not always return the true *k* nearest vectors. But they run efficiently, scaling to large datasets while maintaining good performance.

Choosing an ANN algorithm

Elasticsearch 8.0 uses an ANN algorithm called Hierarchical Navigable Small World graphs (HNSW), which organizes vectors into a graph based on their similarity to each other. HNSW shows strong search performance across a variety of **ann-benchmarks datasets**, and also did well in our own testing. Another benefit of HNSW is that it's widely used in industry, having been implemented in several different systems. In addition to the **original academic paper**, there are many helpful resources for learning about the algorithm's details. Although Elasticsearch ANN is currently based on HNSW, the feature is designed in a flexible way to let us incorporate different approaches in the future.

KNN Search :-

The new `_knn_search` endpoint uses HNSW graphs to efficiently retrieve similar vectors. Unlike exact kNN, which performs a full scan of the data, it scales well to large datasets. Here's an example that compares `_knn_search` to the exact approach based on `script_score` queries on a dataset of 1 million image vectors with 128 dimensions, averaging over 10,000 different queries:

```
Approach         Queries Per Second    Recall (k=10)
script_score              5.257              1.000
_knn_search             849.286              0.945
```

Copy

In this example, ANN search is orders of magnitude faster than the exact approach. Its recall is around 95%, so on average, it finds over 9 out of the 10 true nearest neighbors.

You can check on the performance of kNN search in the **Elasticsearch nightly benchmarks**. These benchmarks are powered by **es-rally**, a tool for Elasticsearch benchmarking, specifically the new **dense_vector Rally track**. We plan to extend Rally to report recall in addition to latency, as it's also important to track the accuracy of the algorithm. Currently these benchmarks test a dataset of a couple million vectors, but ANN search can certainly scale beyond this with a greater index time or the addition of hardware resources.

> Because it is an approximate algorithm, there are special considerations to running ANN compared to other types of search. ANN has both search-time and index-time parameters to control the trade-off between search latency, result accuracy, and indexing cost. It's important to measure the recall of ANN search on your dataset to make sure the configuration is working well. When jumping into kNN search, the **reference guide** can be a helpful place to start.

Search and Insertion in K Dimensional tree

What is K dimension tree?

Explained in later pages

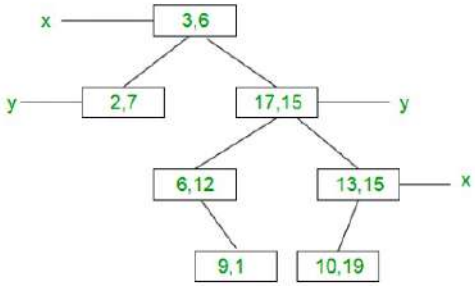
A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space. A non-leaf node in K-D tree divides the space into two parts, called as half-spaces. Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed. For the sake of simplicity, let us understand a 2-D Tree with an example. The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

Generalization: Let us number the planes as 0, 1, 2, ...(K – 1). From the above example, it is quite clear that a point (node) at depth D will have A aligned plane where A is calculated as: A = D mod K

How to determine if a point will lie in the left subtree or in right subtree? If the root node is aligned in planeA, then the left subtree will contain all points whose coordinates in that plane are smaller than that of root node. Similarly, the right subtree will contain all points whose coordinates in that plane are greater-equal to that of root node.

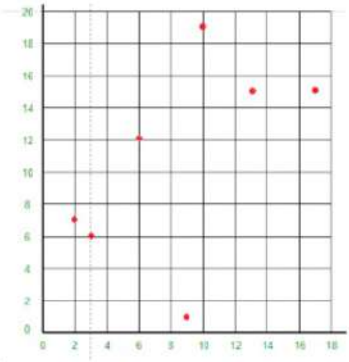
Creation of a 2-D Tree: Consider following points in a 2-D plane: (3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

1. Insert (3, 6): Since tree is empty, make it the root node.
2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the right subtree or in the left subtree. This point will be Y-aligned.
3. Insert (13, 15): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.
4. Insert (6, 12): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, 12 < 15, this point will lie in the left subtree of (17, 15). This point will be X-aligned.
5. Insert (9, 1):Similarly, this point will lie in the right of (6, 12).
6. Insert (2, 7):Similarly, this point will lie in the left of (3, 6).
7. Insert (10, 19): Similarly, this point will lie in the left of (13, 15).

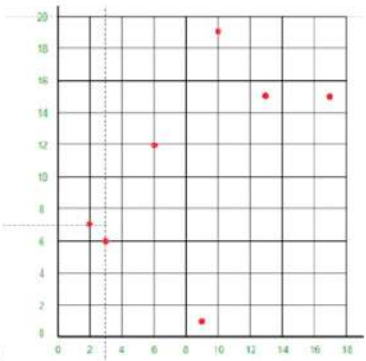


How is space partitioned?

All 7 points will be plotted in the X-Y plane as follows:

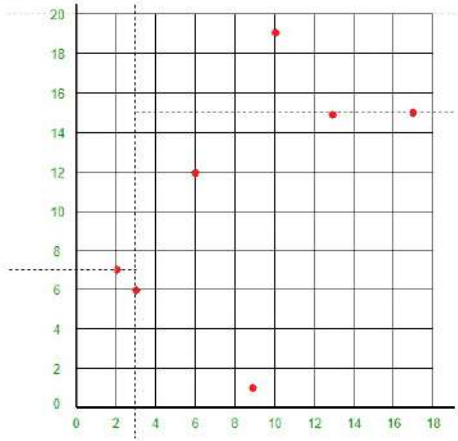


1. Point (3, 6) will divide the space into two parts: Draw line X = 3.

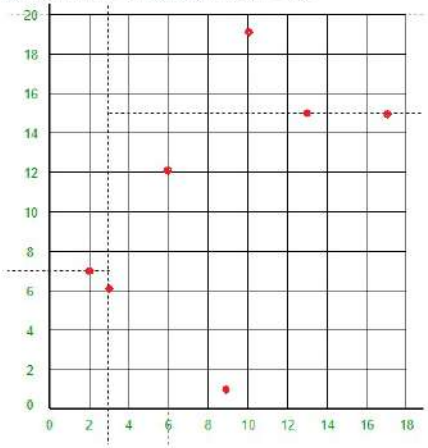


2. Point (2, 7) will divide the space to the left of line X = 3 into two parts horizontally. Draw line Y = 7 to the left of line X = 3.

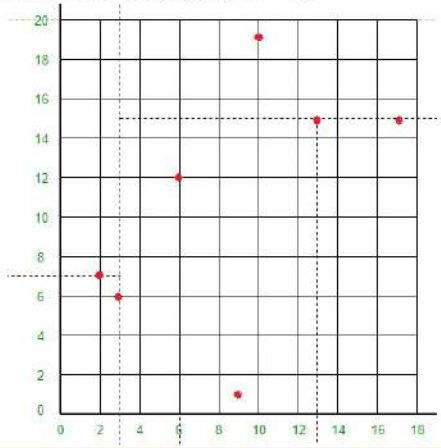
3. Point (17, 15) will divide the space to the right of line X = 3 into two parts horizontally. Draw line Y = 15 to the right of line X = 3.



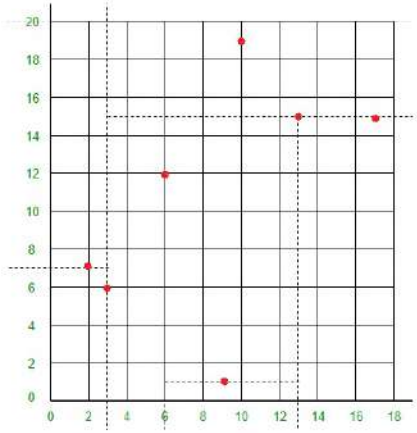
4. Point (6, 12) will divide the space below line Y = 15 and to the right of line X = 3 into two parts. Draw line X = 6 to the right of line X = 3 and below line Y = 15.



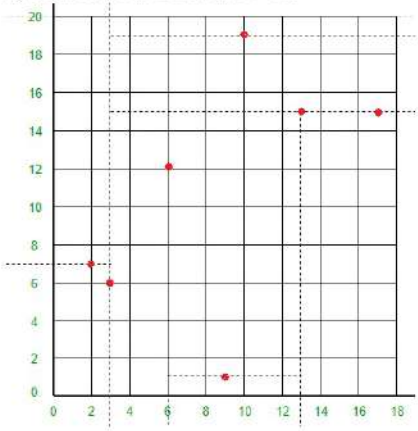
5. Point (13, 15) will divide the space below line Y = 15 and to the right of line X = 6 into two parts. Draw line X = 13 to the right of line X = 6 and below line Y = 15.



6. Point (9, 1) will divide the space between lines X = 3, X = 6 and Y = 15 into two parts. Draw line Y = 1 between lines X = 3 and X = 13.



7. Point (10, 19) will divide the space to the right of line X = 3 and above line Y = 15 into two parts. Draw line Y = 19 to the right of line X = 3 and above line Y = 15.

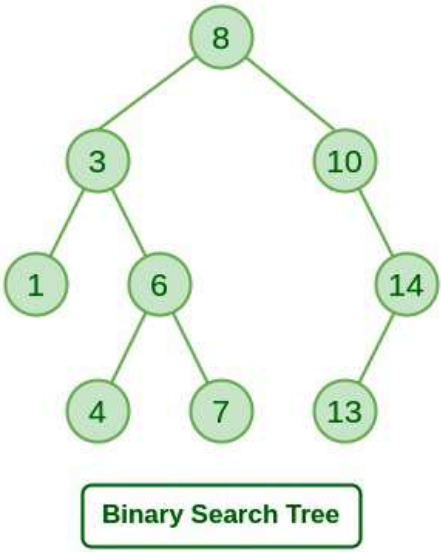


Binary Search Tree

What is Binary Search Tree?

Binary Search Tree is a node-based binary tree data structure which has the following properties:

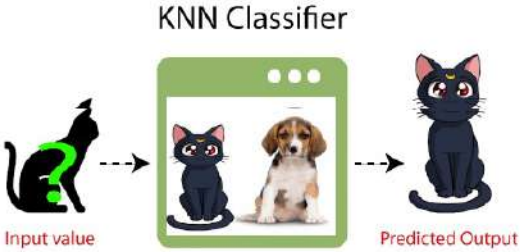
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree

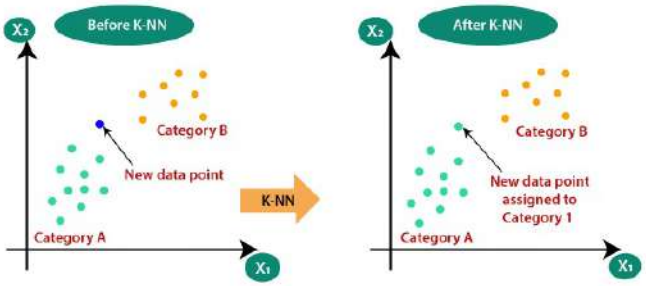
K-Nearest Neighbor(KNN) Algorithm for Machine Learning

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.
- **Example:** Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So for this identification, we can use the KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs images and based on the most similar features it will put it in either cat or dog category.



Why do we need a K-NN Algorithm?

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x1, so this data point will lie in which of these categories. To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram:

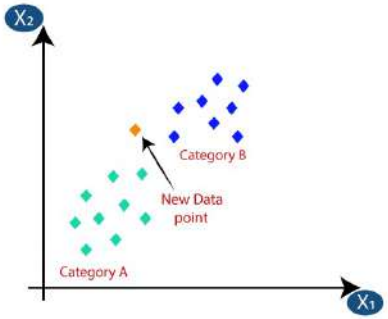


How does K-NN work?

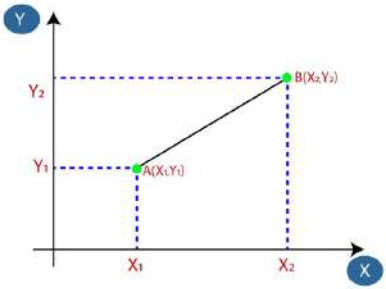
The K-NN working can be explained on the basis of the below algorithm:

- **Step-1:** Select the number K of the neighbors
- **Step-2:** Calculate the Euclidean distance of **K number of neighbors**
- **Step-3:** Take the K nearest neighbors as per the calculated Euclidean distance.
- **Step-4:** Among these k neighbors, count the number of the data points in each category.
- **Step-5:** Assign the new data points to that category for which the number of the neighbor is maximum.
- **Step-6:** Our model is ready.

Suppose we have a new data point and we need to put it in the required category. Consider the below image:

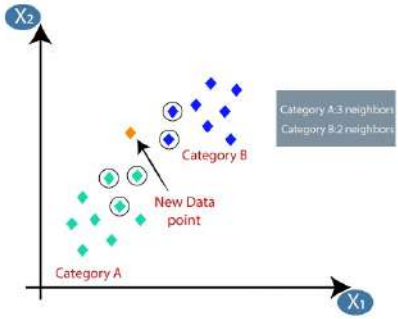


- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



Euclidean Distance between A_1 and $B_2 = \sqrt{(X_2-X_1)^2+(Y_2-Y_1)^2}$

- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B. Consider the below image:



- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

How to select the value of K in the K-NN Algorithm?

Below are some points to remember while selecting the value of K in the K-NN algorithm:

- There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.
- A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model.
- Large values for K are good, but it may find some difficulties.

Advantages of KNN Algorithm:

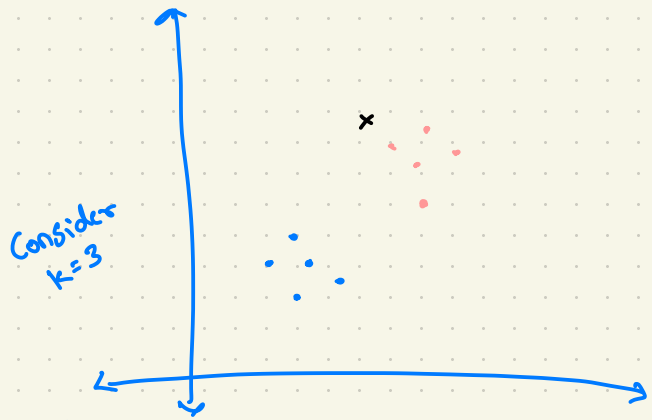
- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of KNN Algorithm:

- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

KNN

- Classification problem:-



Given:-

- Consider data points above 5 blue and 5 red dots in graph in 2D
- Now a new point comes ×
- In KNN, we need to find 3 nearest points to ×
- For this × will visit all points and calculate its distance & we will get 3 minimum distance from ×
- Now here, × is nearer to 3 red points
- So unknown × belongs to red cluster.
- So order of complexity is O(n) since, from above, 10 points are present in dataset, so distance had to be calculated in order to get 3 nearest neighbors. You have to visit every single point since for any unknown point any points in dataset could be your closest neighbors.

ANN

→ Similar to KNN & also same goal as KNN

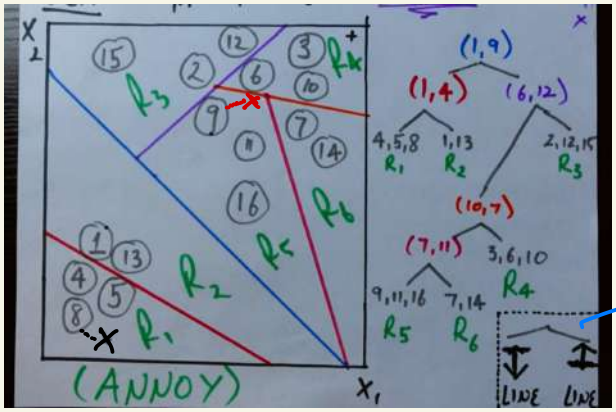
→ But its gonna be fixing one of major issues of KNN

→ Instead of finding exact closest neighbors like in KNN, ANN finds almost closest neighbors and in fraction of time

→ One of common ANN also is ANNOY (approximate nearest neighbor oh yeah)

Classification problem

Condition:
- region should contain atleast 2 data points
- K = 1



All regions below line will go left & above line will go right of tree

-> How ANNOY works:
-> Consider 2 random points ①, ⑨
-> Calculate distance and draw perpendicular line in both as shown in blue line
-> So now graph is divided into 2 regions one above the blue line & one below the blue line
-> Consider below line consider 2 random points ① & ⑭
-> Again calculate distance & draw perpendicular line in both as shown in red line
-> Consider below red. We stop further region creation as it violates the above condition.
-> visit other regions and try to further divide regions. Once all done, stop.
-> Consider x besides ⑧, if we want to see closest neighbor, refer graph x is below blue line, so it will go left of (1,9). Now, in subtree, again x is below red line, so it will go left of (1,4)
-> Now it has come to lowest subregion. Here, in this region there are 3 points ④, ⑤, ⑧. Calculate distance & find least distance among them (as K=1)
-> Here ⑧ is minimum, whatever ⑧ is labelled, the unknown point x will be assigned that.

Disadvantage:

- It is approximate nearest, Consider x sign above. Actually ⑥ is nearest. but since x is in another region then as per ANN logic ④ comes as nearest as least distance distance among that region is calculated

Annoy pip package

https://pypi.org/project/annoy/ → Not optimized and may take memory - (already warned)

Long history of work.. In academia and industry

Many classes of algorithms

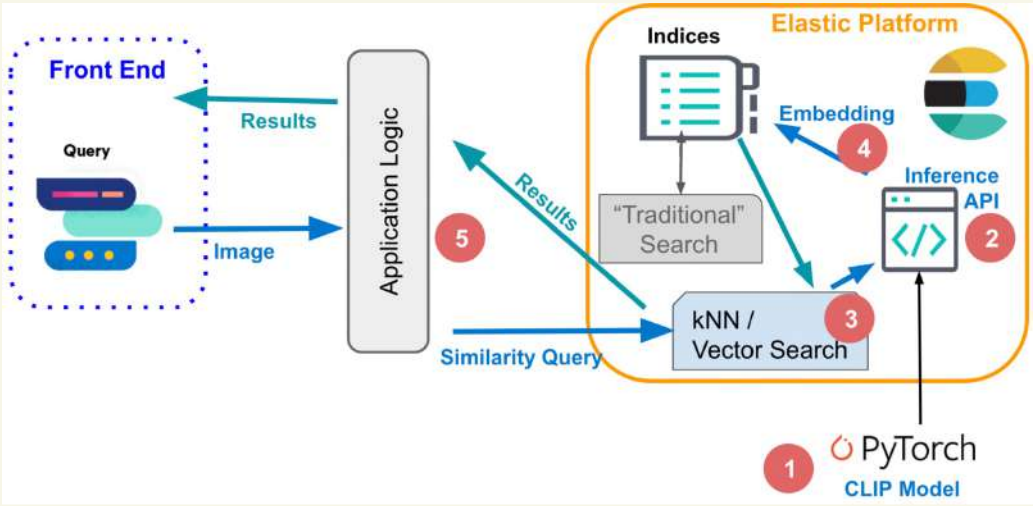
- Trees: k-d trees, Ball trees, cover trees,...
- Clustering: IVF (FAISS)...
- Hashing: LSH, Data dependent hashing, Asymmetric hashing,...
- Graph based: NSW, HNSW, NSG, NGT, SPTAG, NGT, ONNG,...
- Vector compression to reduce index size
  - PQ, OPQ... For L2 metric
  - SCANN for inner product
- Domain specific:
  - Sparse high-dimensional
  - Geospatial

Part 1. 5 technical components of image similarity search

In the **first part** of this series of blog posts, we introduced **image similarity search** and reviewed a high-level architecture that can reduce complexity and facilitate implementation. This blog explains the underlying concepts and technical considerations for each component required to implement an image similarity search application. Learn more about:

1. **Embedding models:** **Machine learning models** that generate the numeric representation of your data needed to apply vector search
2. **Inference endpoint:** API to apply the embedding models to your data in Elastic
3. **Vector search:** How similarity search works with nearest neighbor search
4. **Generate image embeddings:** Scale generation of numeric representations to large data sets
5. **Application logic:** How the interactive front end communicates with the vector search engine on the back end

Diving into these five components gives you a blueprint of how you can implement more intuitive search experiences applying vector search in Elastic.



1. Embedding models

To apply similarity search to natural language or image data, you need machine learning models that translate your data to its numeric representation, also known as vector embeddings. In this example:

- The NLP "transformer" model translates natural language into a vector.
- OpenAI CLIP (Contrastive Language-Image Pre-training) model vectorizes images.

Transformer models are machine learning models trained to process natural language data in various ways, such as language translation, text classification, or named entity recognition. They are trained on extremely large data sets of annotated text data to learn the patterns and structures of human language.

The image similarity application finds images matching given textual, natural language descriptions. To implement that kind of similarity search, you need a model that was trained on both text and images and can translate the text query into a vector. This can then be used to find similar images.

Learn more about how to upload and use the NLP model in Elasticsearch >>

CLIP is a large-scale language model developed by OpenAI that can handle both text and images. The model is trained to predict the textual representation of an image, given a small piece of text as input. This involves learning to align the visual and textual representations of an image in a way that allows the model to make accurate predictions.

Another important aspect of CLIP is that it is a "zero-shot" model, which allows it to perform tasks it has not been specifically trained on. For example, it can translate between languages it has not seen during training or classify images into categories it has not seen before. This makes CLIP a very flexible and versatile model.

You will use the CLIP model to vectorize your images, using the inference endpoint in Elastic as described next and executing inference on a large set of images as described in section 3 further below.

2. The inference endpoint

Once the NLP model is loaded into Elasticsearch, you can process an actual user query. First, you need to translate the text of the query into a vector using the Elasticsearch _infer endpoint. The endpoint provides a built-in method of using the NLP model natively in Elastic and does not require querying an external service, significantly simplifying the implementation.

```
POST _ml/trained_models/sentence-transformers__clip-vit-b-32-multilingual-v1/depl
{
  "docs" : [
    {"text_field": "A mountain covered in snow"}
    ]
}
```

Copy

3. Vector (similarity) search

After indexing both queries and documents with vector embeddings, similar documents are the nearest neighbors of your query in embedding space. One popular algorithm to achieve that is k-nearest neighbor (kNN), which finds the k nearest vectors to a query vector. However, on the large data sets you'd typically process in image search applications, kNN requires very high computational resources and can lead to excessive execution times. As a solution, **approximate nearest neighbor (ANN)** search sacrifices perfect accuracy in exchange for executing efficiently in high dimensional embedding spaces, at scale.

In Elastic, the _search endpoint supports both exact and approximate nearest neighbor searches. Use the code below for the kNN search. It assumes the embeddings for all the images in `your-image-index` are available in the `image_embedding` field. The next section discusses how you can create the embeddings.

```
#  Run kNN search against <query-embedding> obtained above
POST <your-image-index>/_search
{
  "fields": [...],
  "knn": {
    "field": "image_embedding",
    "k": 5,
    "num_candidates": 10,
    "query_vector": <query-embedding>
  }
}
```

Copy

To learn more about kNN in Elastic, refer to our documentation: **https://www.elastic.co/guide/en/elasticsearch/reference/current/knn-search.html**.

4. Generate image embeddings

The image embeddings mentioned above are critical for good performance of your image similarity search. They should be stored in a separate index that holds the image embeddings, which is referred to as `you-image-index` in the code above. The index consists of a document per image together with fields for the context and the dense vector (image embedding) interpretation of the image. Image embeddings represent an image in a lower-dimensional space. Similar images are mapped to nearby points in this space. The raw image can be several MB large, depending on its resolution.

The specific details of how these embeddings are generated can vary. In general, this process involves extracting features from the images and then mapping them to a lower-dimensional space using a mathematical function. This function is typically trained on a large data set of images to learn the best way to represent the features in the lower-dimensional space. Generating embeddings is a one-time task.

In this blog, we'll employ the **CLIP model** for this purpose. It is distributed by OpenAI and provides a good starting point. You may need to train a custom embedding model for specialized use cases to achieve desired performance, depending how well the types of images you want to classify are represented in the publicly available data used to train the CLIP model.

Embedding generation in Elastic needs to occur at ingest time, and therefore in a process external to the search, with the following steps:

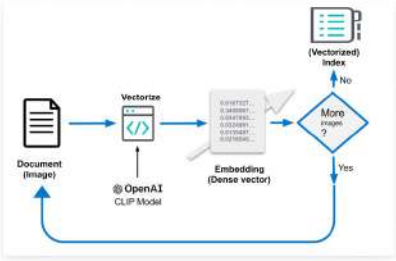
1. Load the CLIP model.
2. For every image:
    1. Load the image.
    2. Evaluate the image using the model.
    3. Save the generated embeddings into a document.
    4. Save the document into the datastore/Elasticsearch.

The pseudo code makes these steps more concrete, and you can access the full **code** in the example repository.

```
...
img_model = SentenceTransformer('clip-ViT-B-32')
...
for filename in glob.glob(PATH_TO_IMAGES, recursive=True):
    doc = {}
    image = Image.open(filename)
    embedding = img_model.encode(image)
    doc['image_name'] = os.path.basename(filename)
    doc['image_embedding'] = embedding.tolist()
    lst.append(doc)
...
```

Copy

Or refer to the figure below as an illustration:



The document after processing might look like the following. The critical part is the field `"image_embedding"` where the dense vector representation is stored.

```
{
  "_index": "my-image-embeddings",
  "_id": "_g9ACIU8MEjlQge4tztV",
  "_score": 6.703597,
  "_source": {
    "image_id": "IMG_4032",
    "image_name": "IMG_4032.jpeg",
    "image_embedding": [
      -0.3415695130825043,
      0.1906963288784027,
      .....
      -0.10289803147315979,
      -0.15871885618881207
```

Copy

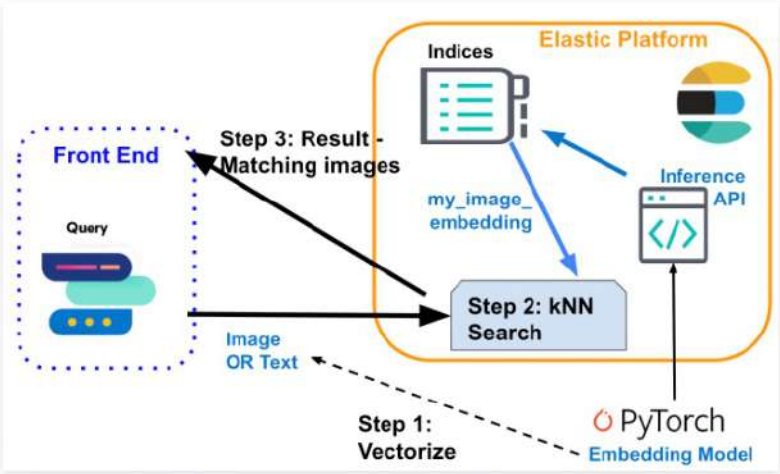
5. The application logic

Building on these basic components, you can finally put all pieces together and work through the logic to implement an interactive image similarity search. Let's start conceptually, with what needs to happen when you want to interactively retrieve images that match a given description.

For textual queries, the input can be as simple as a single word like roses or a more extended description like "a mountain covered in snow." Or you can also provide an image and ask for similar images to the one you have.

Even though you are using different modalities to formulate your query, both are executed using the same sequence of steps in the underlying vector search, namely using a query (kNN) over documents represented by their embeddings (as "dense" vectors). We have described the mechanisms in earlier sections that enable Elasticsearch to execute very fast and scalable vector search necessary on large image data sets. Refer to **this documentation** to learn more about tuning kNN search in Elastic for efficiency.

- So how can you implement the logic described above? In the flow diagram below you can see how information flows: The query issued by the user, as text or image, is vectorized by the embedding model — depending on input type: an NLP model for text descriptions, whereas the CLIP model for images.
- Both convert the input query into their numeric representation and store the result into a dense vector type in Elasticsearch ([number, number, number...]).
- The vector representation is then used in a kNN search to find similar vectors (images), which are returned as result.



Flow

Inference: Vectorize user queries

The application in the background will send a request to the inference API in Elasticsearch. For text input, something like this:

```
POST _ml/trained_models/sentence-transformers__clip-vit-b-32-multilingual-v1/depl
{
  "docs" : [
    {"text_field": "A mountain covered in snow"}
    ]
}
```

Copy

For images, you can use below simplified code to process a single image with CLIP model, which you needed to load into your Elastic machine learning node ahead of time:

```
model = SentenceTransformer('clip-ViT-B-32')
image = Image.open(file_path)
embedding = model.encode(image)
```

Copy

p.t.o.

You will get back a 512-long array of Float32 values, like this:

```
{
  "predicted_value" : [
    -0.26385045051574707,
    0.14752596616744995,
    0.4033305048942566,
    0.22902603447437286,
    -0.15598160028457642,
    ...
  ]
}
```

Copy

Search: For similar images

Searching works the same for both types of input. Send the query with `kNN search definition against the index with image embeddings my-image-embeddings. Put in the dense vector from the previous query ("query_vector": [ ... ]) and execute the search.`

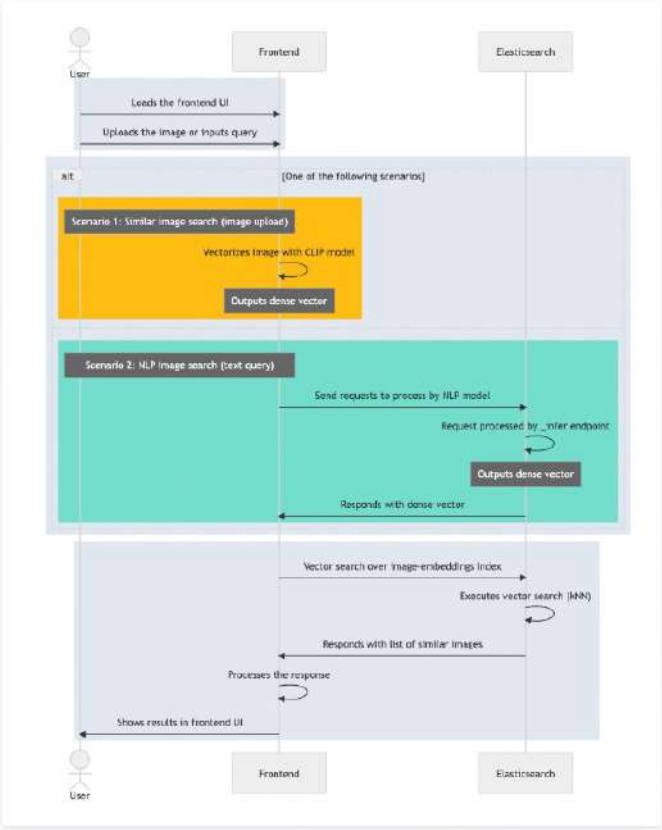
```
GET my-image-embeddings/_search
{
  "knn": {
    "field": "image_embedding",
    "k": 5,
    "num_candidates": 10,
    "query_vector": [
    -0.19898493587970734,
    0.1074572503566742,
    -0.05087625980377197,
    ...
    0.08200495690107346,
    -0.07852292060852051
```

Copy

query, stored in Elastic as documents.

The flow graph below summarizes the steps your interactive application moves through while processing a user query:

1. Load the interactive application, its front end.
2. The user selects an image they're interested in.
3. Your application vectorizes the image by applying the CLIP model, storing the resulting embedding as dense vector.
4. The application initiates a kNN query in Elasticsearch, which takes the embedding and returns its nearest neighbors.
5. Your application processes the response and renders one (or more) matching images.



Now that you understand the main components and information flow required to implement an interactive image similarity search, you can walk through the final part of the series to learn how to make it happen. You'll get a step-by-step guide on how to set up the application environment, import the NLP model, and finally complete the image embedding generation. Then you will be able to search through images with natural language — no keywords required.

Part 2. How to implement image similarity search in Elasticsearch

→ Create machine in elasticsearch cluster

↓

upload the CLIP model from OpenAI using Eland library

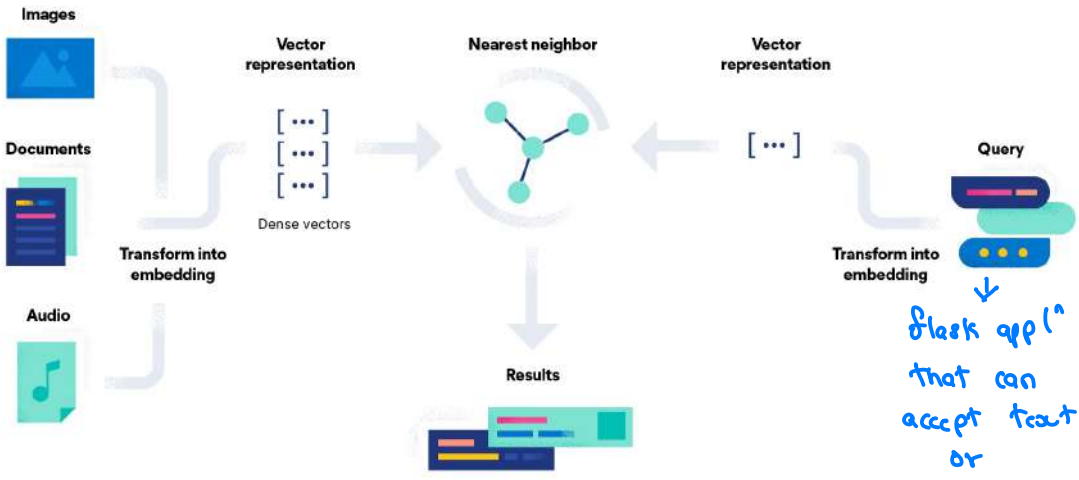
(Get elastic search endpoint URL. And using that URL upload model by Eland import command)

↓

Make model up & running

↓

Create image embeddings



Create image embedings with simple python script

↓

store in database of all image dataset

↓

Use flask application to Search images

Text similarity search with vector fields

A straightforward approach to similarity search would be to rank documents based on how many words they share with the query. But a document may be similar to the query even if they have very few words in common — a more robust notion of similarity would take into account its syntactic and **semantic** content as well.

The natural language processing (NLP) community has developed a technique called text embedding that encodes words and sentences as numeric vectors. These vector representations are designed to capture the linguistic content of the text, and can be used to assess similarity between a query and a document.

This post explores how text embeddings and Elasticsearch's dense_vector type could be used to support similarity search. We'll first give an overview of embedding techniques, then step through a simple prototype of similarity search using Elasticsearch.

What are text embeddings?

Let's take a closer look at different types of text embeddings, and how they compare to traditional search approaches.

Word embeddings

A **word embedding** model represents a word as a dense numeric vector. These vectors aim to capture semantic properties of the word — words whose vectors are close together should be similar in terms of semantic meaning. In a good embedding, directions in the vector space are tied to different aspects of the word's meaning. As an example, the vector for "Canada" might be close to "France" in one direction, and close to "Toronto" in another.

The NLP and search communities have been interested in vector representations of words for quite some time. There was a resurgence of interest in word embeddings in the past few years, when many traditional tasks were being revisited using neural networks. Some successful word embedding algorithms were developed, including **word2vec** and **GloVe**. These approaches make use of large text collections, and examine the context each word appears in to determine its vector representation:

- The word2vec Skip-gram model trains a neural network to predict the context words around a word in a sentence. The internal weights of the network give the word embeddings.
- In GloVe, the similarity of words depends on how frequently they appear with other context words. The algorithm trains a simple linear model on word co-occurrence counts.

Sentence embeddings

More recently, researchers have started to focus on embedding techniques that represent not only words, but longer sections of text. Most current approaches are based on complex neural network architectures, and sometimes incorporate labelled data during training to aid in capturing semantic information.

Once trained, the models are able to take a sentence and produce a vector for each word in context, as well as a vector for the entire sentence. Similarly to word embedding, pre-trained versions of many models are available, allowing users to skip the expensive training process. While the training process can be very resource-intensive, invoking the model is much more lightweight — sentence embedding models are typically fast enough to be used as part of real-time applications.

Some common sentence embedding techniques include **InferSent**, **Universal Sentence Encoder**, **ELMo**, and **BERT**. Improving word and sentence embeddings is an active area of research, and it's likely that additional strong models will be introduced.

Comparison to traditional search approaches

In traditional information retrieval, a common way to represent text as a numeric vector is to assign one dimension for each word in the vocabulary. The vector for a piece of text is then based on the number of times each term in the vocabulary appears. This way of representing text is often referred to as "bag of words," because we simply count word occurrences without regard to sentence structure.

Text embeddings differ from traditional vector representations in some important ways:

- The encoded vectors are dense and relatively low-dimensional, often ranging from 100 to 1,000 dimensions. In contrast, bag of words vectors are sparse and can comprise 50,000+ dimensions. Embedding algorithms encode the text into a lower-dimensional space as part of modeling its semantic meaning. Ideally, synonymous words and phrases end up with a similar representation in the new vector space.
- Sentence embeddings can take the order of words into account when determining the vector representation. For example the phrase "tune in" may be mapped as a very different vector than "in tune".
- In practice, sentence embeddings often don't generalize well to large sections of text. They are not commonly used to represent text longer than a short paragraph.

Using embeddings for similarity search

Let's suppose we had a large collection of questions and answers. A user can ask a question, and we want to retrieve the most similar question in our collection to help them find an answer.

We could use text embeddings to allow for retrieving similar questions:

- During indexing, each question is run through a sentence embedding model to produce a numeric vector.
- When a user enters a query, it is run through the same sentence embedding model to produce a vector. To rank the responses, we calculate the vector similarity between each question and the query vector. When comparing embedding vectors, it is common to use **cosine similarity**.

Implementation details

The script begins by downloading and creating the embedding model in TensorFlow. We chose Google's Universal Sentence Encoder, but it's possible to use many other embedding methods. The script uses the embedding model as-is, without any additional training or fine-tuning.

Next, we create the Elasticsearch index, which includes mappings for the question title, tags, and also the question title encoded as a vector:

```
"mappings": {
  "properties": {
    "title": {
      "type": "text"
    },
    "title_vector": {
      "type": "dense_vector",
      "dims": 512
    }
    "tags": {
      "type": "keyword"
    },
    ...
  }
}
```

In the mapping for dense_vector, we're required to specify the number of dimensions the vectors will contain. When indexing a title_vector field, Elasticsearch will check that it has the same number of dimensions as specified in the mapping.

To index documents, we run the question title through the embedding model to obtain a numeric array. This array is added to the document in the title_vector field.

When a user enters a query, the text is first run through the same embedding model and stored in the parameter query_vector. As of 7.3, Elasticsearch provides a **cosineSimilarity function** in its native scripting language. So to rank questions based on their similarity to the user's query, we use a script_score query:

```
{
  "script_score": {
    "query": {"match_all": {}},
    "script": {
      "source": "cosineSimilarity(params.query_vector, 'title_vector') + 1.0",
      "params": {"query_vector": query_vector}
    }
  }
}
```

We make sure to pass the query vector as a script parameter to **avoid recompiling the script**() on every new query. Since Elasticsearch does not allow negative scores, it's necessary to add one to the cosine similarity.

Important limitations

The script_score query is designed to wrap a restrictive query, and modify the scores of the documents it returns. However, we've provided a match_all query, which means the script will be run over all documents in the index. This is a current limitation of vector similarity in Elasticsearch — vectors can be used for scoring documents, but not in the initial retrieval step. Support for retrieval based on vector similarity is an important area of **ongoing work**.

To avoid scanning over all documents and to maintain fast performance, the match_all query can be replaced with a more selective query. The right query to use for retrieval is likely to depend on the specific use case.

While we saw some encouraging examples above, it's important to note that the results can also be noisy and unintuitive. For example, "zipping up files" also assigns high scores to "Partial .csproj Files" and "How to avoid .pyc files?". And when the method returns surprising results, it is not always clear how to debug the issue — the meaning of each vector component is often opaque and doesn't correspond to an interpretable concept. With traditional scoring techniques based on word overlap, it is often easier to answer the question "why is this document ranked highly?"

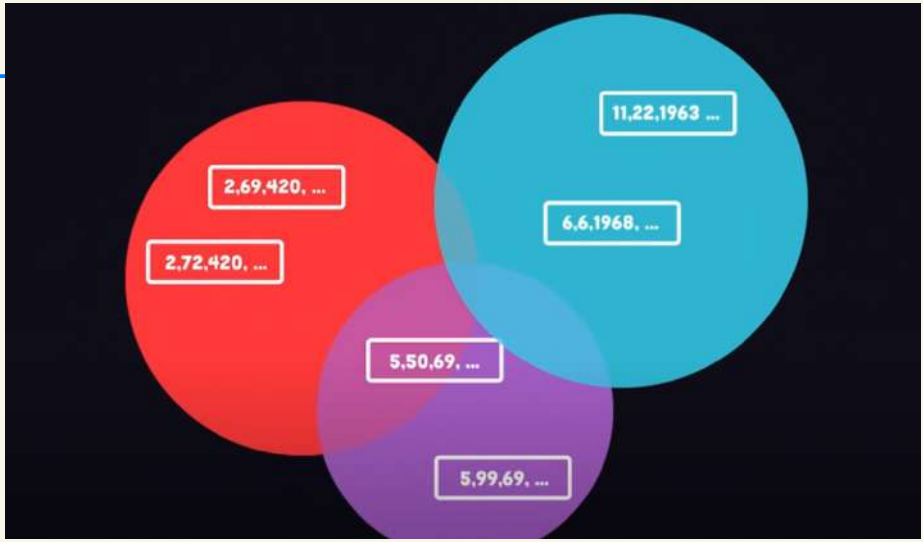
As mentioned earlier, this prototype is meant as an example of how embedding models could be used with vector fields, and not as a production-ready solution. When developing a new search strategy, it is critical to test how the approach performs on your own data, making sure to compare against a strong baseline like a match query. It may be necessary to make major changes to the strategy before it achieves solid results, including fine-tuning the embedding model for the target dataset, or trying different ways of incorporating embeddings such as word-level query expansion.

Conclusions

Embedding techniques provide a powerful way to capture the linguistic content of a piece of text. By indexing embeddings and scoring based on vector distance, we can compare documents using a notion of similarity that goes beyond their word-level overlap.

Vector Database

Vector databases are so hot right now. WTF are they?

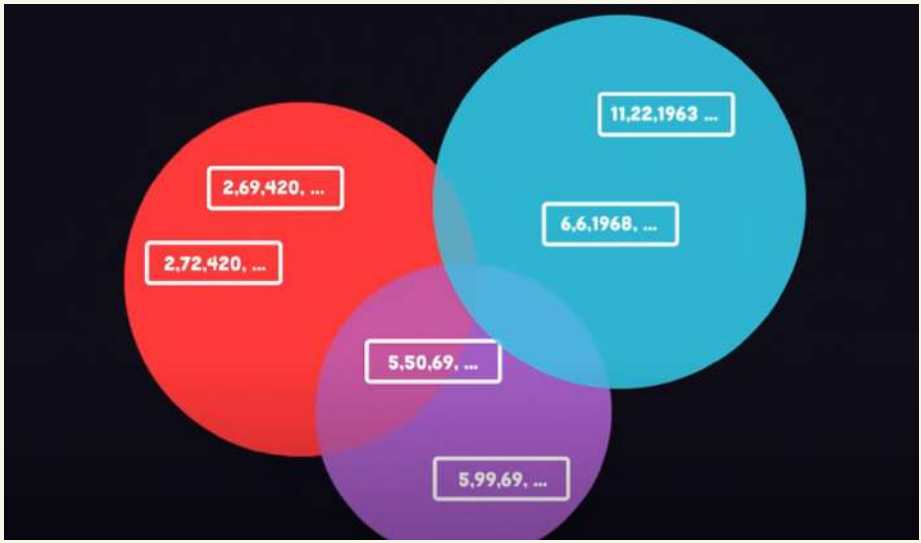


- It is array of numbers clustered together based on similarity. which can be queried with ultra-low latency

Types of Vector Database

- Weaviate
- Pinecone
- Chroma - an open source project
- Rektor - a local developer from Firestip youtube channel

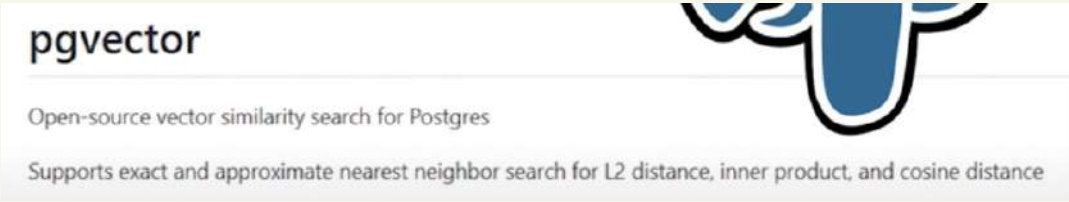
- Embeding working:



-> Similar arrays are grouped together like vector database

-> Except they map the semantic meaning of words together or similar features (here our words) or any other virtual data

→ Relational databases like postgres have tools like pg vector



→ Redis also has similar functionality like above

→ New databases are coming up like Weaviate, Milvus } Open source

→ Also Pinecone [closed source]
(its popular

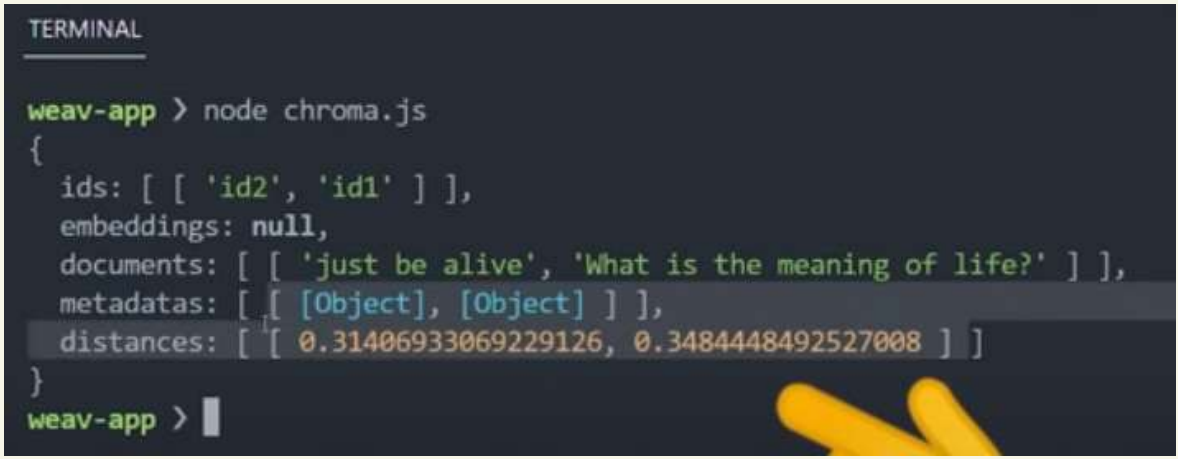
→ chroma, based on clickhouse
under the hood

→ Chroma example-

```
import { ChromaClient, OpenAIEmbeddingFunction } from "chromadb";
const client = new ChromaClient();

const embedder = new OpenAIEmbeddingFunction("MY-API-KEY")
const collection = await client.createCollection("my_collection", {}, embedder)

await collection.add(
    ["id1", "id2"], // ids
    undefined, // embeddings
    [{"source": "my_source"}, {"source": "my_source"}], // metadata
    ["What is the meaning of life?", "just be alive"], // documents
)


const results = await collection.query(
    undefined,
    2,
    undefined,
    ["Am I really alive?"]
)
```

QUERY WITH T

can query the database by passing a



- query result from previous image
→ here smaller no. indicates high similarity
- These are becoming because they can extend LLM to long-term memory

→ get user query

↓

Search in your vector database and get relevant documents

↓

update the context of user query based on relevant documents

↓

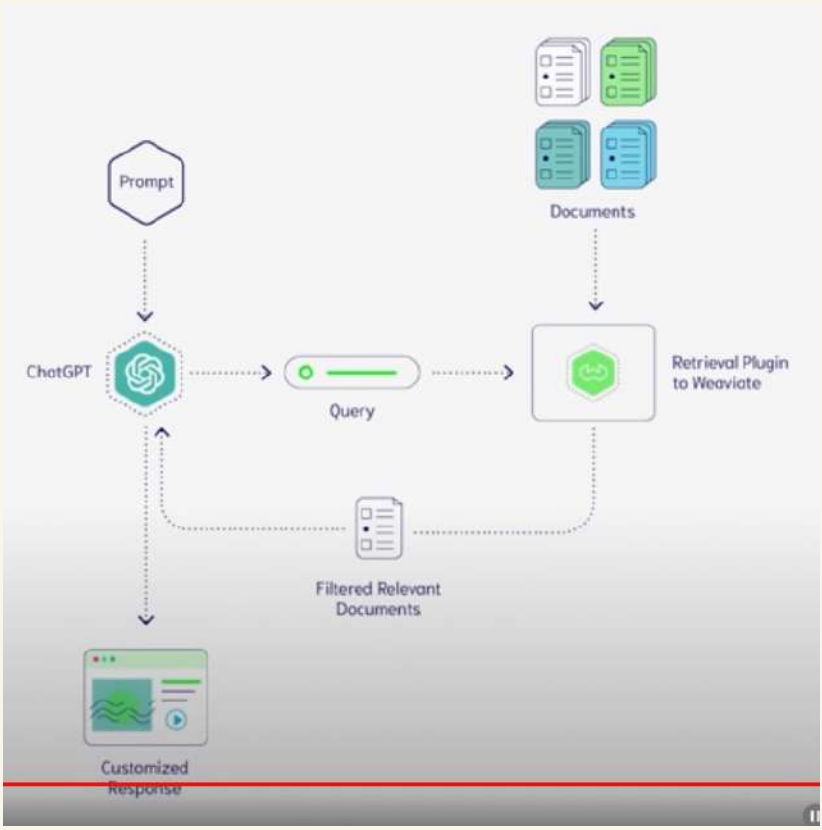
Send to any of these

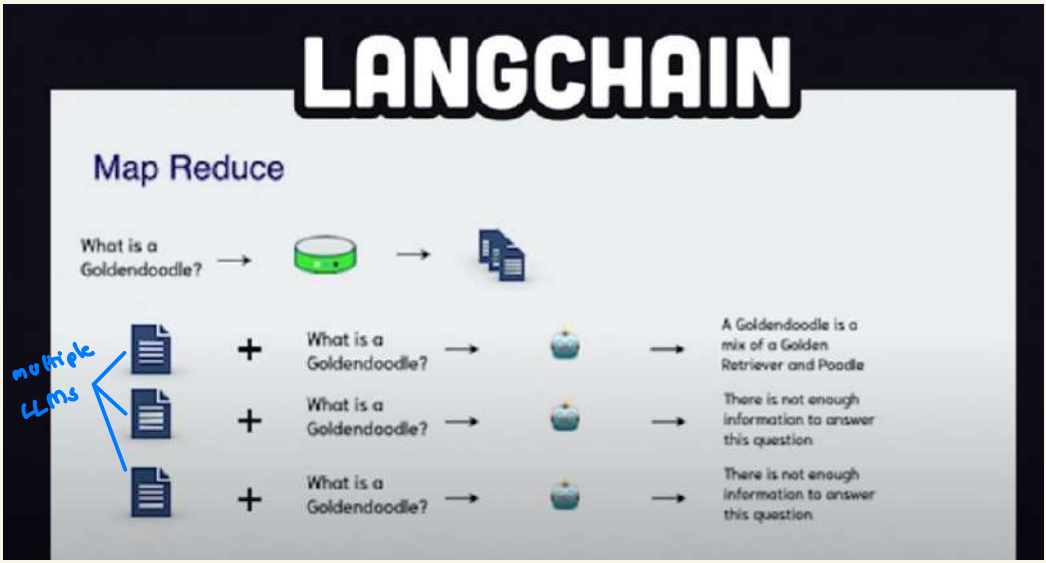
↓

get response

Advantages:-

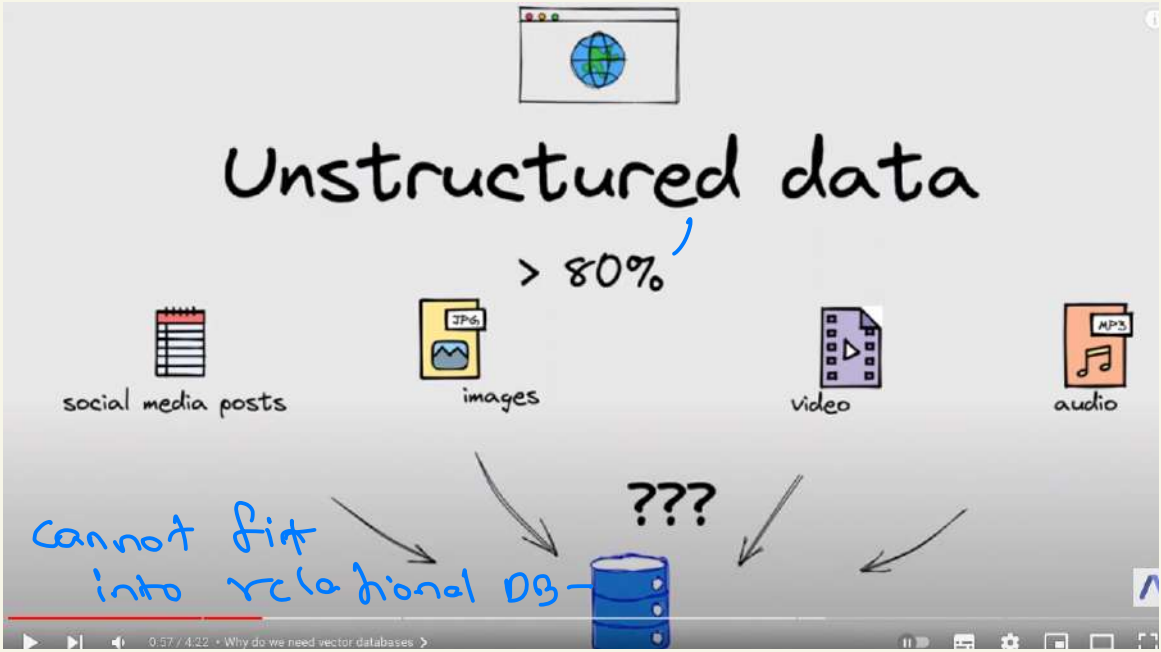
It can help retrive historical data to give AI long term memory

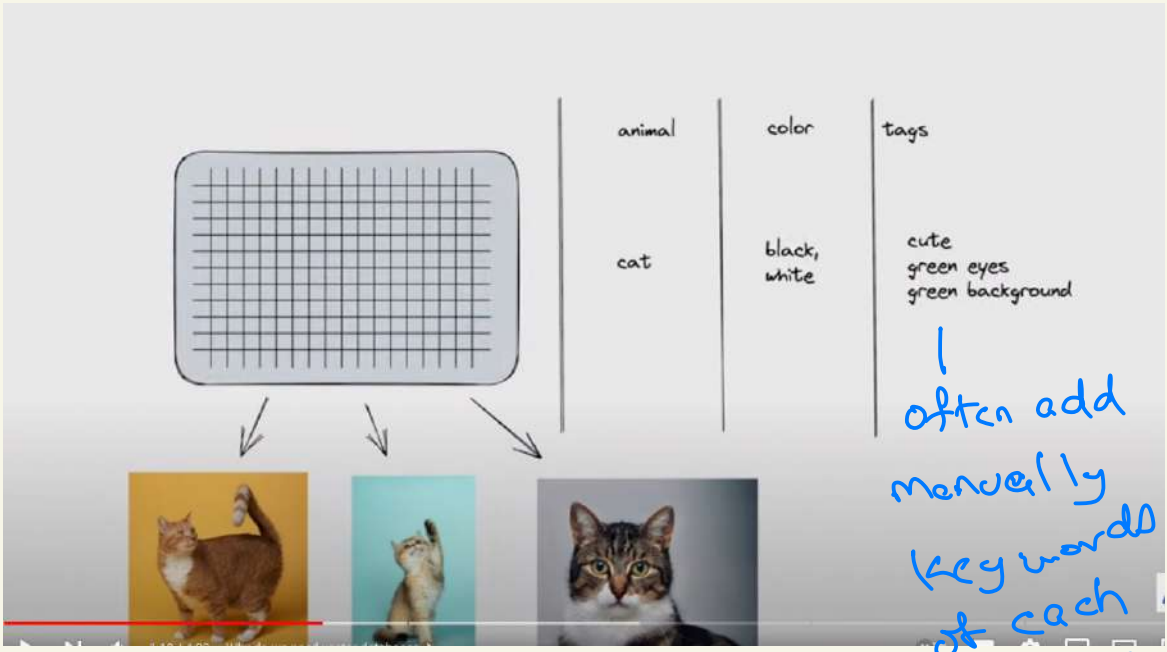




→ We can use Langchain further that combine multiple LLMs together

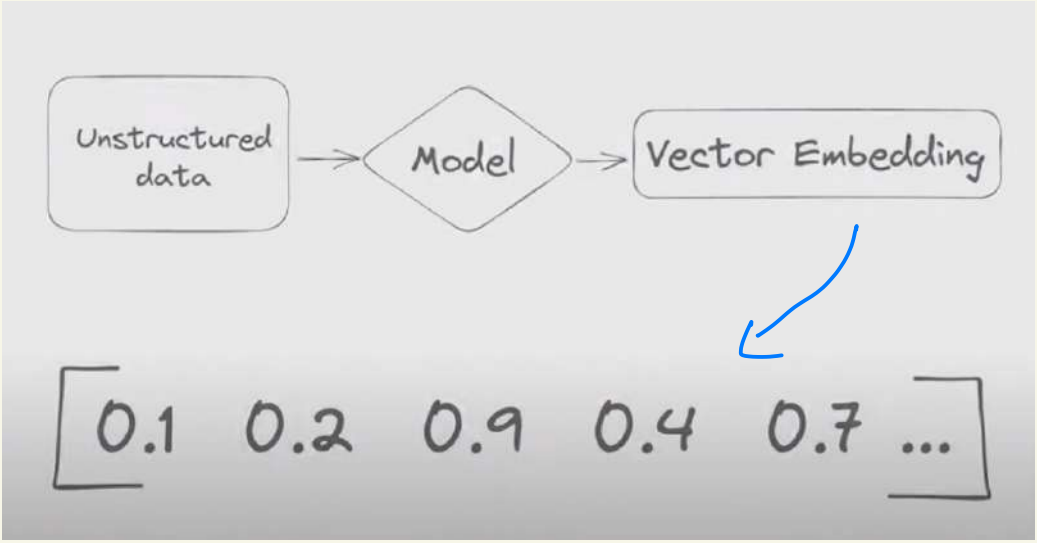
Vector Databases simply explained! (Embeddings & Indexes)

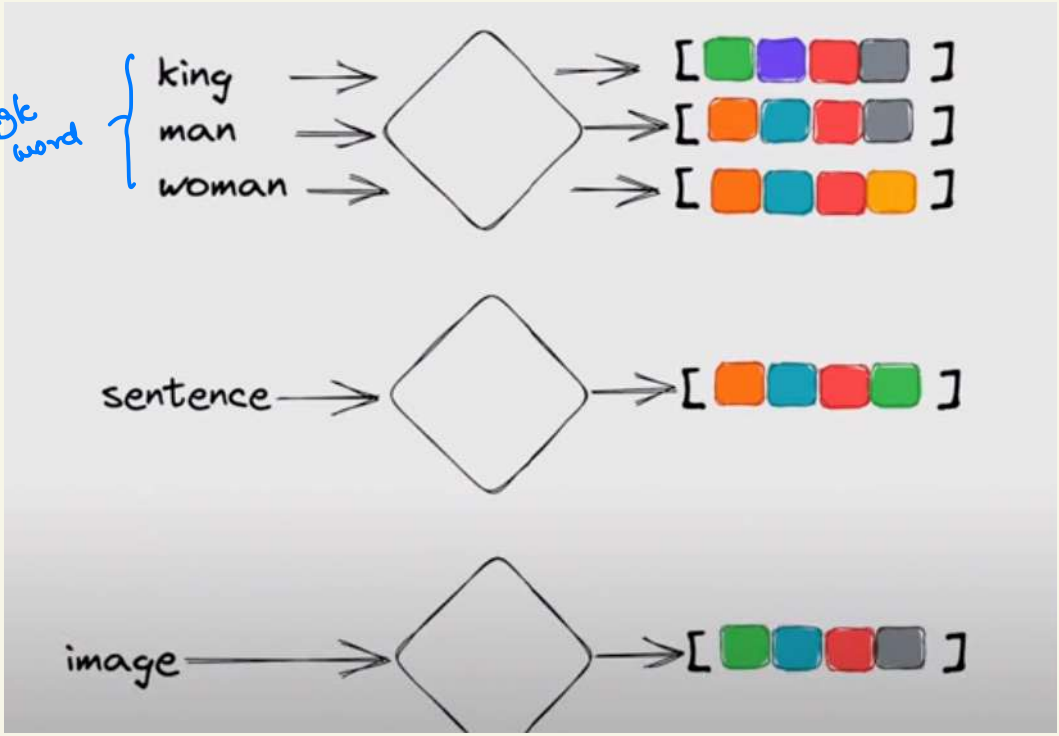




I often add manually keywords of each cat image in relational database

A vector database indexes and stores vector embeddings for fast retrieval and similarity search.

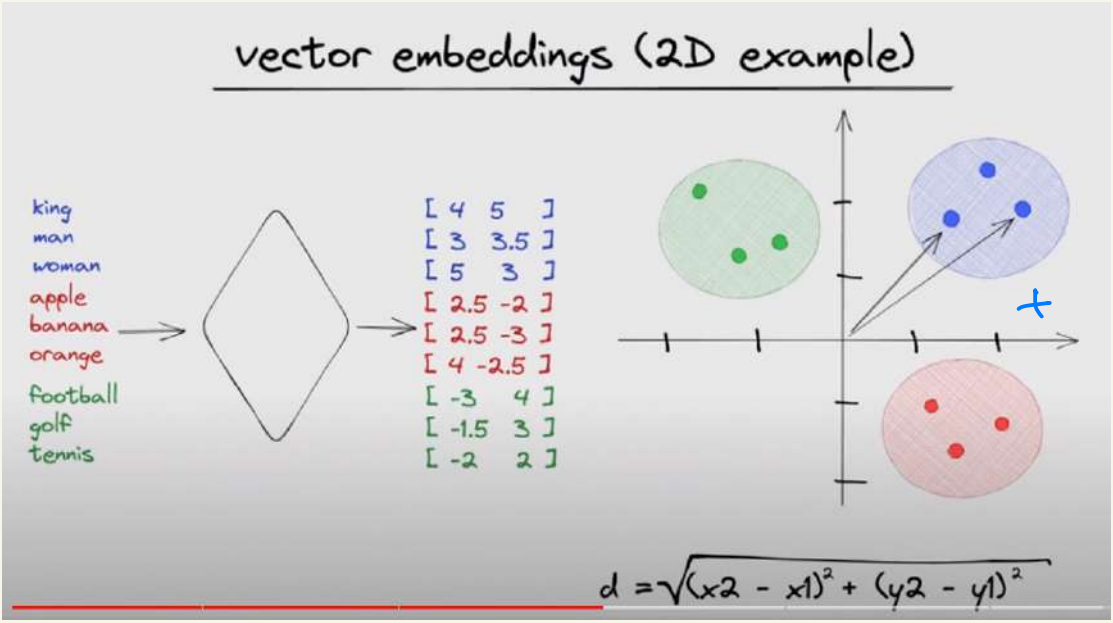




— We can create embeding

for single word or sentence

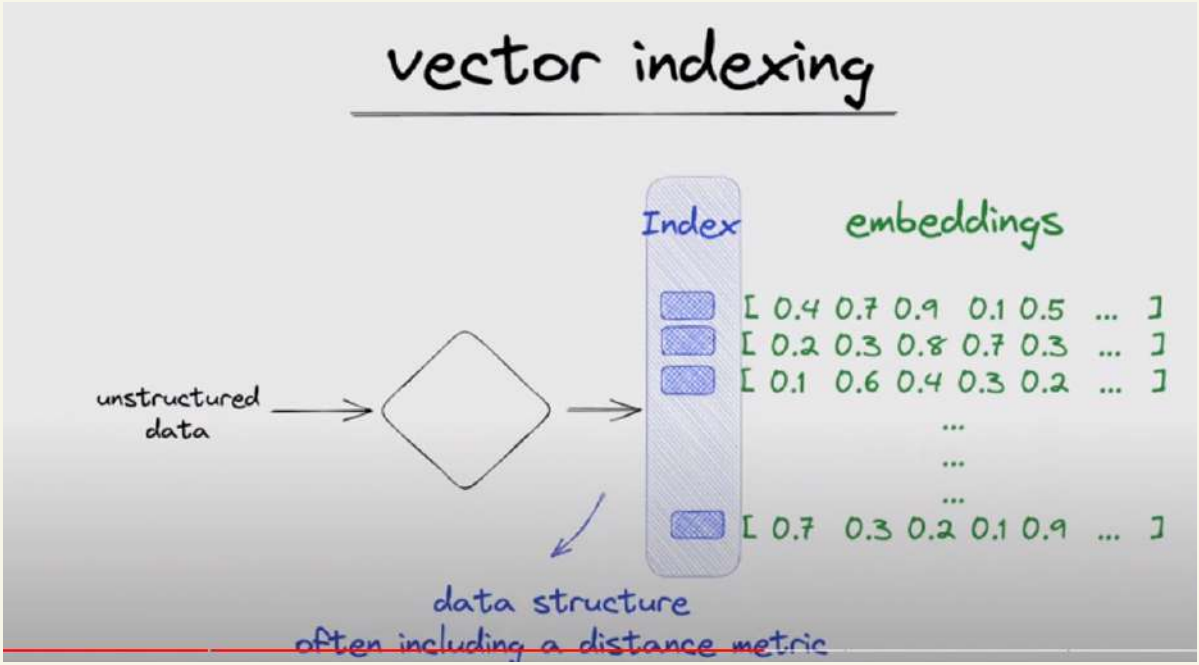
or image



- Suppose a new item 't' comes we can see which points are closer to t by nearest neighbour [Here blue ones]

→ Then once we got nearest neighbour we can find similar items [blue clustered]

→ querying a vector across thousands of vector is computational expensive as each distance are compared



→ Index is a data structure that facilitates search process

use cases

1. long-term memory for LLMs
2. semantic search: search based on the meaning or context
3. similarity search for text, images, audio, or video data
4. recommendation engine

Vector Databases

- Pinecone
- Qdrant
- Weaviate
- Milvus
- Chroma
- Vespa
- Redis

Vector DataBase fundamentals

https://www.youtube.com/playlist?list=PLPg7_faNDlT6wXMi2vfG0zJ6pK-gq6KE8

— Microsoft uses Approximate Nearest Neighbor

Weaviate

https://colab.research.google.com/github/semi-technologies/weaviate-examples/blob/main/harrypotter-qa-haystack-weaviate/COLAB-HarryPotter-QA-Haystack-Weaviate.ipynb

Weaviate - Vector Search Engine

Knowledge Graph Conference 2021

By Laura Ham, Community Solution Engineer at SeMI Technologies

Agenda

- Introduction to Weaviate: Vector Search Engine
- Demo
- Weaviate's core features
- Use cases

Weaviate

The AI-based vector search engine

Weaviate - why a vector search engine?

```
{ "data": [{
    "Wine": "Covey Run 2005 Chardonnay",
    "Description": "...  good with fish ..."
}]}
```

"Wine for seafood"

No products found ...

Traditional search engine

"Wine for seafood"

Covey Run 2005
Chardonnay

Vector search engine

The question is very **abstract.**

And how so extremely **fast**?

How did Google find exactly this **data node**?
How can we predict this **relation?**

How to do this on your **own data**?

SeMI

What's the problem?

- 80% to 90% of data is **unstructured**, that's about **34 trillion gigabytes**!
- We keep storing more and more data but businesses can't get **valuable insights** from it.
- 95% of businesses have issues **getting value** from their unstructured data.

How to achieve **search** and automatically make **relations** within your **own data**?

In a **easy**, **fast**, **secure** and **scalable** way?

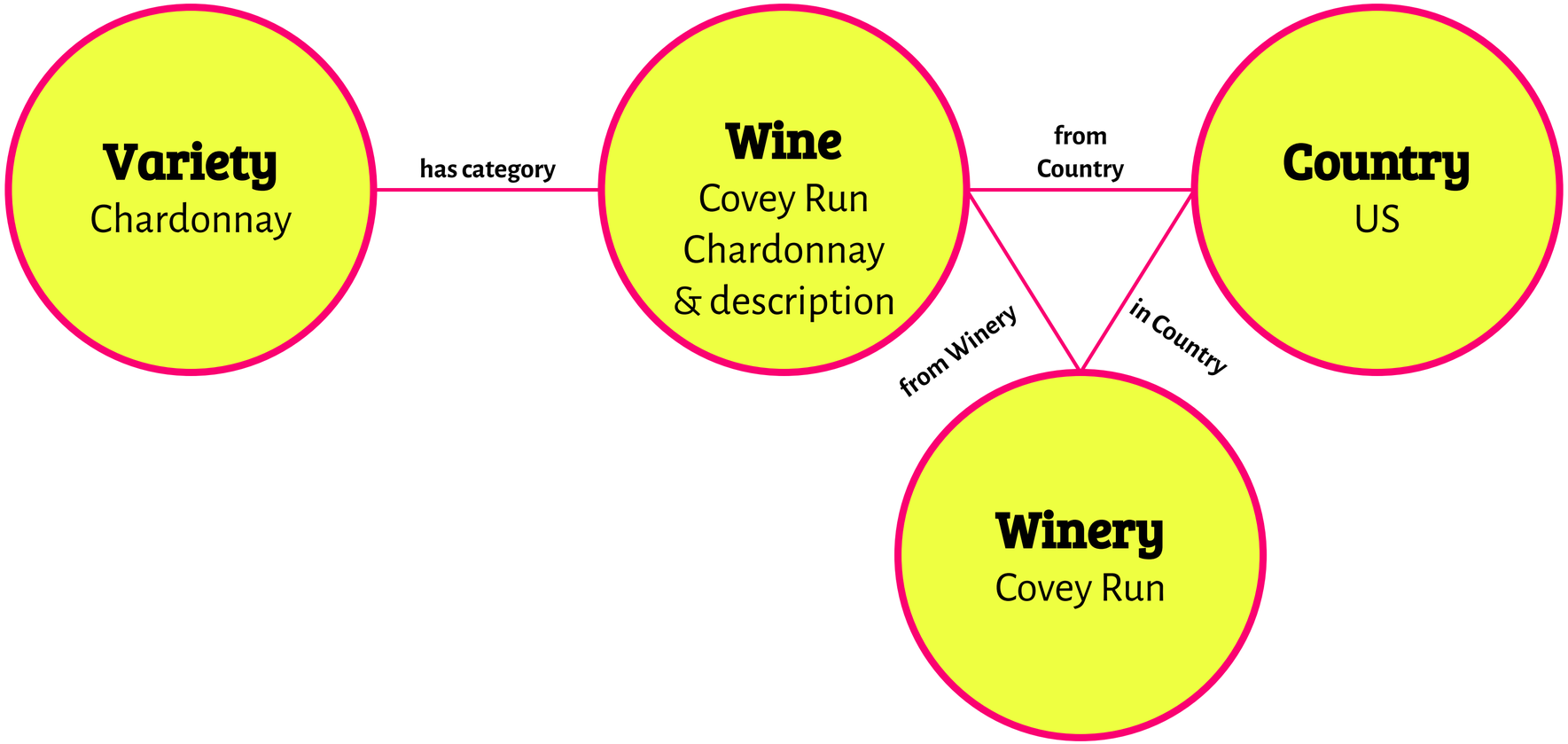
Weaviate is a cloud-native, modular,
real-time vector search engine
built to scale your machine learning models

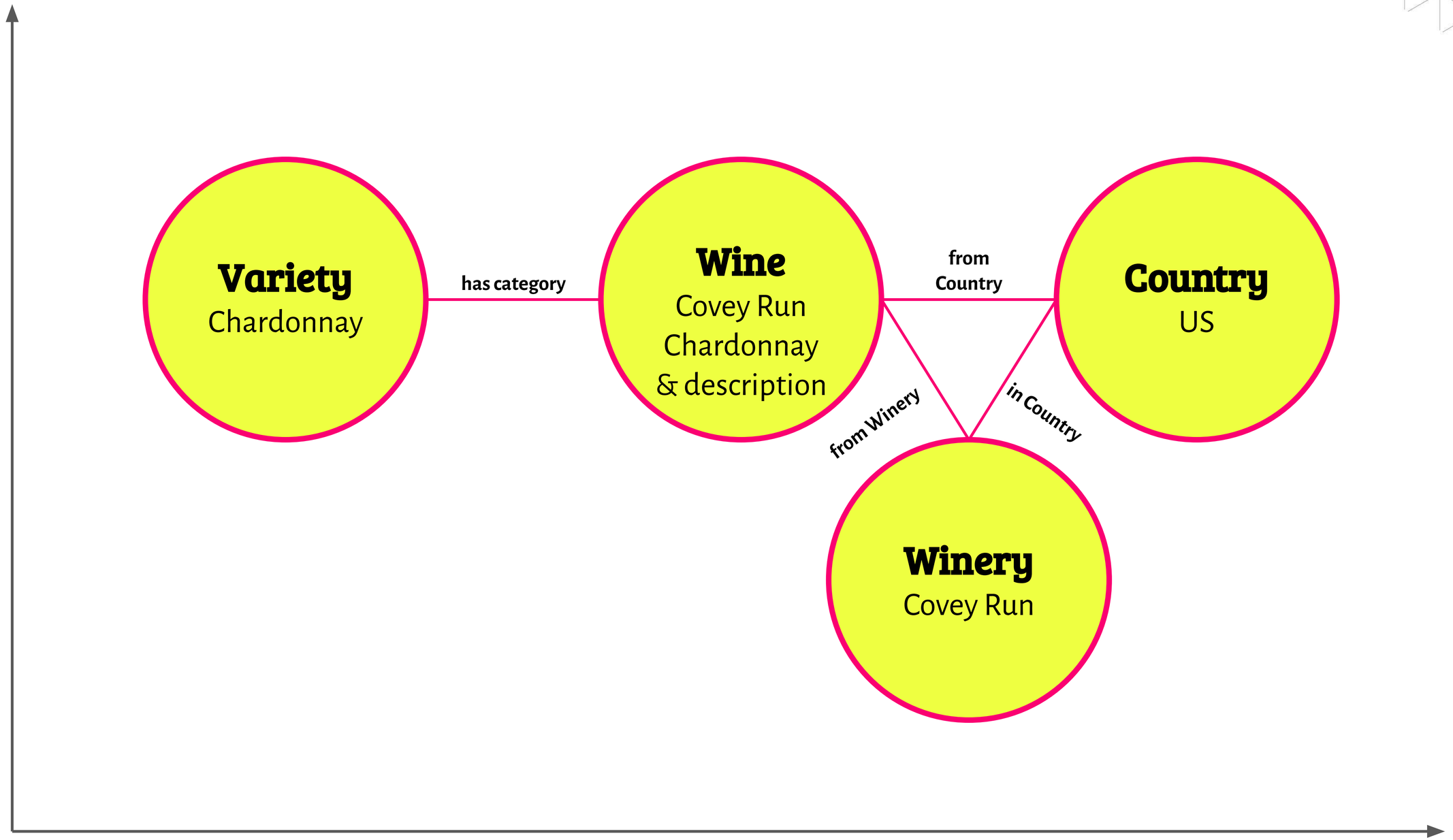
Weaviate is a cloud-native, modular,
real-time vector search engine
built to scale your machine learning models

Weaviate is a cloud-native, modular,
real-time vector search engine
built to scale your machine learning models

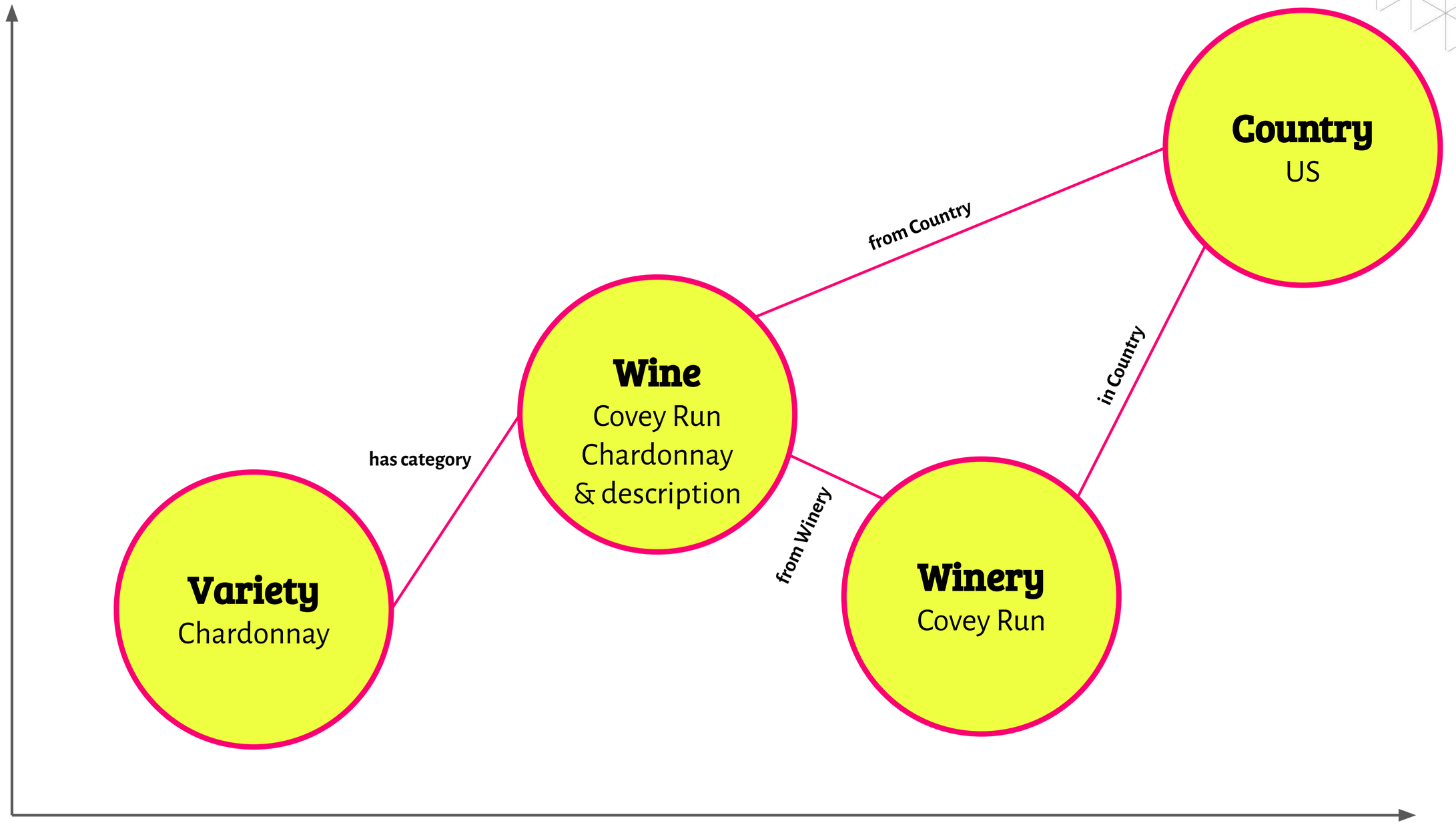
Weaviate is a cloud-native, modular,
real-time vector search engine
built to scale your machine learning models

Weaviate is a cloud-native, modular,
real-time vector search engine
built to scale your machine learning models

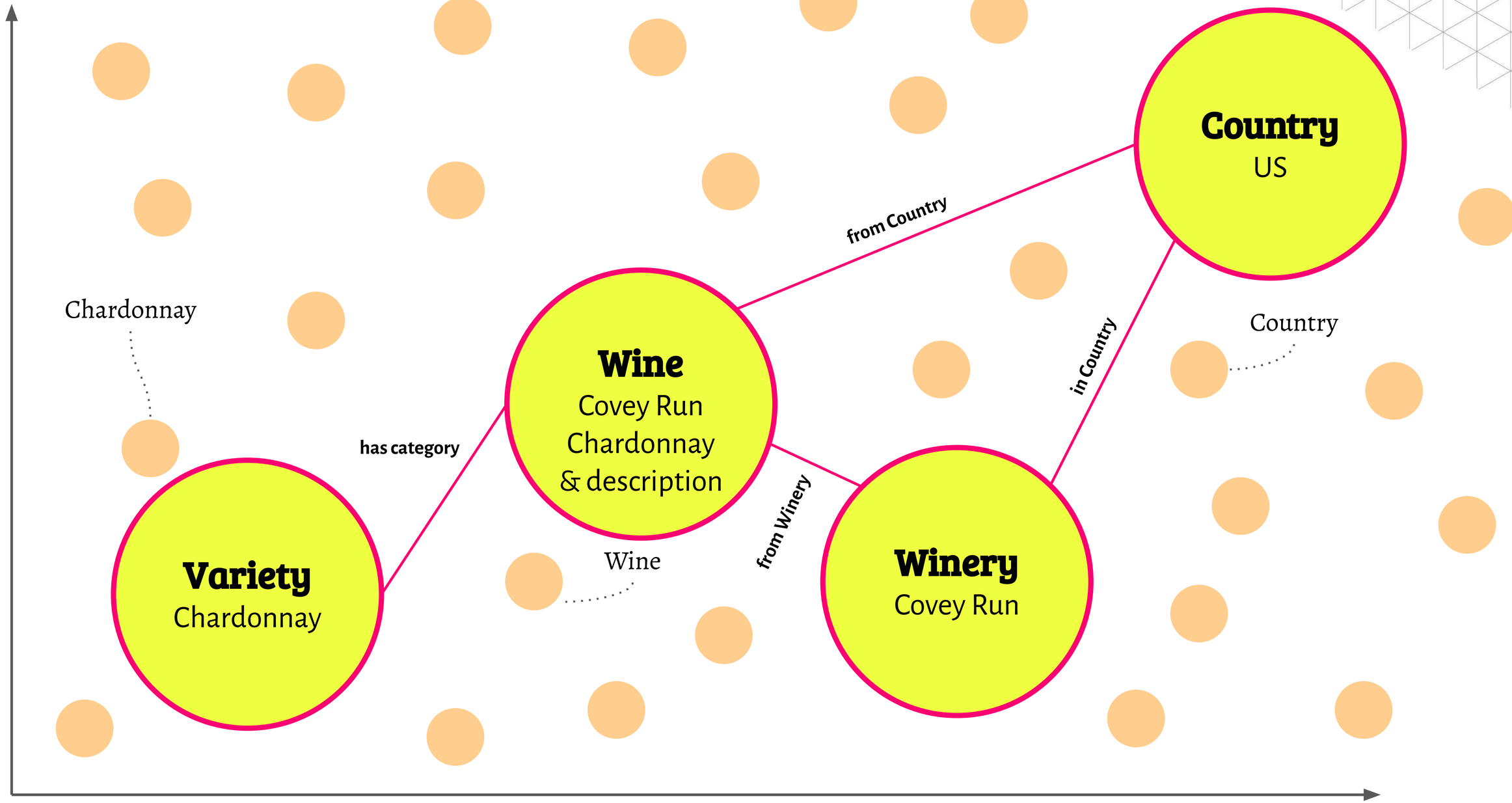

Variety
Chardonnay
has category
Wine
Covey Run
Chardonnay
& description
from
Country
Country
US
from Winery
in Country
Winery
Covey Run


Variety
Chardonnay
has category
Wine
Covey Run
Chardonnay
& description
from
Country
Country
US
from Winery
in Country
Winery
Covey Run

SeMI


Country
US
from Country
Wine
Covey Run
Chardonnay
& description
has category
in Country
from Winery
Variety
Chardonnay
Winery
Covey Run

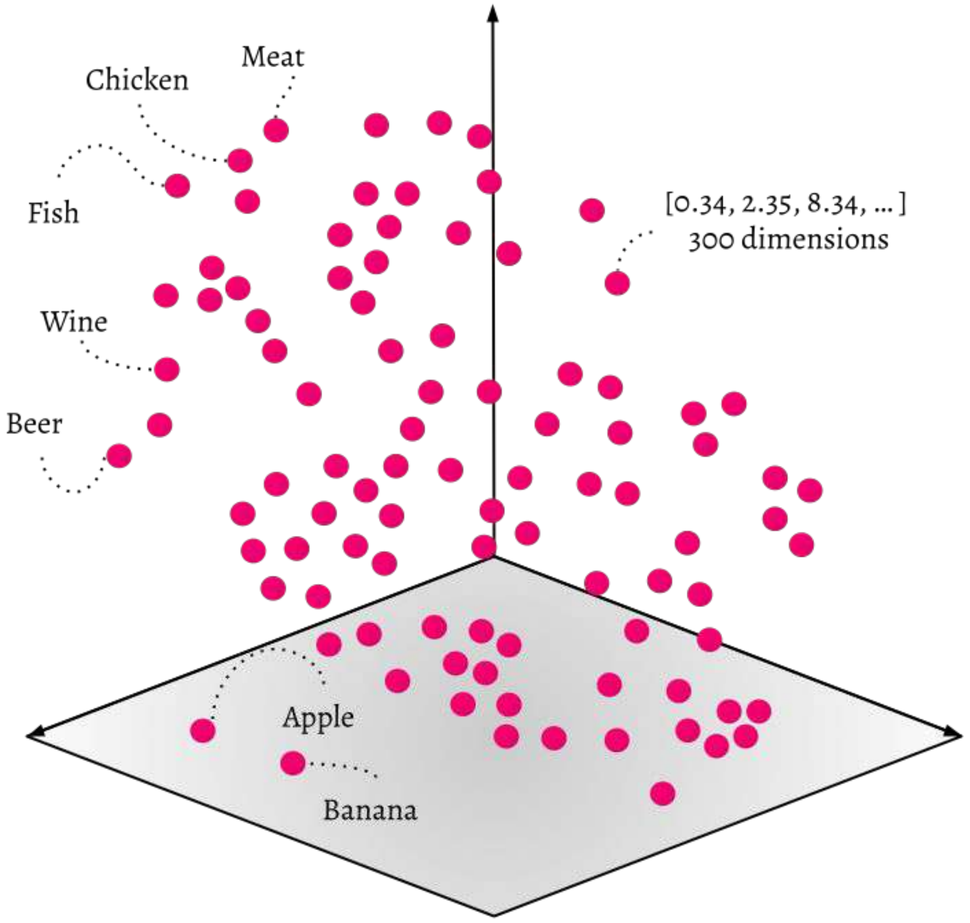
SeMI


Country
US
from Country
Chardonnay
Wine
Covey Run
Chardonnay
& description
in Country
Country
has category
from Winery
Variety
Chardonnay
Wine
Winery
Covey Run

Weaviate - How does it work?

- Data is stored as high dimensional vectors
- Vector positions capture data meaning and context
- Pre-trained NLP module for automatic
  - Vectorization
  - Classification
  - Nearest neighbor search

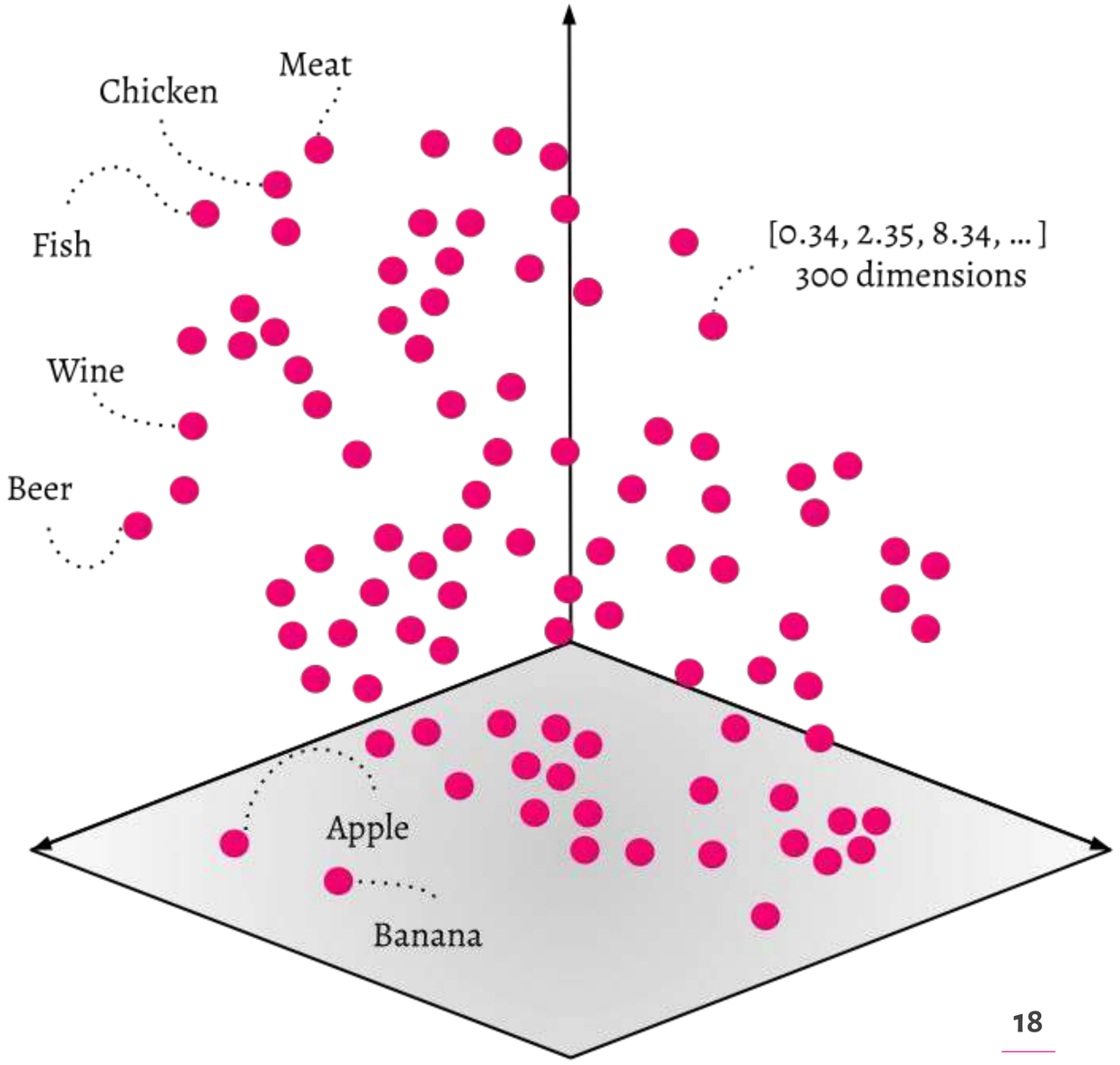
⇒ Weaviate *understands* the data



Weaviate - How does it work?

1. Weaviate Machine Learning Modules

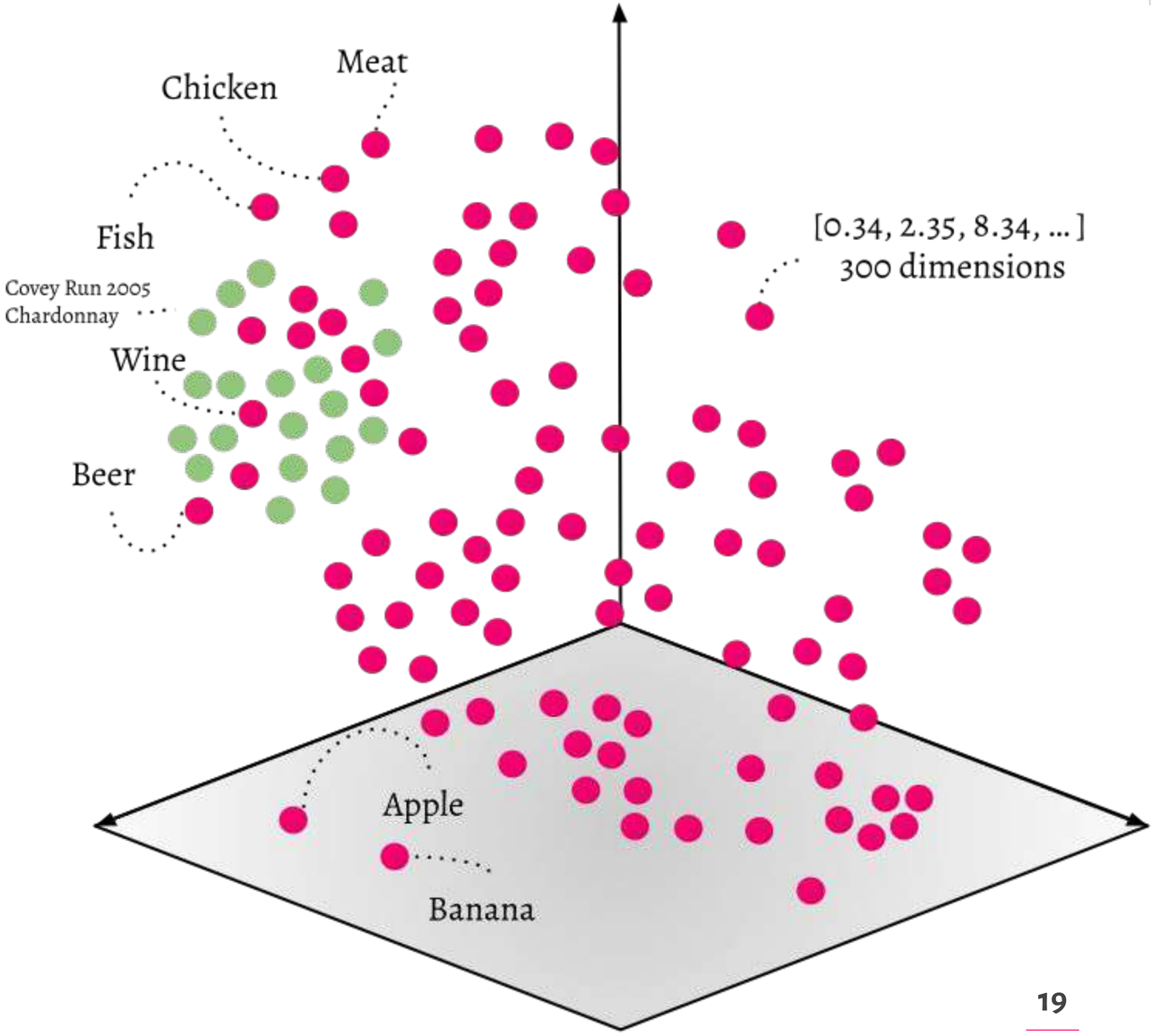
- E.g. NLP module trained with *fasttext*
- This language model represents all words and concepts in the hyperspace.



Weaviate - How does it work?

2. Automatically vectorize and index your data

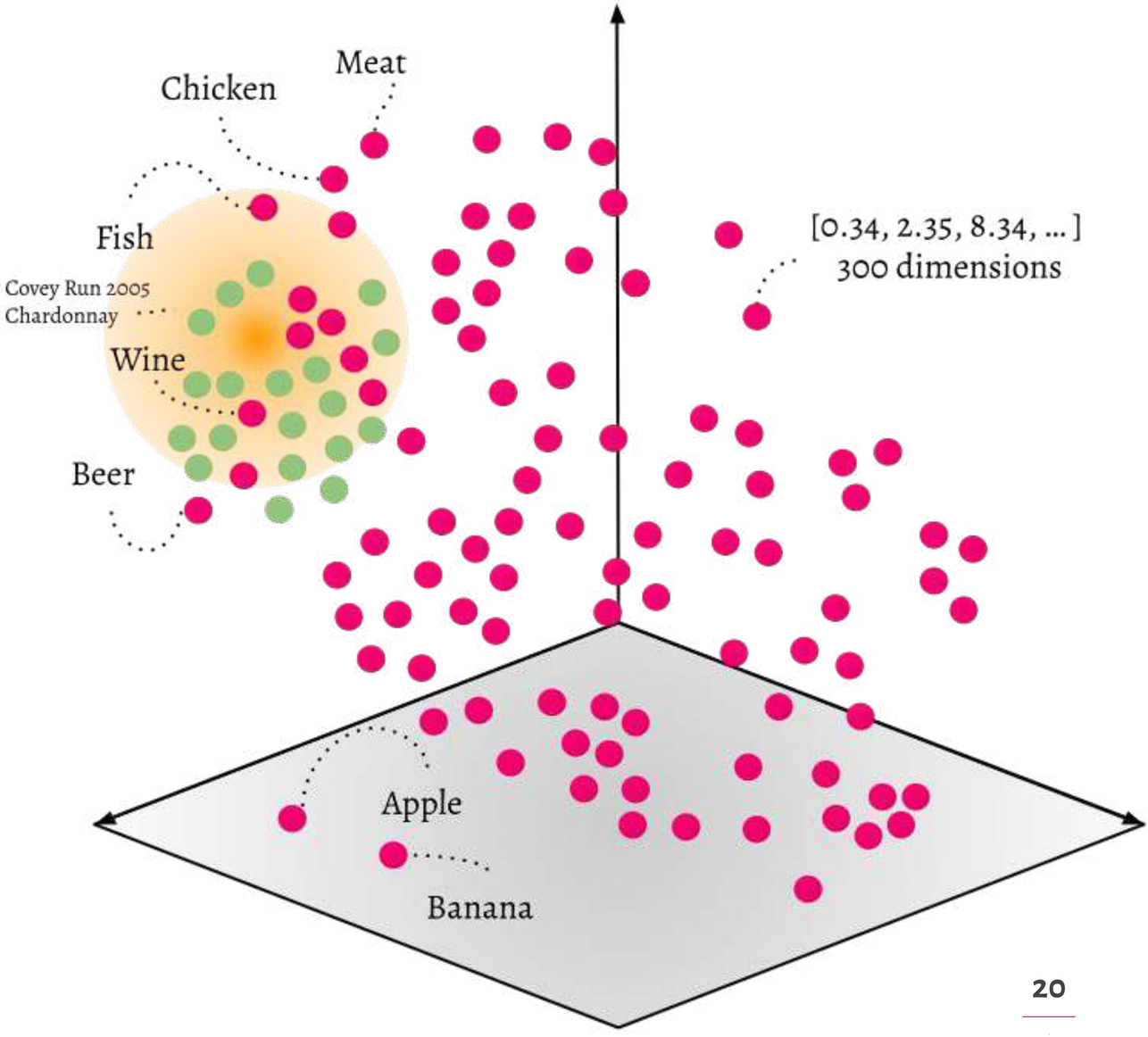
- Weaviate *understands* your data
- When you import data: Weaviate looks at the language in your data object
- E.g. a *Chardonnay* is closely related to *Wine*, *White* and the *food* it fits with



Weaviate - How does it work?

3. Search query

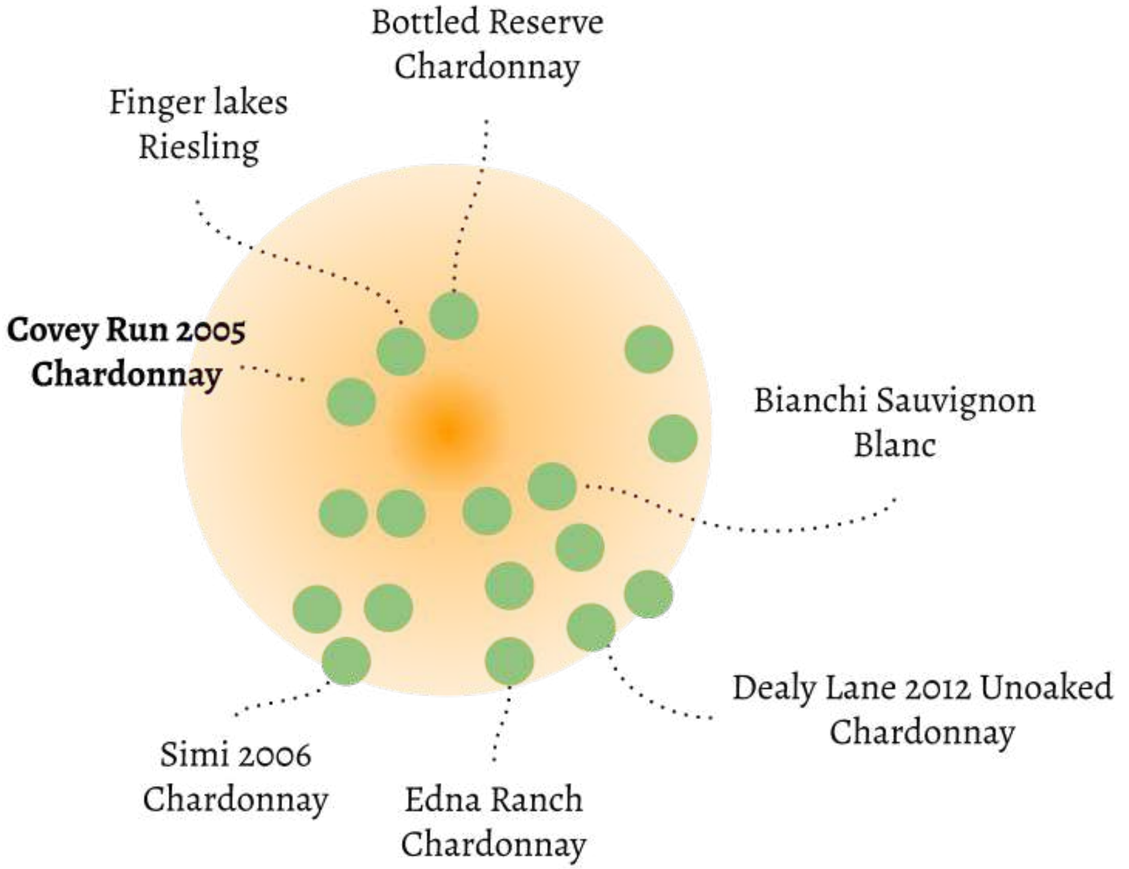
- Your search queries in natural language will also be vectorized and understood by the machine learning module of Weaviate
- It is places close to the words and data object that are semantically related to the query
- E.g. "*Wine that fits with a seafood dish*"



Weaviate - How does it work?

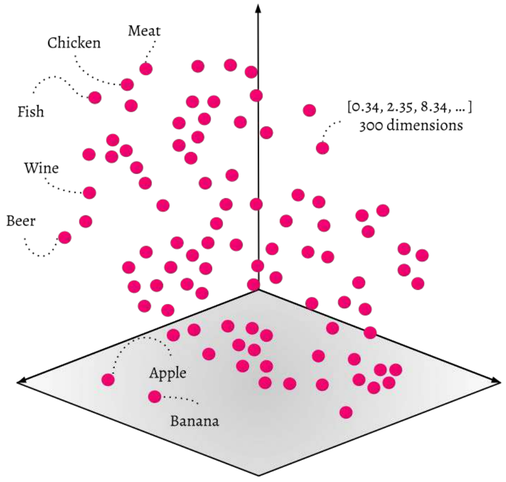
4. Results

- The data objects that are closest to the search query are retrieved from the dataset



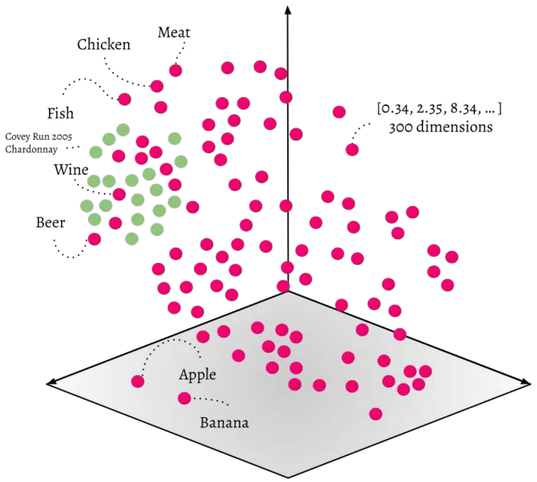
Weaviate - How does it work?

Pretrained ML model



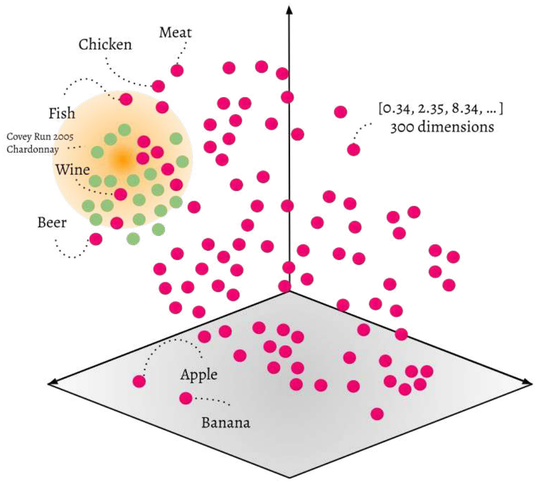
(e.g. NLP)

Your own data



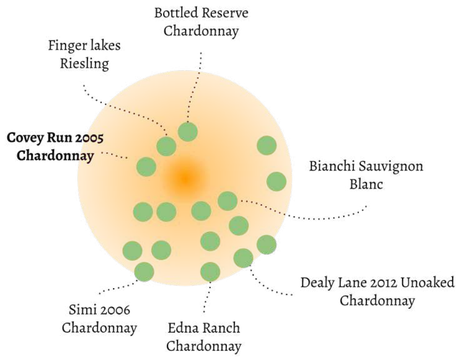
(e.g. Wines)

Your search query



(e.g. Wine for seafood)

Results



(e.g. Chardonnay)

SeMI

Demo

Weaviate - Scalable Vector Search Engine

Search

how to search the vector space

Weaviate Modules

out-of-the-box modules you can use
e.g. Weaviate NLP modules,
Transformer modules
Custom modules

Classification

how to classify in the vector space

SeMI

Weaviate Vectorization Modules

Machine learning models

- **Weaviate's NLP module: trained with *fasttext***
  - Contextionary, in multiple languages
- **Transformer modules: general-purpose NLP architectures**
  - e.g. BERT, GPT-2, RoBERTa, XLM, DistilBert, etc
- **Custom modules**

Weaviate - Scalable Vector Search Engine

Search

how to search the vector space

Weaviate Modules

out-of-the-box modules you can use
e.g. Weaviate NLP modules,
Transformer modules
Custom modules

End-to-end

Complete solution for industry,
API driven
Cloud-native
Secure

Classification

how to classify in the vector space

Open Source

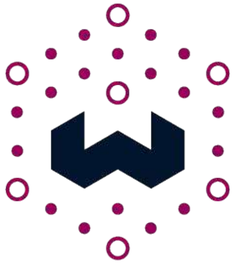
all code and models are open

Scalable and fast

Vectorizing and querying big data

Example use cases

Weaviate
Enterprise Search



Weaviate
ERP classifications

Weaviate
Cybersecurity

Weaviate
Biology

Weaviate
Data Harmonization

Weaviate
Social Media

Weaviate
Service Management

SeMI Technologies

Team

Etienne

Donna

Laura

Micha

Bob

Stefan

Marcin

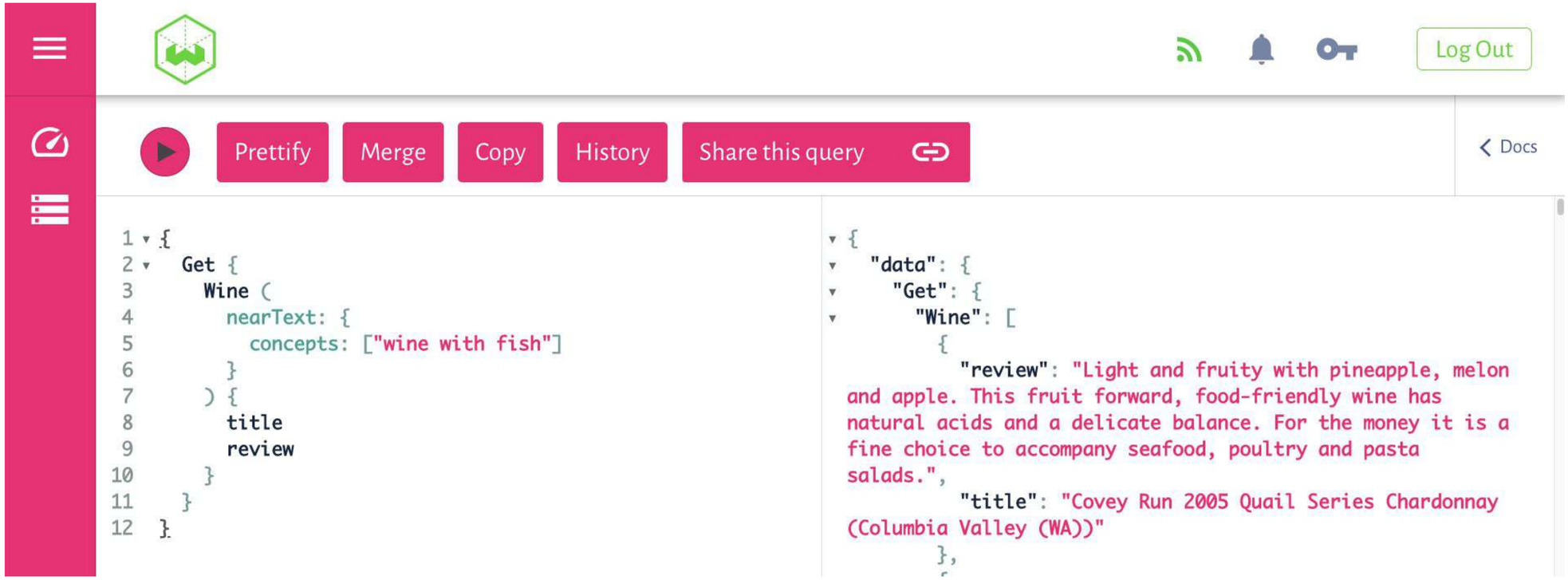
Henrique

Amar

Alicja

Try out Weaviate yourself!

- Go to www.semi.technology/developers



Thank you!

www.semi.technology

Laura Ham

laura@semi.technology

SeMI

Milvus

Getting Started with Milvus

Getting Started with Milvus - 1.1 What is Milvus?

Getting Started with Milvus. - 1.3 Milvus API

Milvus

Build Up the Unstructured Data Service

Jun Gu

09.2020

ZILLIZ

Speaker bio

Jun Gu

Database engineer, SME

LF AI — Voting member in Technical Advisory Council (TAC)

ZILLIZ — Partner, Chief Evangelist

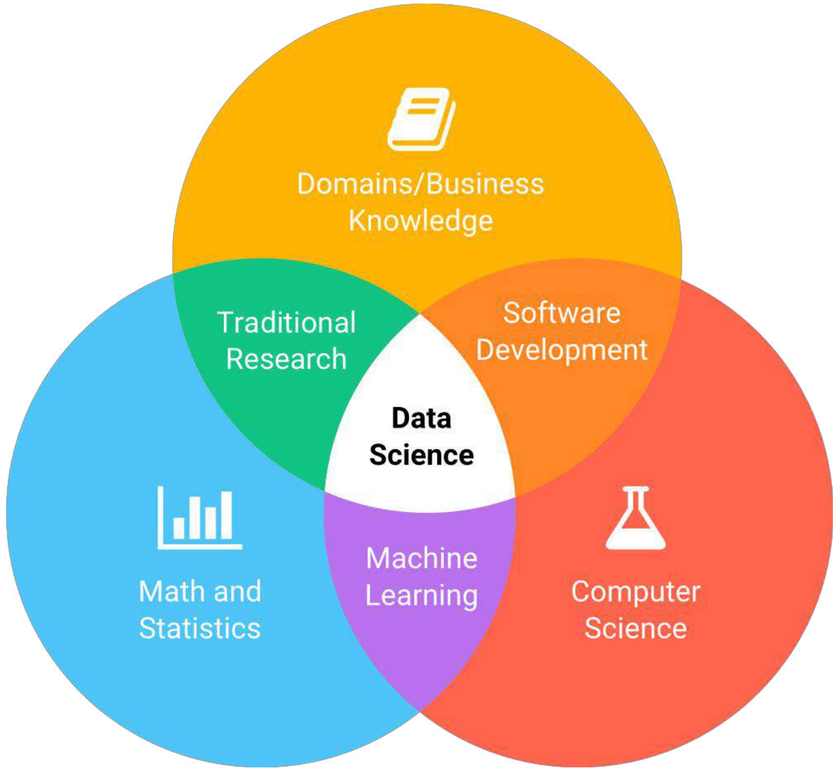
Career history

HUAWEI, Morgan Stanley, IBM, ICBC

Education

Peking University 1898

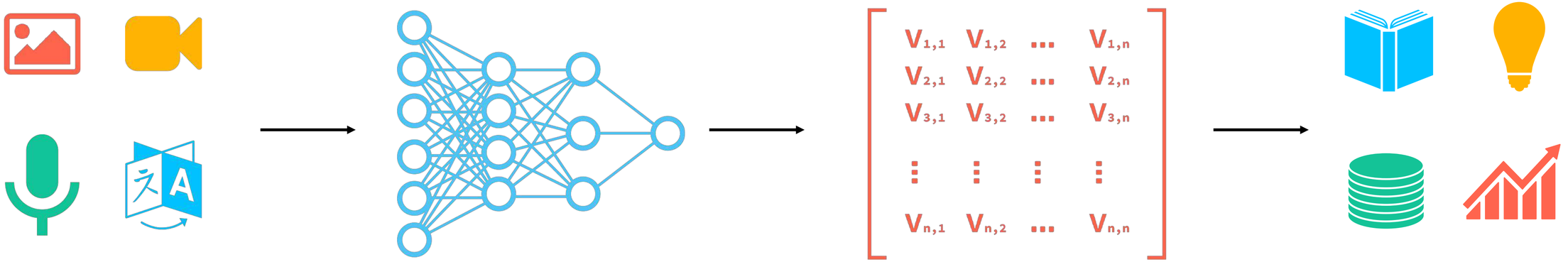
Zilliz: Who we are



- Open source software company based in Shanghai
- Mission: Reinvent data science
- Main contributor of Milvus project

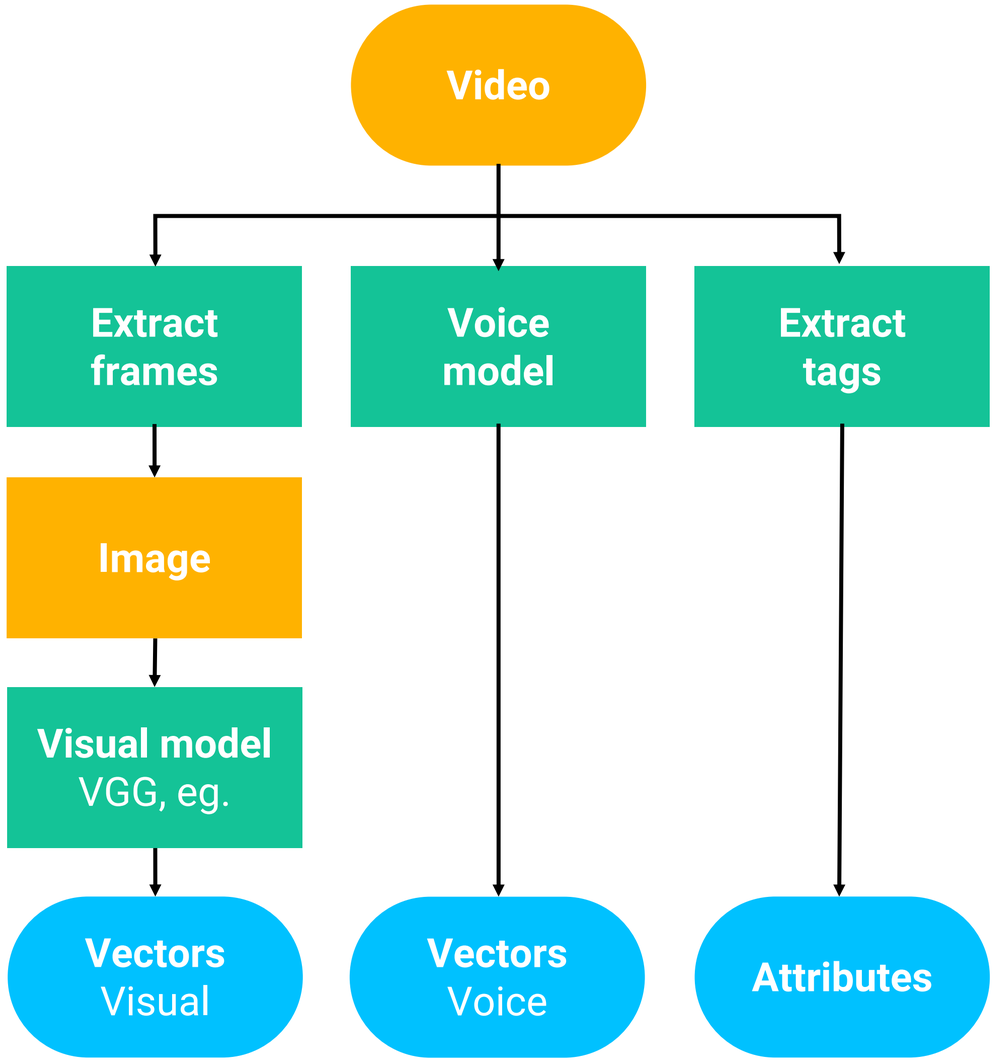
Unlock the treasure of unstructured data

AI algorithms transform image, video, voice, natural language into vectors, and enables understanding and utilization of unstructured data at scale.



Unstructured data **Deep learning models** **Vectors** **Knowledge, insight, $**

The flow-based AI applications



The sample pipelines for video processing

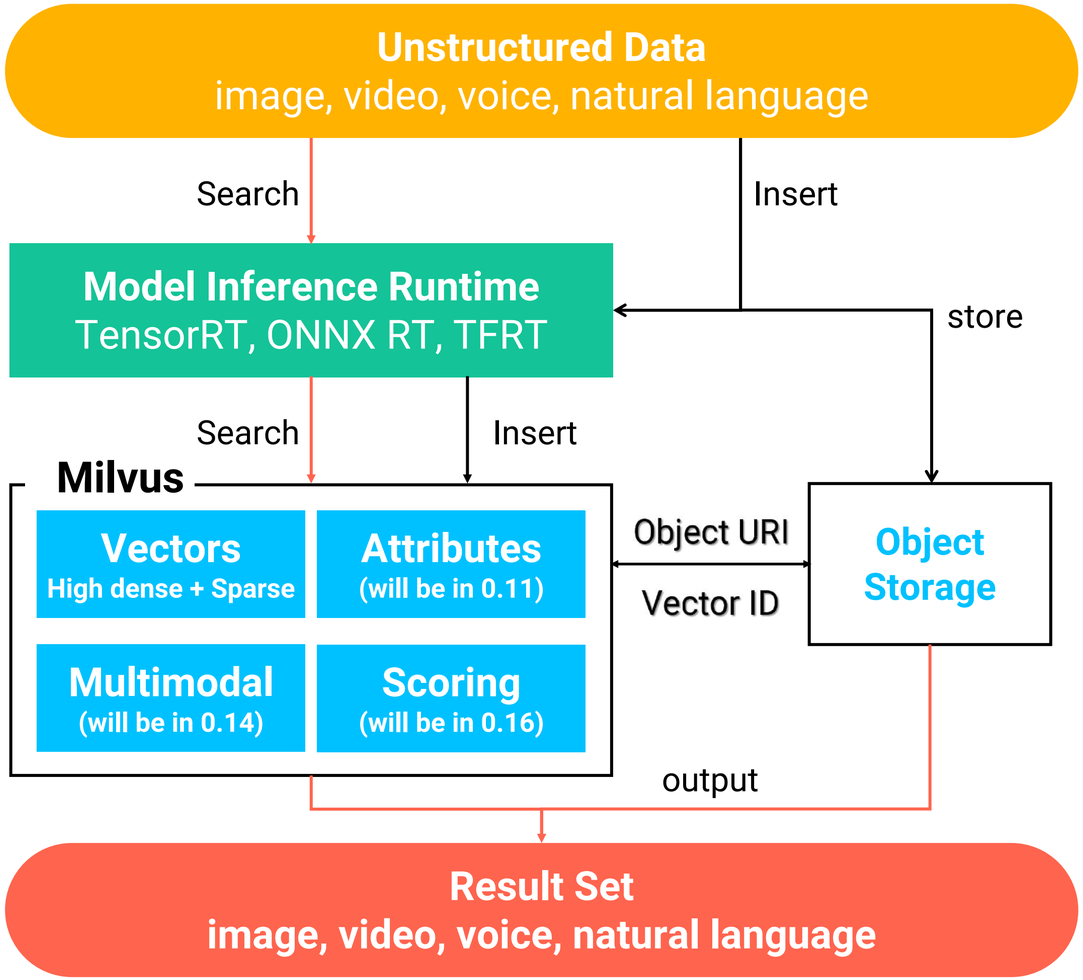
The most popular way

- Flexible
- Easy to compose, web-based UI
- Sample piplelines

The challenge

- Data fragmentation

The unstructured data service (UDS) for AI



Inference Layer

Data Service Layer

Milvus LF AI INCUBATION PROJECT

Why Milvus: Vectors are different

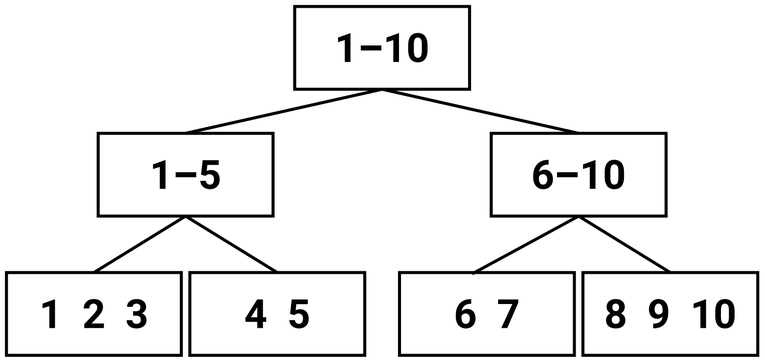
Numbers

Arithmetic operation

$+ - \times \div$

Number comparison

$a <=> b$



Operation

Organization

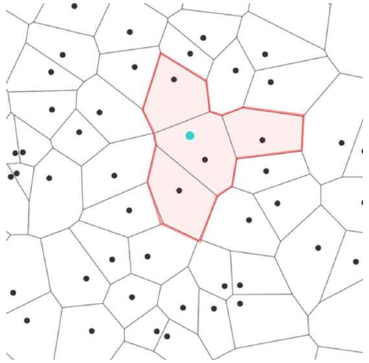
Vectors

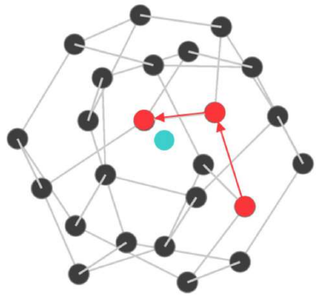
Similarity (eg. Euclidean distance)

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

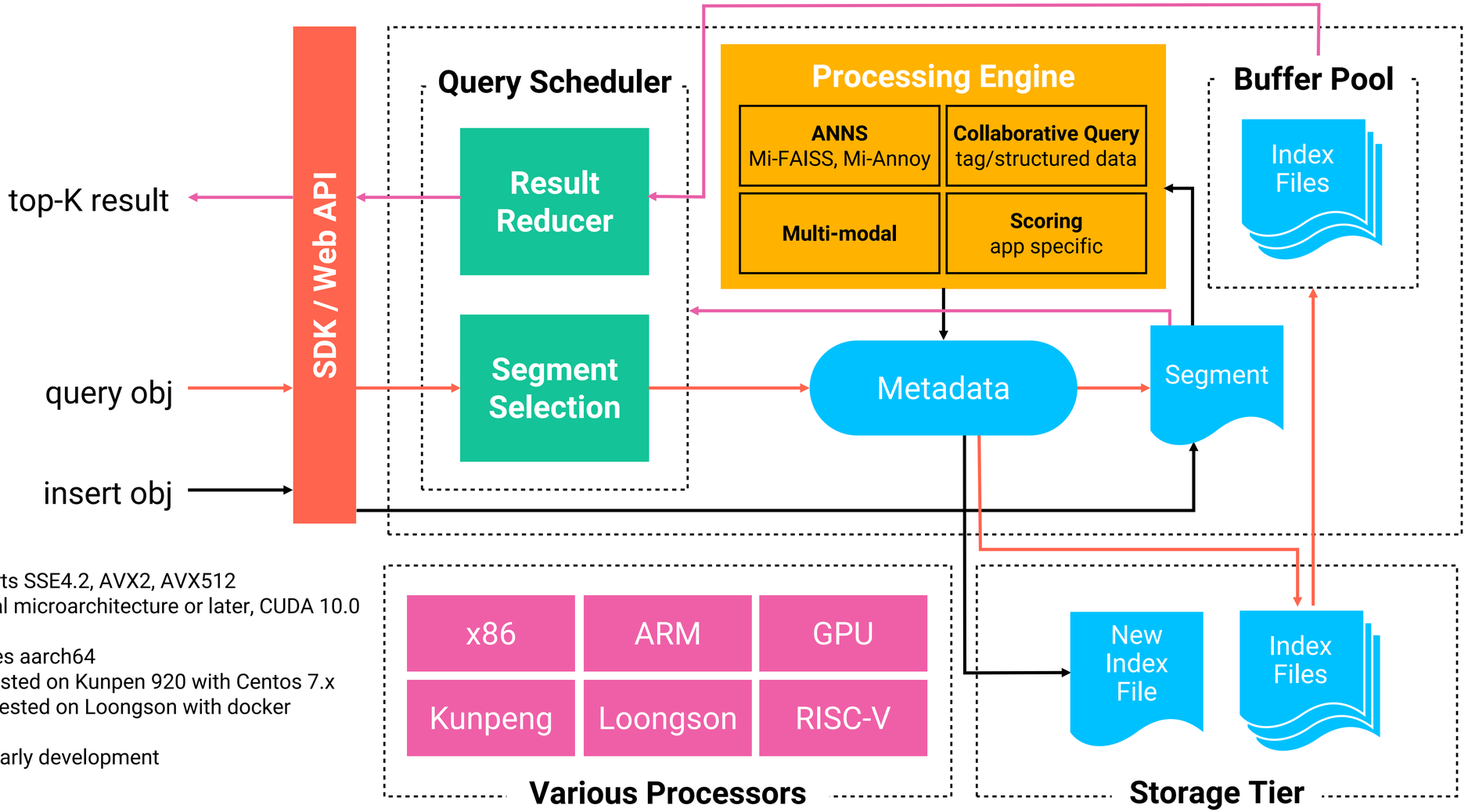
Similarity comparison

$$\mathrm{TopK}(A) = \arg\min_{B \in \gamma}(d(A,B))$$



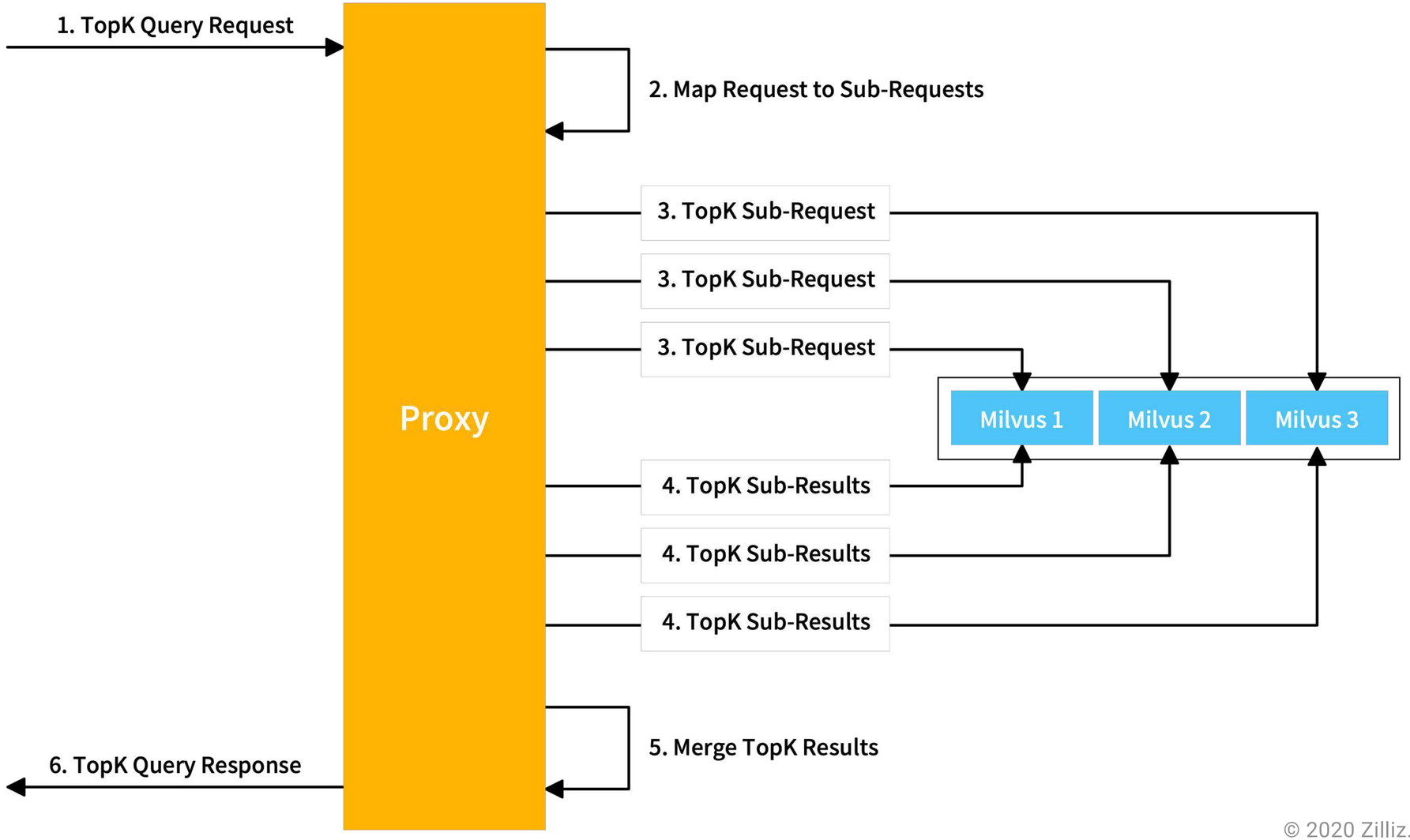


Milvus: The big picture



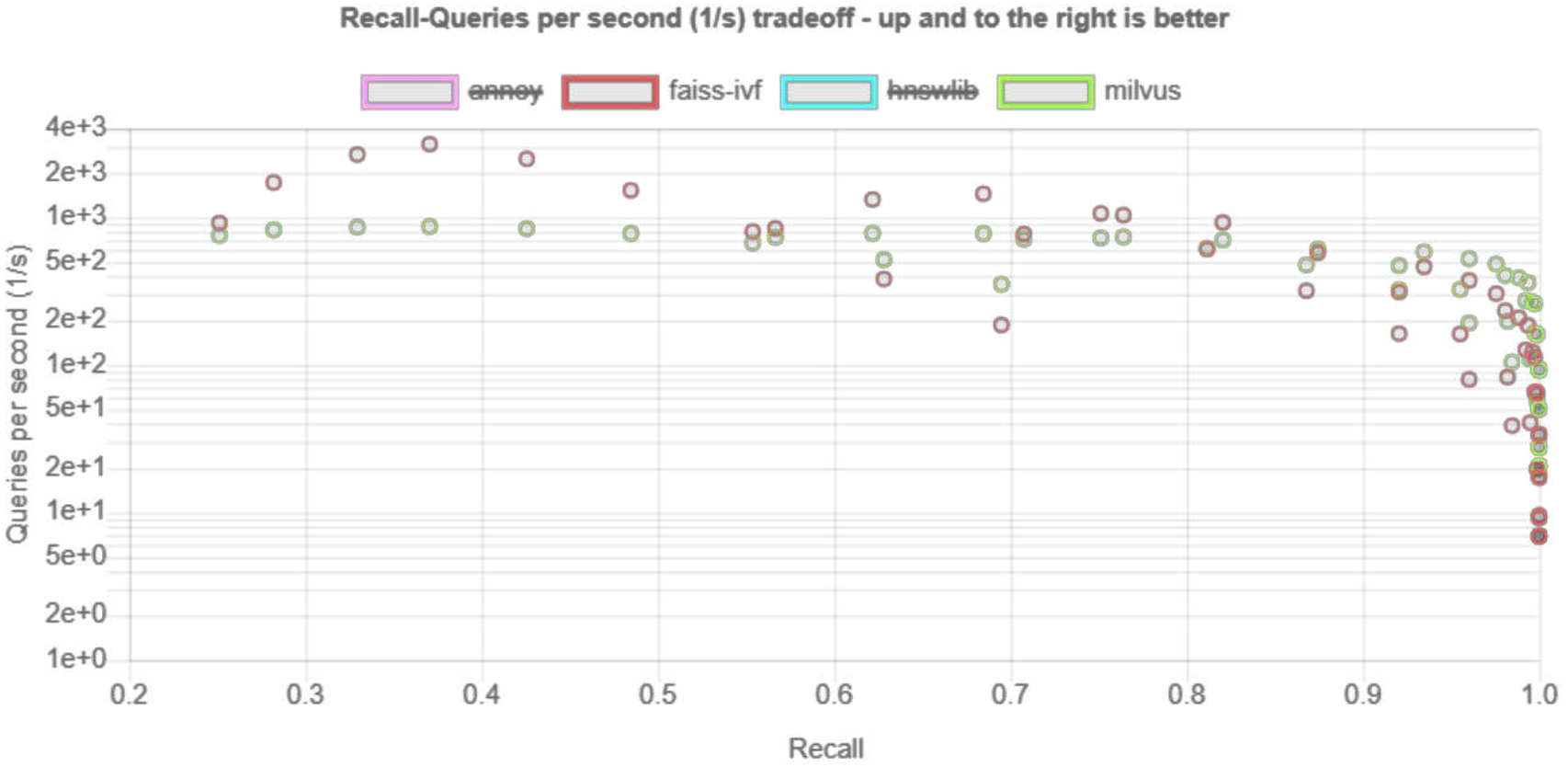
X86: supports SSE4.2, AVX2, AVX512
GPU: Pascal microarchitecture or later, CUDA 10.0 or later
Arm: requires aarch64
Kunpeng: tested on Kunpen 920 with Centos 7.x
Loongson: tested on Loongson with docker container
RSIC-V: in early development

Milvus: Distributed deployment



Milvus LF AI INCUBATION PROJECT

Milvus: The ANN benchmark

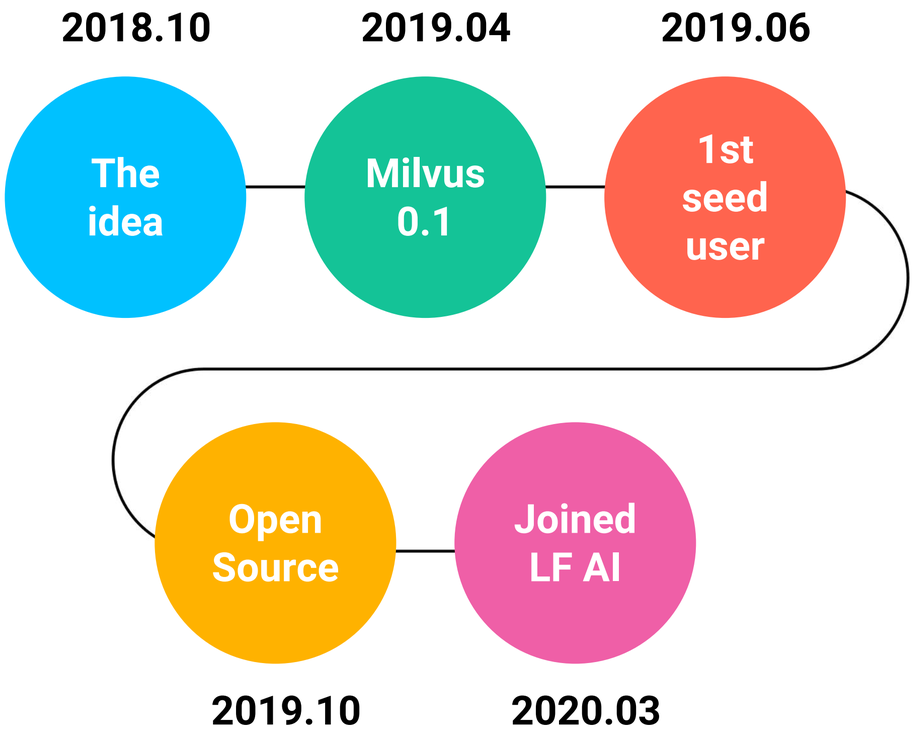


Milvus: 0.8.0
OS: Ubuntu 18.04
ECS: AWS c5.4xlarge (16c, 32GB), Intel XeonPlatinum 8275CL
Data set: sift-128-euclidean (1 million vectors)
More info: https://milvus.io/docs/benchmarks_aws

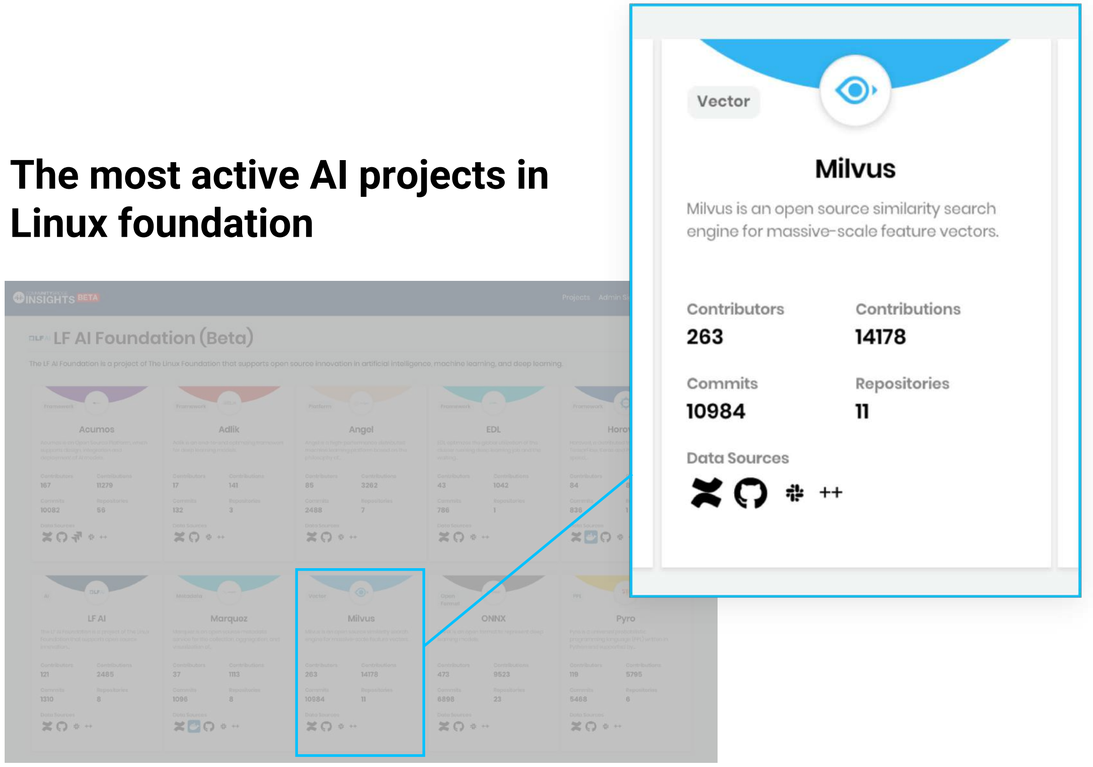
Milvus LF AI INCUBATION PROJECT

Special thanks to **ANN-Benchmarks** (developed by **Martin Aumueller**, **Erik Bernhardsson** and **Alec Faitfull**)

Milvus: The journey

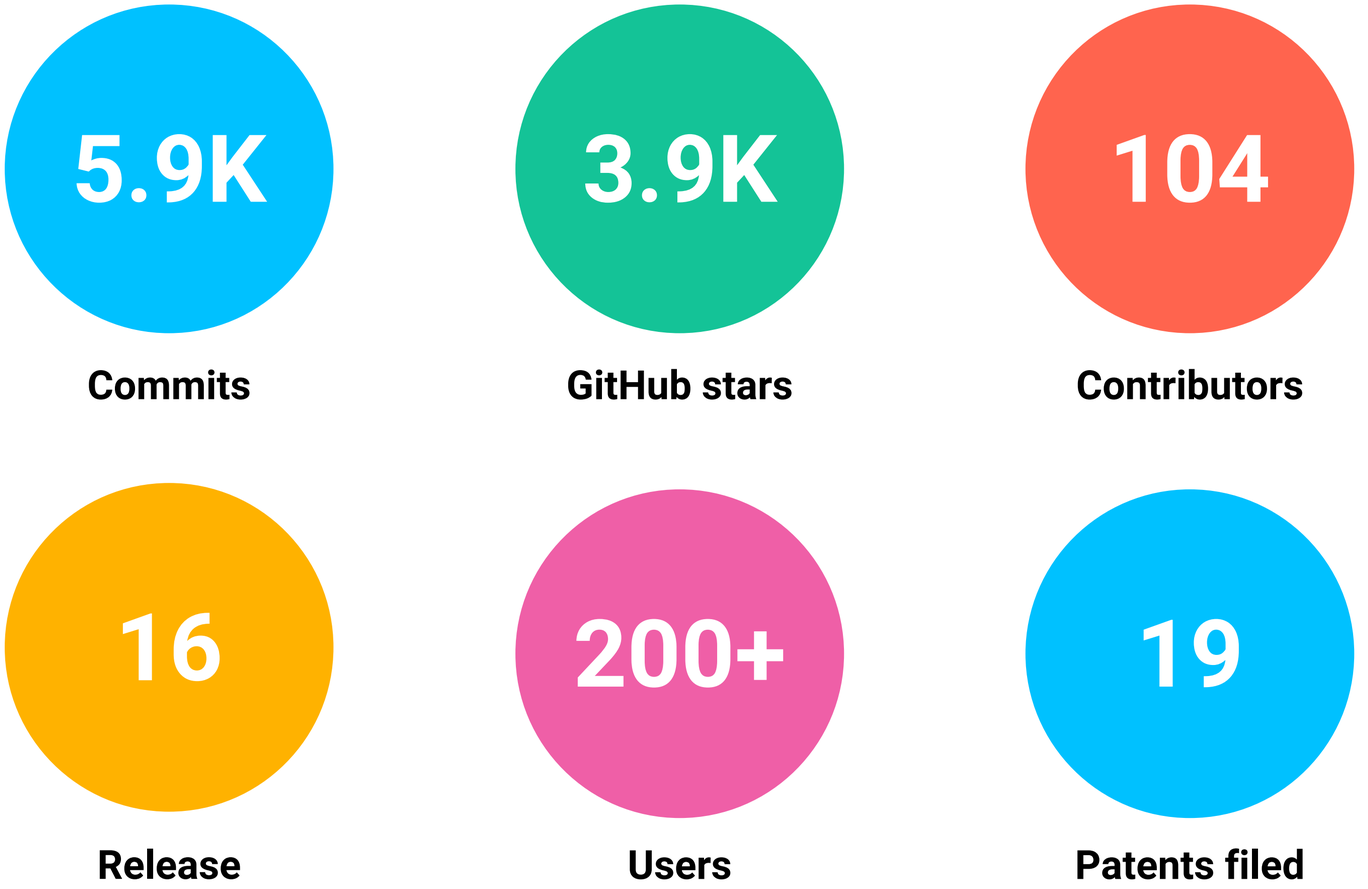


The most active AI projects in Linux foundation



Progress

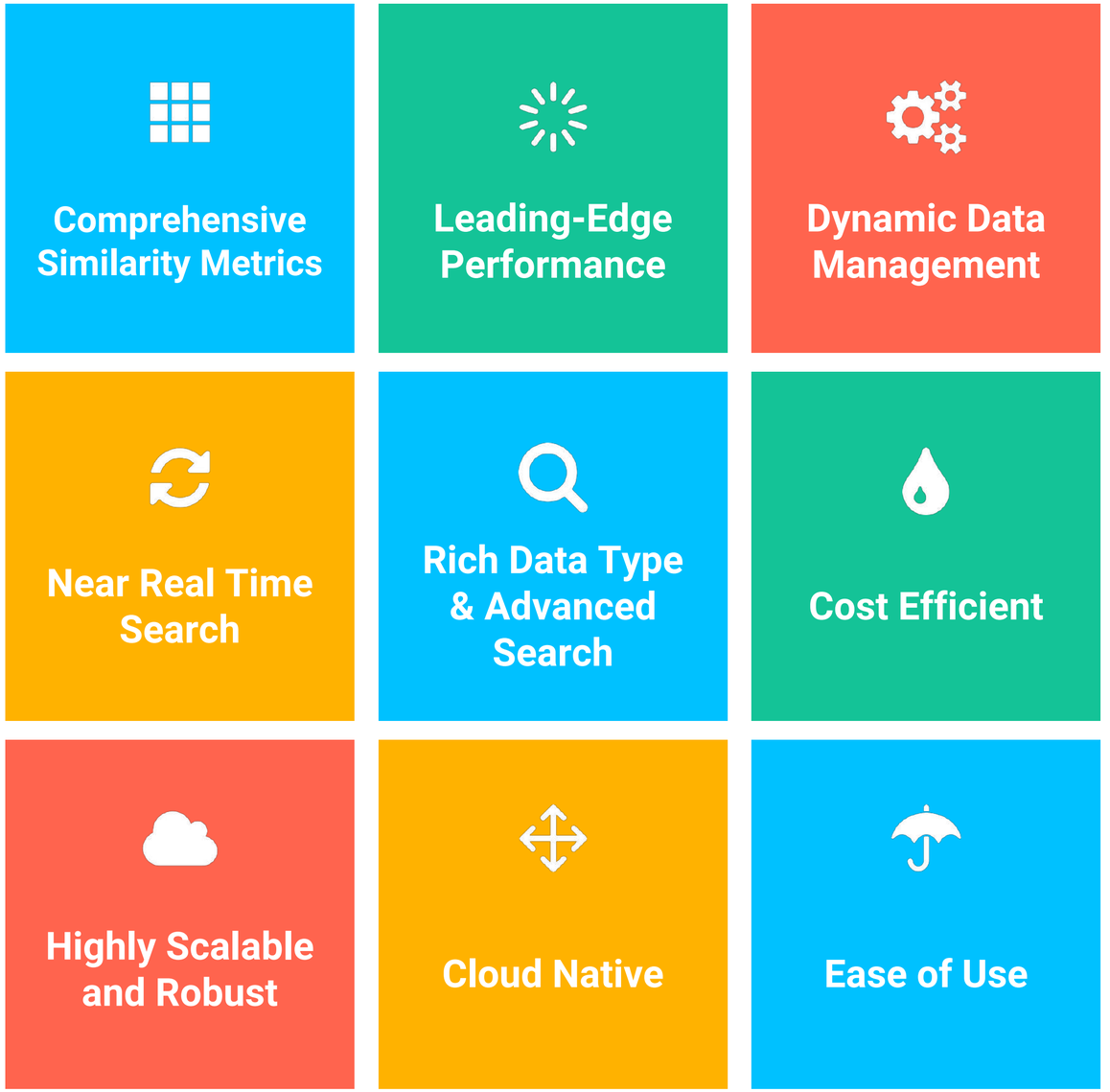
Unstoppable momentum since its debut.



Milvus LF AI INCUBATION PROJECT

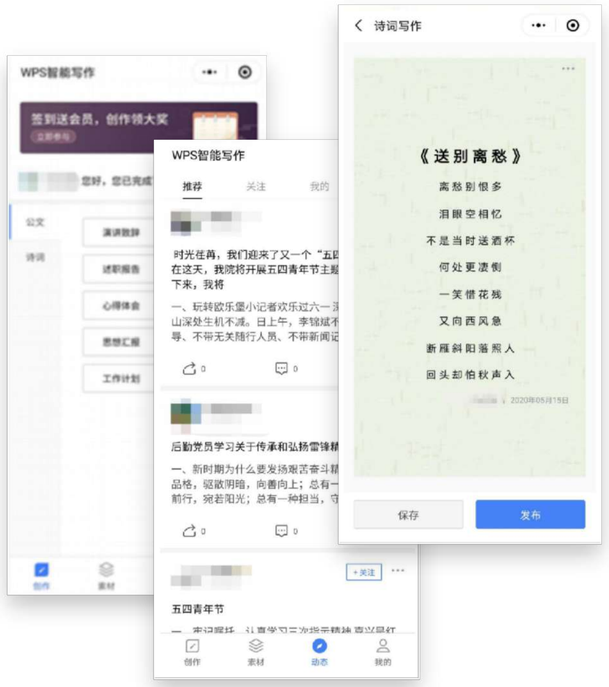
Milvus
Features & benefits

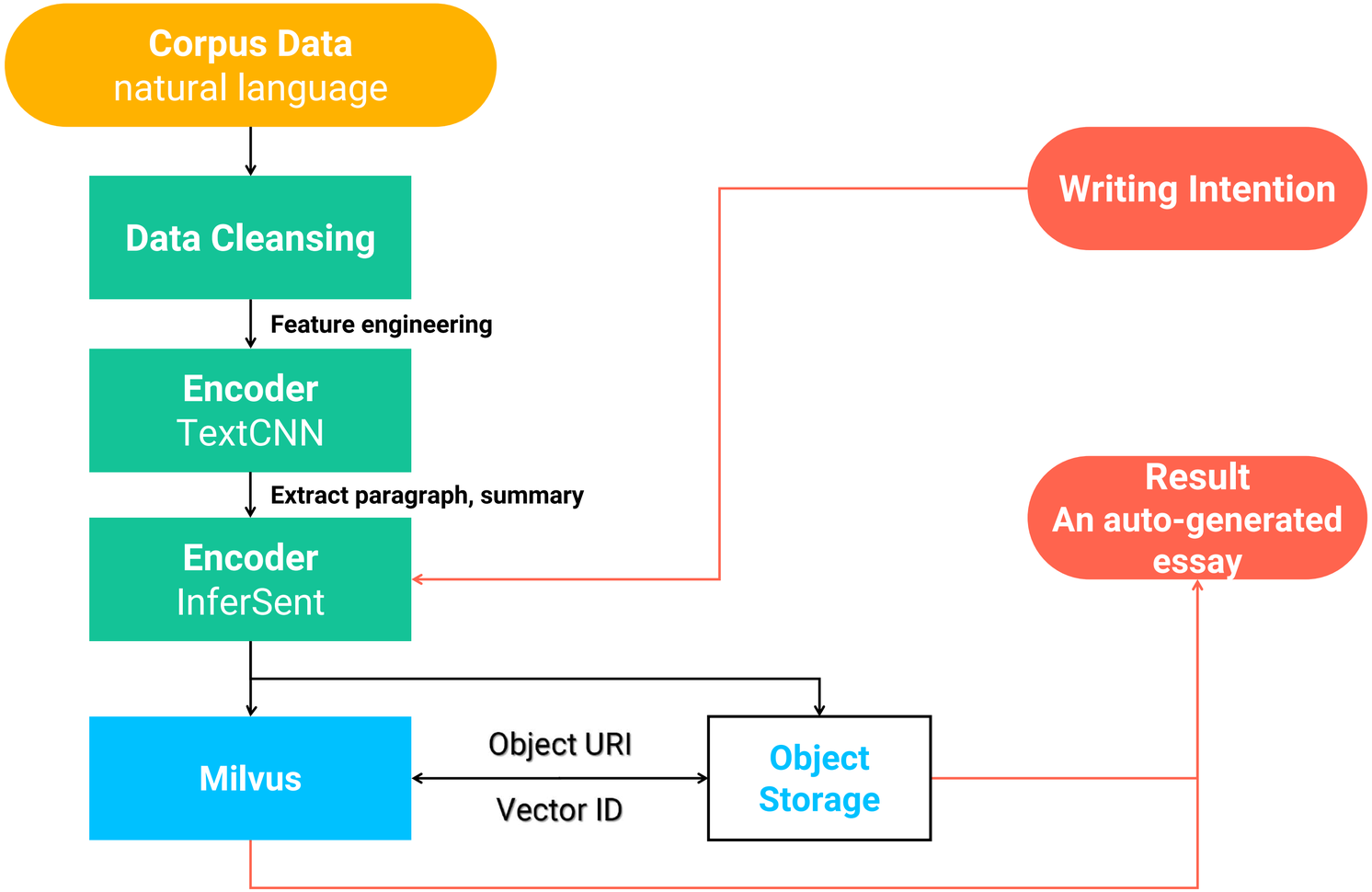
The world’s most advanced, our target ☺



Milvus LF AI INCUBATION PROJECT

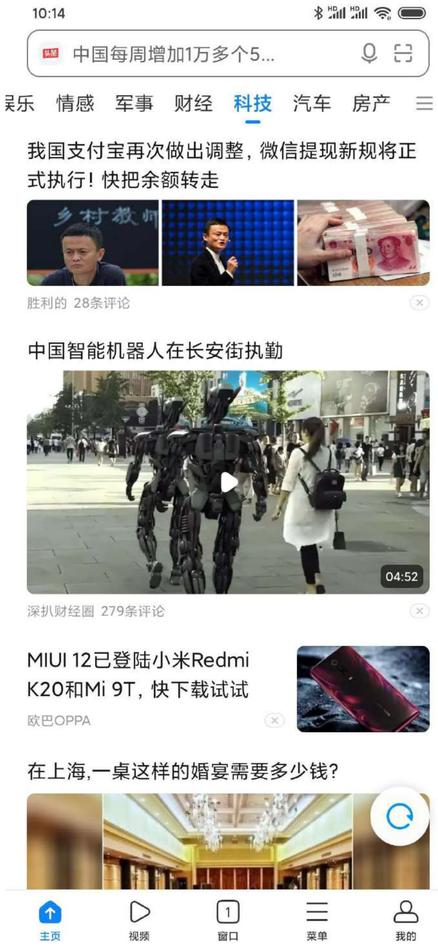
Use case: Inteligent writing assistant

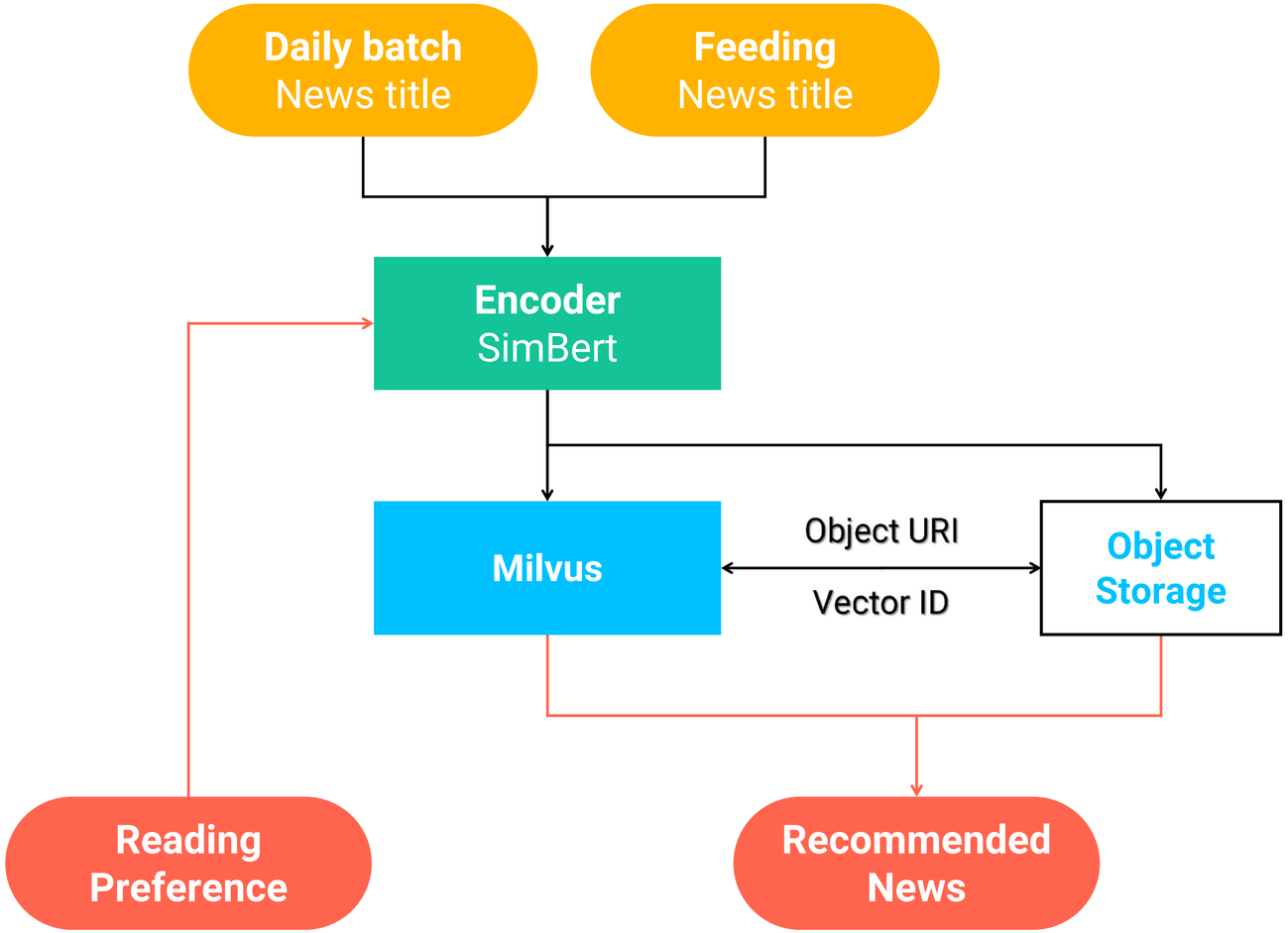




Milvus LF AI INCUBATION PROJECT

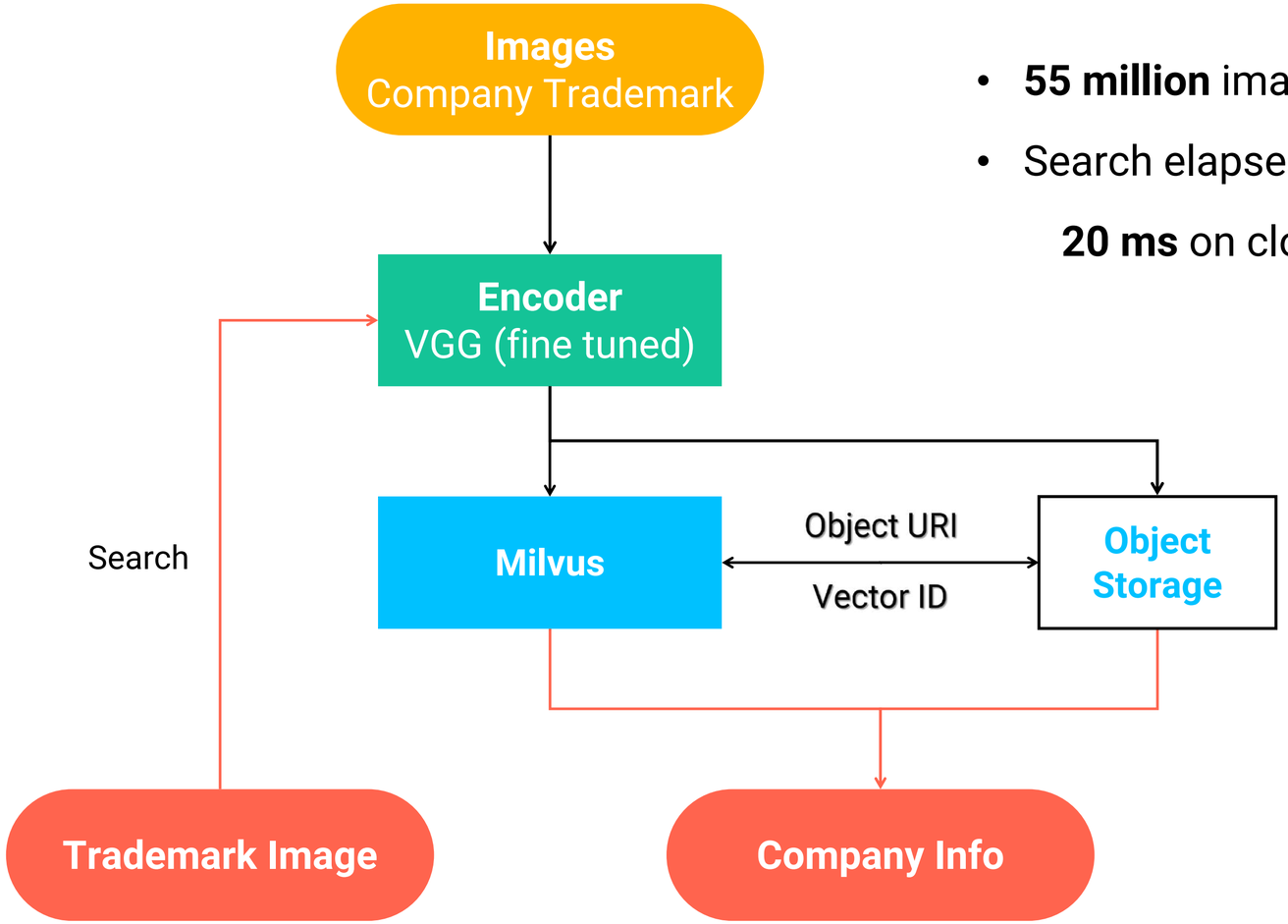
Use case: News recommendation on mobile





Milvus LF AI INCUBATION PROJECT

Use case: Image search for company trademark



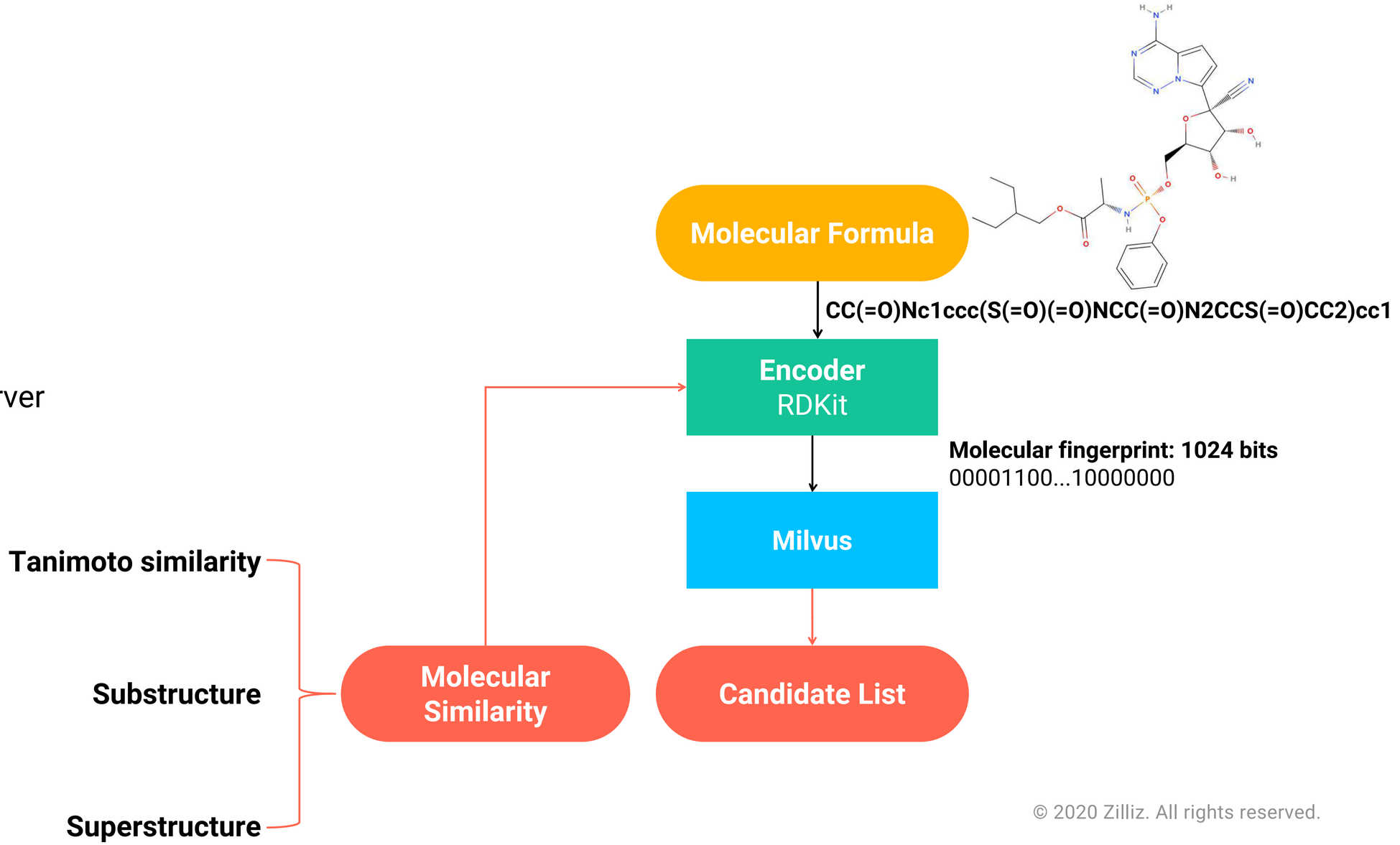
- **55 million** images
- Search elapsed time:

  20 ms on cloud GPU server

Use case: Pharmaceutical molecule analysis

- **800 million** molecules
- Search elapsed time:

  500 ms on single server



Useful Links

https://milvus.io

https://github.com/milvus-io/milvus

https://milvusio.slack.com

https://twitter.com/milvusio

https://medium.com/unstructured-data-service

https://zhuanlan.zhihu.com/ai-search

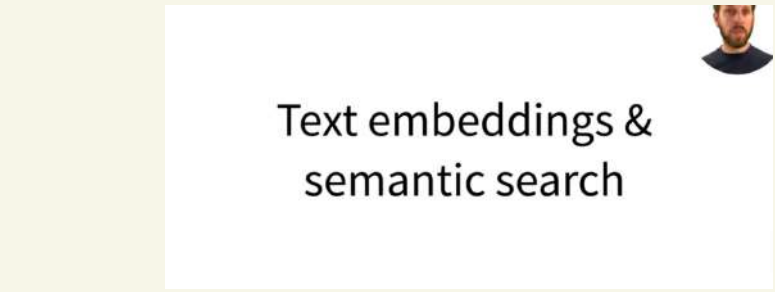
Performnance benchmark:
https://milvus.io/docs/benchmarks_aws

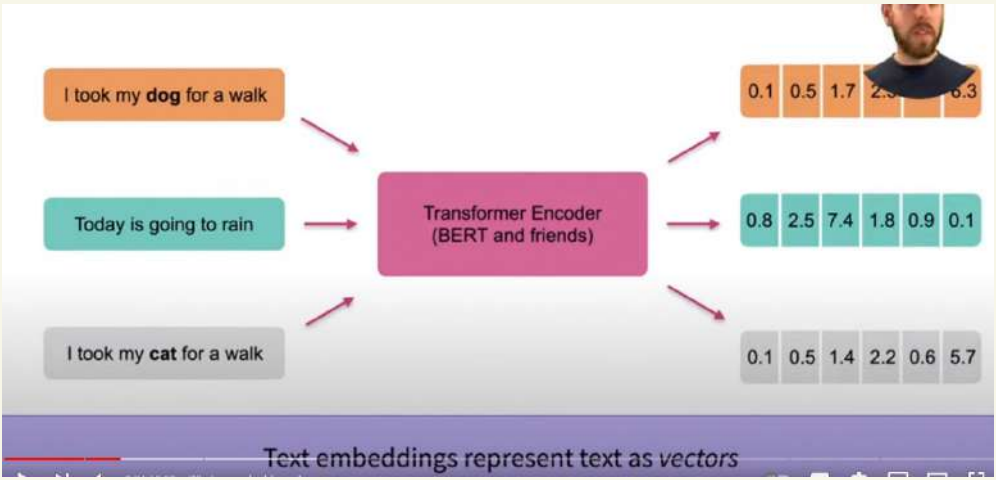
Live demo:
https://milvus.io/scenarios

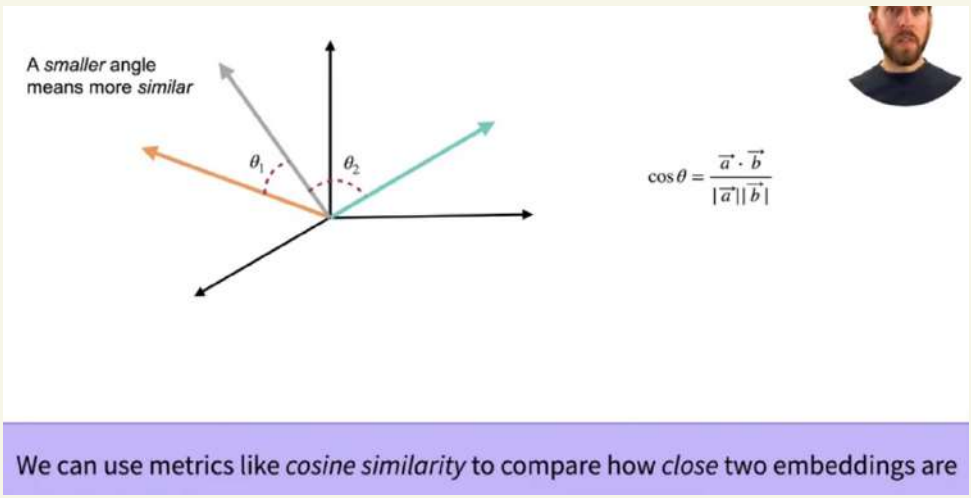
- Content-based image retrieval system (以图搜图)
- Q&A chatbot powered by NLP (智能客服机器人)
- Molecular analysis (化合物分析)

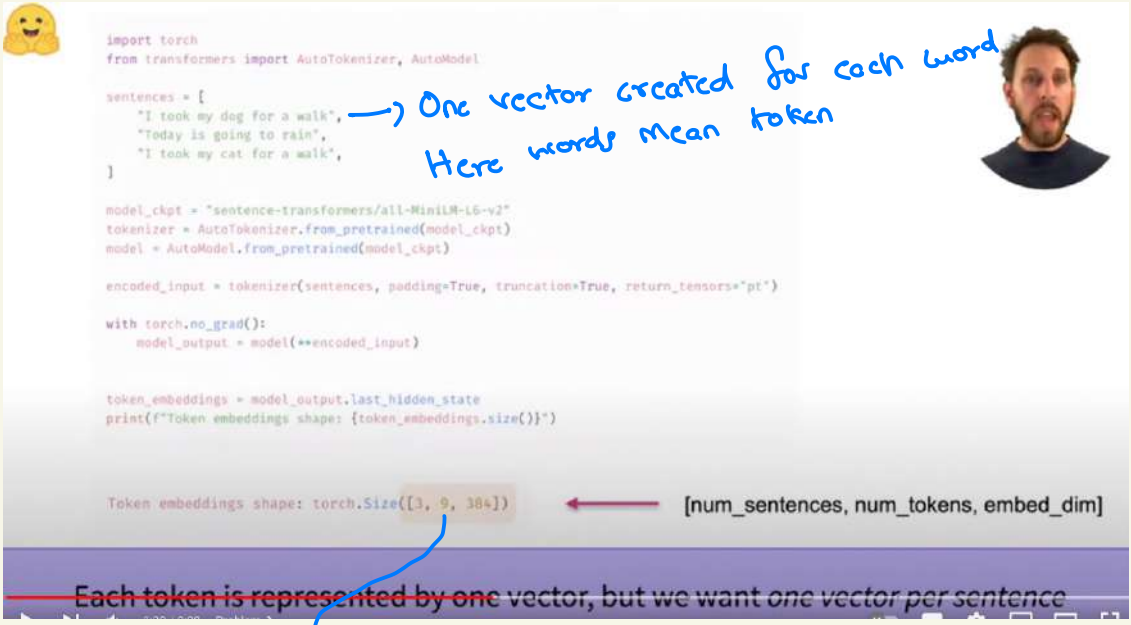
Thanks!

FAISS semantic search


Text embeddings &
semantic search

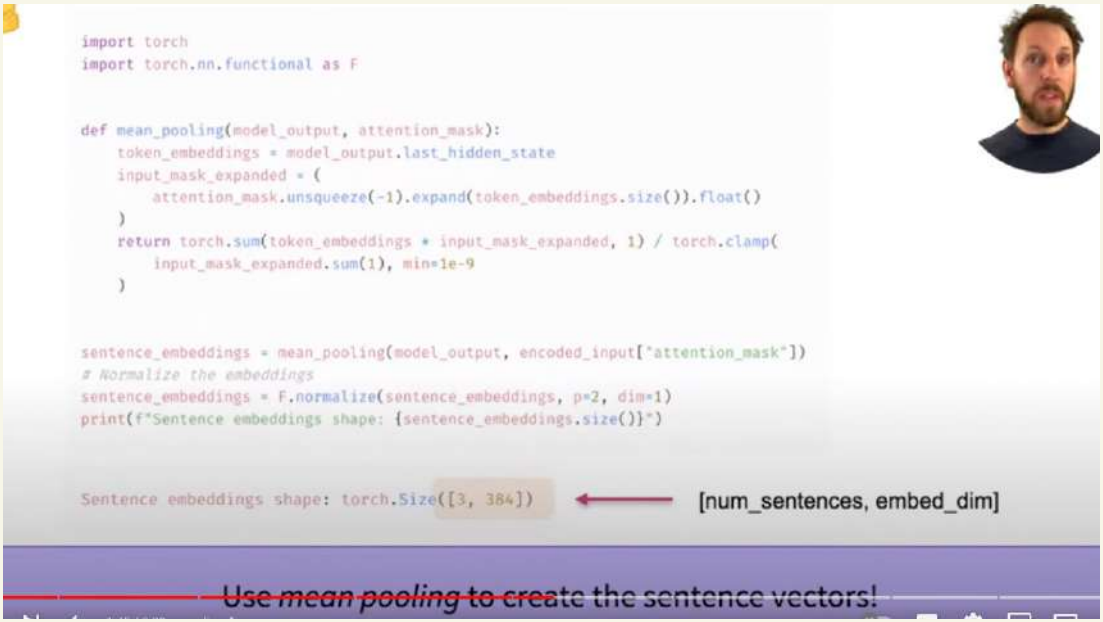

I took my dog for a walk
Today is going to rain
I took my cat for a walk
Transformer Encoder
(BERT and friends)
0.1 0.5 1.7
0.8 2.5 7.4 1.8 0.9 0.1
0.1 0.5 1.4 2.2 0.6 5.7
Text embeddings represent text as vectors


A smaller angle
means more similar
θ_1
θ_2
$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$
We can use metrics like cosine similarity to compare how close two embeddings are



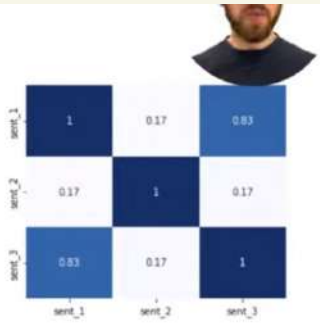
One vector created for each word
Here words mean token

Here 9 vectors created for each sentence



During pooling you don't add padding tokens in the average

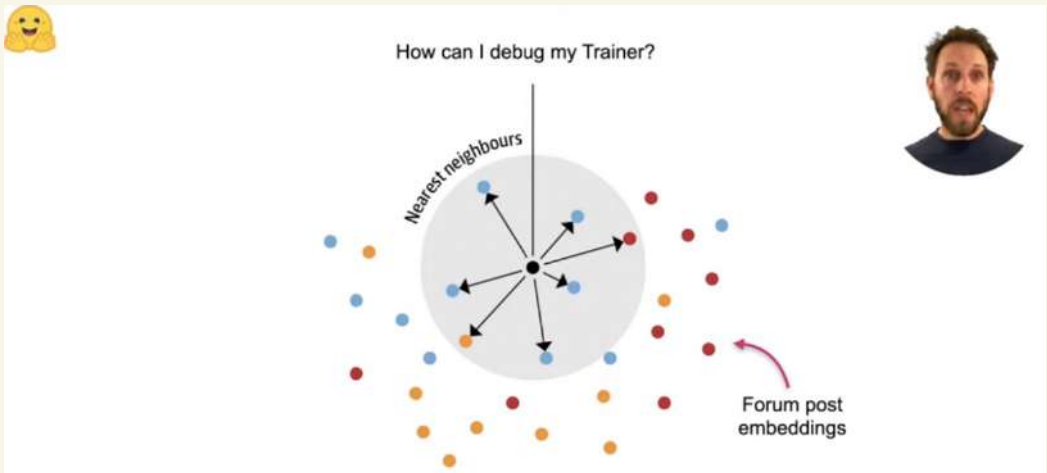
```python
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

sentence_embeddings = sentence_embeddings.detach().numpy()

scores = np.zeros((sentence_embeddings.shape[0], sentence_embeddings.shape[0]))

for idx in range(sentence_embeddings.shape[0]):
    scores[idx, :] = cosine_similarity([sentence_embeddings[idx]], sentence_embeddings)[0]
```



I took my **dog** for a walk

I took my **cat** for a walk

And once we have our embeddings we can calculate cosine similarity



You can use the same trick to measure similarity of *query* against a corpus of docs

```python
from datasets import load_dataset

squad = load_dataset("squad", split="validation[:100]")


def get_embeddings(text_list):
    encoded_input = tokenizer(
        text_list, padding=True, truncation=True, return_tensors="pt"
    )
    encoded_input = {k: v.to("cuda") for k, v in encoded_input.items()}
    with torch.no_grad():
        model_output = model(**encoded_input)
    return mean_pooling(model_output, encoded_input["attention_mask"])


squad_with_embeddings = squad.map(
    lambda x: {"embeddings": get_embeddings(x["context"]).cpu().numpy()[0]}
)
```

You can use the same trick to measure similarity of *query* against a corpus of docs

```python
squad_with_embeddings.add_faiss_index(column="embeddings")

question = "Who headlined the halftime show for Super Bowl 50?"
question_embedding = get_embeddings([question]).cpu().detach().numpy()
question_embedding.shape

scores, samples = squad_with_embeddings.get_nearest_examples(
    "embeddings", question_embedding, k=3
)
```

	context	scores
2	After a punt from both teams, Carolina got on ...	34.298462
1	The league announced on October 16, 2012, that...	32.570614
0	CBS broadcast Super Bowl 50 in the U.S., and c...	23.663607

We use a special *FAISS index* for fast nearest neighbour lookup

Member-only story

Master Semantic Search at Scale: Index Millions of Documents with Lightning-Fast Inference Times using FAISS and Sentence Transformers

Dive into an end-to-end demo of a high-performance semantic search engine leveraging GPU acceleration, efficient indexing techniques, and robust sentence encoders on datasets up to 1M documents, achieving 50 ms inference times

Luís Roque · Follow
Published in Towards Data Science
15 min read · Apr 1

Introduction

In search and information retrieval, semantic search has emerged as a game-changer. It allows us to search and retrieve documents based on their meaning or concepts rather than just keyword matching. The semantic search leads to more sophisticated and relevant results than traditional keyword-based search methods. However, the challenge lies in scaling semantic search to handle large corpora of documents without being overwhelmed by the computational complexity of analyzing every semantic content of a document.

In this article, we rise to the challenge of achieving scalable semantic search by harnessing the power of two cutting-edge techniques: FAISS for efficient indexing of semantic vectors and Sentence Transformers for encoding sentences into these vectors. FAISS is an outstanding library designed for the fast retrieval of nearest neighbors in high-dimensional spaces, enabling quick semantic nearest neighbor search even at a large scale. Sentence Transformers, a deep learning model, generates dense vector representations of sentences, effectively capturing their semantic meanings.

This article shows how we can use the synergy of FAISS and Sentence Transformers to build a scalable semantic search engine with remarkable performance. By integrating FAISS and Sentence Transformers, we can index semantic vectors from an extensive corpus of documents, resulting in a rapid and accurate semantic search experience at scale. Our approach can enable new applications such as contextualized question-answering and advanced recommendation systems with inference times as low as 50 ms when searching a corpus of 1M documents. We will guide you through implementing this state-of-the-art end-to-end solution and demonstrate its performance on benchmark datasets.

Figure 1: Search is all you need to navigate this world (source)

This article belongs to “Large Language Models Chronicles: Navigating the NLP Frontier”, a new weekly series of articles that will explore how to leverage the power of large models for various NLP tasks. By diving into these cutting-edge technologies, we aim to empower developers, researchers, and enthusiasts to harness the potential of NLP and unlock new possibilities.

Articles published so far:

1. Summarizing the latest Spotify releases with ChatGPT

As always, the code is available on my Github.

Sentence Transformers for Semantic Encoding

Deep learning brings forth the power of sentence transformers, which craft dense vector representations that capture the essence of a sentence's meaning. Trained on massive amounts of data, these models produce contextualized word embeddings, aiming to reconstruct input sentences accurately and draw semantically similar sentence pairs closer together.

To harness the potential of sentence transformers in semantic encoding, you'll first need to choose a suitable model architecture, such as BERT, RoBERTa, or XLNet. With a model in place, we will feed a corpus of documents into it, generating fixed-length semantic vectors for each sentence. These vectors are compact numerical representations of the core themes and topics within the sentences.

Let's take two sentences as examples: 'The dog chased the cat' and 'The cat chased the dog.' When processed through a sentence transformer, their resulting semantic vectors will be closely related, even with word order differences, because the underlying meaning is similar. On the other hand, a sentence like 'The sky is blue' will yield a more distant vector due to its contrasting meaning.

Using sentence transformers to encode an entire corpus, we obtain a collection of semantic vectors that encapsulate the overarching meanings of the documents. To make this transformed representation ready for efficient retrieval, we index it using FAISS. Stay tuned, as we'll dive into this topic in the next section.

FAISS for Efficient Indexing

FAISS supports various index structures optimized for different use cases. It is a library designed for scenarios where one must quickly find the closest matches to a given query vector in a large collection of vectors.

- Inverted files (IVF): Indexes clusters of similar vectors. Suitable for medium-dimensional vectors.
- Product quantization (PQ): Encodes vectors into quantized subspaces. Suitable for high-dimensional vectors.
- Cluster-based strategies: Organizes vectors into a hierarchical set of clusters for multi-level search. Suitable for very large datasets.

To use FAISS for semantic search, we first load our vector dataset (semantic vectors from sentence transformer encoding) and construct a FAISS index. The specific index structure we choose depends on factors like the dimensionality of our semantic vectors and desired efficiency. We then index the semantic vectors by passing them into the FAISS index, which will efficiently organize them to enable fast retrieval.

For search, we encode a new sentence into a semantic vector query and pass it to the FAISS index. FAISS will retrieve the closest matching semantic vectors and return the most similar sentences. Compared to linear search, which scores the query vector against every indexed vector, FAISS enables much faster retrieval times that typically scale logarithmically with the number of indexed vectors. Additionally, the indexes are highly memory-efficient because they compress the original dense vectors.

Inverted Files Index

The Inverted Files (IVF) index in FAISS clusters similar vectors into 'inverted files' and is suitable for medium-dimensional vectors (e.g., 100–1000 dimensions). Each inverted file contains a subset of vectors that are close together. At search time, FAISS searches only the inverted files closest to the query vector instead of searching through all vectors, enabling efficient search even with many vectors.

To construct an IVF index, we specify the number of inverted files (clusters) we want and the maximum number of vectors per inverted file. Then, FAISS assigns each vector to the closest inverted file until no inverted file exceeds the maximum. The inverted files contain representative points that summarize the vectors within them. At query time, FAISS computes the distance between the query vector and each inverted file representative point and searches only the closest inverted files for the closest matching vectors.

For example, if we have 1024-dimensional image feature vectors and want to perform a fast search over 1 million vectors, we could create an IVF index with 1024 inverted files (clusters) and a maximum of 1000 vectors per inverted file. In this approach, FAISS would search only the closest inverted files to the query, resulting in faster search times than linear search.

Putting It All Together

In this section, we will build a scalable semantic search with FAISS and Sentence Transformers. We will show you how to evaluate the performance benchmarks of

this approach and discuss further improvements and applications.

Scalable Semantic Search Engine

To build a scalable semantic search engine, we first initialize the `ScalableSemanticSearch` class. This class takes care of encoding sentences using Sentence Transformers and indexing them using FAISS for efficient searching. It also provides utility methods for saving and loading indices, measuring time, and memory usage.

```
semantic_search = ScalableSemanticSearch(device="cuda")
```

Next, we encode the large corpus of documents using the encoding method, which returns a numpy array of semantic vectors. The method also creates a mapping between indices and sentences that will be useful later when retrieving the top results.

```
embeddings = semantic_search.encode(corpus)
```

Now, we build the FAISS index using the build_index method, which takes the embeddings as input. This method creates an IndexIVFPQ or IndexFlatL2 index, depending on the number of data points in the embeddings.

```
semantic_search.build_index(embeddings)
```

Selecting Indexing Approaches Based on Dataset Size

We define two indexing approaches: Exact Search with L2 distance and Approximate Search with Product Quantization and L2 distance. We will also discuss the rationale behind selecting the first approach for smaller datasets (less than 1500 documents) and the second for larger datasets.

1. Exact Search with L2 distance

Exact Search with L2 distance is an exact search method that computes the L2 (Euclidean) distance between a query vector and every vector in the dataset. This method guarantees to find the exact nearest neighbors but can be slow for large datasets, as it performs a linear scan of the data.

Use case: This method is suitable for small datasets where the exact nearest neighbors are required, and the computational cost is not a concern.

2. Approximate Search with Product Quantization and L2 distance

Approximate Search with Product Quantization and L2 distance is an approximate nearest neighbor search method that combines an inverted file structure, product quantization, and L2 distance to search for similar vectors in large datasets efficiently. The method first clusters the dataset using k-means (faiss.IndexFlatL2 as the quantizer) and then applies product quantization to compress the residual vectors. This approach allows for a faster search using less memory than brute-force methods.

Use case: This method is suitable for large datasets where the exact nearest neighbors are not strictly required, and the primary focus is on search speed and memory efficiency.

The rationale for selecting different approaches based on dataset size

For datasets containing less than 1500 documents, we set the Exact Search with L2 distance approach because the computational cost is not a significant concern in this case. Furthermore, this approach guarantees to find the nearest neighbors, which is desirable for smaller datasets.

We prefer using the Approximate Search with Product Quantization and L2 distance approach for larger datasets because it offers a more efficient search and consumes less memory than the exact search method. The Approximate Search approach proves to be more suitable for large datasets when prioritizing search speed and memory efficiency over finding the exact nearest neighbors.

Search Procedure

After building the index, we can perform a semantic search by providing an input query and the number of top results to return. The `search` method computes cosine similarity between the input sentence and the indexed embeddings and returns the indices and scores of the top matching sentences.

```
query = "What is the meaning of life?"
top = 5
top_indices, top_scores = semantic_search.search(query, top)
```

Finally, we can retrieve the top sentences using the `get_top_sentences` method, which takes the index to sentence mapping and the top indices as input and returns a list of the top sentences.

```
top_sentences = ScalableSemanticSearch.get_top_sentences(semantic_search.hashma
```

The Complete Model

The complete class of our model looks like the following:

```
class ScalableSemanticSearch:
    """Vector similarity using product quantization with sentence transformers

    def __init__(self, device="cpu"):
        self.device = device
        self.model = SentenceTransformer(
            "sentence-transformers/all-mpnet-base-v2", device=self.device
        )
        self.dimension = self.model.get_sentence_embedding_dimension()
        self.quantizer = None
        self.index = None
        self.hashmap_index_sentence = None

        log_directory = "log"
        if not os.path.exists(log_directory):
            os.makedirs(log_directory)
        log_file_path = os.path.join(log_directory, "scalable_semantic_search.l

        logging.basicConfig(
            filename=log_file_path,
            level=logging.INFO,
            format="%(asctime)s %(levelname)s: %(message)s",
        )
        logging.info("ScalableSemanticSearch initialized with device: %s", self

    @staticmethod
    def calculate_clusters(n_data_points: int) -> int:
        return max(2, min(n_data_points, int(np.sqrt(n_data_points))))
```

```
    def encode(self, data: List[str]) -> np.ndarray:
        """Encode input data using sentence transformer model.

        Args:
            data: List of input sentences.

        Returns:
            Numpy array of encoded sentences.
        """
        embeddings = self.model.encode(data)
        self.hashmap_index_sentence = self.index_to_sentence_map(data)
        return embeddings.astype("float32")

    def build_index(self, embeddings: np.ndarray) -> None:
        """Build the index for FAISS search.

        Args:
            embeddings: Numpy array of encoded sentences.
        """
        n_data_points = len(embeddings)
        if (
            n_data_points >= 1500
        ):  # Adjust this value based on the minimum number of data points requ
            self.quantizer = faiss.IndexFlatL2(self.dimension)
            n_clusters = self.calculate_clusters(n_data_points)
            self.index = faiss.IndexIVFPQ(
                self.quantizer, self.dimension, n_clusters, 8, 4
            )
            logging.info("IndexIVFPQ created with %d clusters", n_clusters)
        else:
            self.index = faiss.IndexFlatL2(self.dimension)
            logging.info("IndexFlatL2 created")

        if isinstance(self.index, faiss.IndexIVFPQ):
            self.index.train(embeddings)
        self.index.add(embeddings)
        logging.info("Index built on device: %s", self.device)

    @staticmethod
    def index_to_sentence_map(data: List[str]) -> Dict[int, str]:
        """Create a mapping between index and sentence.

        Args:
            data: List of sentences.

        Returns:
            Dictionary mapping index to the corresponding sentence.
        """
        return {index: sentence for index, sentence in enumerate(data)}

    @staticmethod
    def get_top_sentences(
```

```
        index_map: Dict[int, str], top_indices: np.ndarray
    ) -> List[str]:
        """Get the top sentences based on the indices.

        Args:
            index_map: Dictionary mapping index to the corresponding sentence.
            top_indices: Numpy array of top indices.

        Returns:
            List of top sentences.
        """
        return [index_map[i] for i in top_indices]

    def search(self, input_sentence: str, top: int) -> Tuple[np.ndarray, np.nda
        """Compute cosine similarity between an input sentence and a collection

        Args:
            input_sentence: The input sentence to compute similarity against.
            top: The number of results to return.

        Returns:
            A tuple containing two numpy arrays. The first array contains the c
            sentence and the embeddings, ordered in descending order. The secon
            corresponding embeddings in the original array, also ordered by des
        """
        vectorized_input = self.model.encode(
            [input_sentence], device=self.device
        ).astype("float32")
        D, I = self.index.search(vectorized_input, top)
        return I[0], 1 - D[0]

    def save_index(self, file_path: str) -> None:
        """Save the FAISS index to disk.

        Args:
            file_path: The path where the index will be saved.
        """
        if hasattr(self, "index"):
            faiss.write_index(self.index, file_path)
        else:
            raise AttributeError(
                "The index has not been built yet. Build the index using `build
            )

    def load_index(self, file_path: str) -> None:
        """Load a previously saved FAISS index from disk.

        Args:
            file_path: The path where the index is stored.
        """
        if os.path.exists(file_path):
            self.index = faiss.read_index(file_path)
        else:
```

```
            raise FileNotFoundError(f"The specified file '{file_path}' does not

    @staticmethod
    def measure_time(func: Callable, *args, **kwargs) -> Tuple[float, Any]:
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        elapsed_time = end_time - start_time
        return elapsed_time, result

    @staticmethod
    def measure_memory_usage() -> float:
        process = psutil.Process(os.getpid())
        ram = process.memory_info().rss
        return ram / (1024**2)

    def timed_train(self, data: List[str]) -> Tuple[float, float]:
        start_time = time.time()
        embeddings = self.encode(data)
        self.build_index(embeddings)
        end_time = time.time()
        elapsed_time = end_time - start_time
        memory_usage = self.measure_memory_usage()
        logging.info(
            "Training time: %.2f seconds on device: %s", elapsed_time, self.dev
        )
        logging.info("Training memory usage: %.2f MB", memory_usage)
        return elapsed_time, memory_usage

    def timed_infer(self, query: str, top: int) -> Tuple[float, float]:
        start_time = time.time()
        _, _ = self.search(query, top)
        end_time = time.time()
        elapsed_time = end_time - start_time
        memory_usage = self.measure_memory_usage()
        logging.info(
            "Inference time: %.2f seconds on device: %s", elapsed_time, self.de
        )
        logging.info("Inference memory usage: %.2f MB", memory_usage)
        return elapsed_time, memory_usage

    def timed_load_index(self, file_path: str) -> float:
        start_time = time.time()
        self.load_index(file_path)
        end_time = time.time()
        elapsed_time = end_time - start_time
        logging.info(
            "Index loading time: %.2f seconds on device: %s", elapsed_time, sel
        )
        return elapsed_time
```

End-to-End Demo

This section will provide an end-to-end demo of the scalable semantic search engine using the SemanticSearchDemo class and the
main function from the code above. The goal is to understand how the different concepts and components combine to create a practical application.

Initializing the SemanticSearchDemo class: To initialize the SemanticSearchDemo class, provide the dataset path, the ScalableSemanticSearch model, an optional index path, and an optional subset size. This flexibility enables using different datasets, models, and subset sizes.

```
demo = SemanticSearchDemo(
    dataset_path, model, index_path=index_path, subset_size=subset_size
)
```

Loading data: The `load_data` function actively reads and processes data from a file, then returns a list of sentences. The system uses this data to train the semantic search model.

```
sentences = demo.load_data(file_name)
subset_sentences = sentences[:subset_size]
```

Training the model: The `train` function trains the semantic search model on the dataset and returns the training process's elapsed time and memory usage.

```
training_time, training_memory_usage = demo.train(subset_sentences)
```

Performing inference: The `infer` function takes a query, a list of sentences to search in, and the number of top results to return. It performs inference on the model and returns the top matching sentences, elapsed time, and memory usage for the inference process.

```
top_sentences, inference_time, inference_memory_usage = demo.infer(
    query, subset_sentences, top=3
)
```

The full class for the demo is below:

```
class SemanticSearchDemo:
    """A demo class for semantic search using the ScalableSemanticSearch model.

    def __init__(
        self,
        dataset_path: str,
        model: ScalableSemanticSearch,
        index_path: Optional[str] = None,
        subset_size: Optional[int] = None,
    ):
        self.dataset_path = dataset_path
        self.model = model
        self.index_path = index_path
        self.subset_size = subset_size

        if self.index_path is not None and os.path.exists(self.index_path):
            self.loading_time = self.model.timed_load_index(self.index_path)
        else:
            self.train()

    def load_data(self, file_name: str) -> List[str]:
        """Load data from a file.

        Args:
            file_name: The name of the file containing the data.

        Returns:
            A list of sentences loaded from the file.
        """
        with open(f"{self.dataset_path}/{file_name}", "r") as f:
            reader = csv.reader(f, delimiter="\t")
            next(reader)  # Skip the header
            sentences = [row[3] for row in reader]  # Extract the sentences
        return sentences

    def train(self, data: Optional[List[str]] = None) -> Tuple[float, float]:
        """Train the semantic search model and measure time and memory usage.

        Args:
            data: A list of sentences to train the model on. If not provided, t
```

```
            Returns:
                A tuple containing the elapsed time in seconds and the memory usage
            """
            if data is None:
                file_name = "GenericsKB-Best.tsv"
                data = self.load_data(file_name)

                if self.subset_size is not None:
                    data = data[: self.subset_size]

            elapsed_time, memory_usage = self.model.timed_train(data)

            if self.index_path is not None:
                self.model.save_index(self.index_path)

            return elapsed_time, memory_usage

        def infer(
            self, query: str, data: List[str], top: int
        ) -> Tuple[List[str], float, float]:
            """Perform inference on the semantic search model and measure time and

            Args:
                query: The input query to search for.
                data: A list of sentences to search in.
                top: The number of top results to return.

            Returns:
                A tuple containing the list of top sentences that match the input q
            """
            elapsed_time, memory_usage = self.model.timed_infer(query, top)
            top_indices, _ = self.model.search(query, top)
            index_map = self.model.index_to_sentence_map(data)
            top_sentences = self.model.get_top_sentences(index_map, top_indices)

            return top_sentences, elapsed_time, memory_usage
```

Performance Evaluation of our Scalable Semantic Search Engine

In order to evaluate the performance of our scalable semantic search engine, we can measure the time and memory usage for various operations like training, inference, and loading indices. The `ScalableSemanticSearch` class provides the `timed_train`, `timed_infer`, and `timed_load_index` methods to measure these benchmarks.

```
train_time, train_memory = semantic_search.timed_train(corpus)
```

```
infer_time, infer_memory = semantic_search.timed_infer(query, top)
```

The plots for both the training and inference performance in terms of execution time and memory usage can be found below. We will be discussing and interpreting the results in light of the selection of algorithms based on the size of the corpus we used.

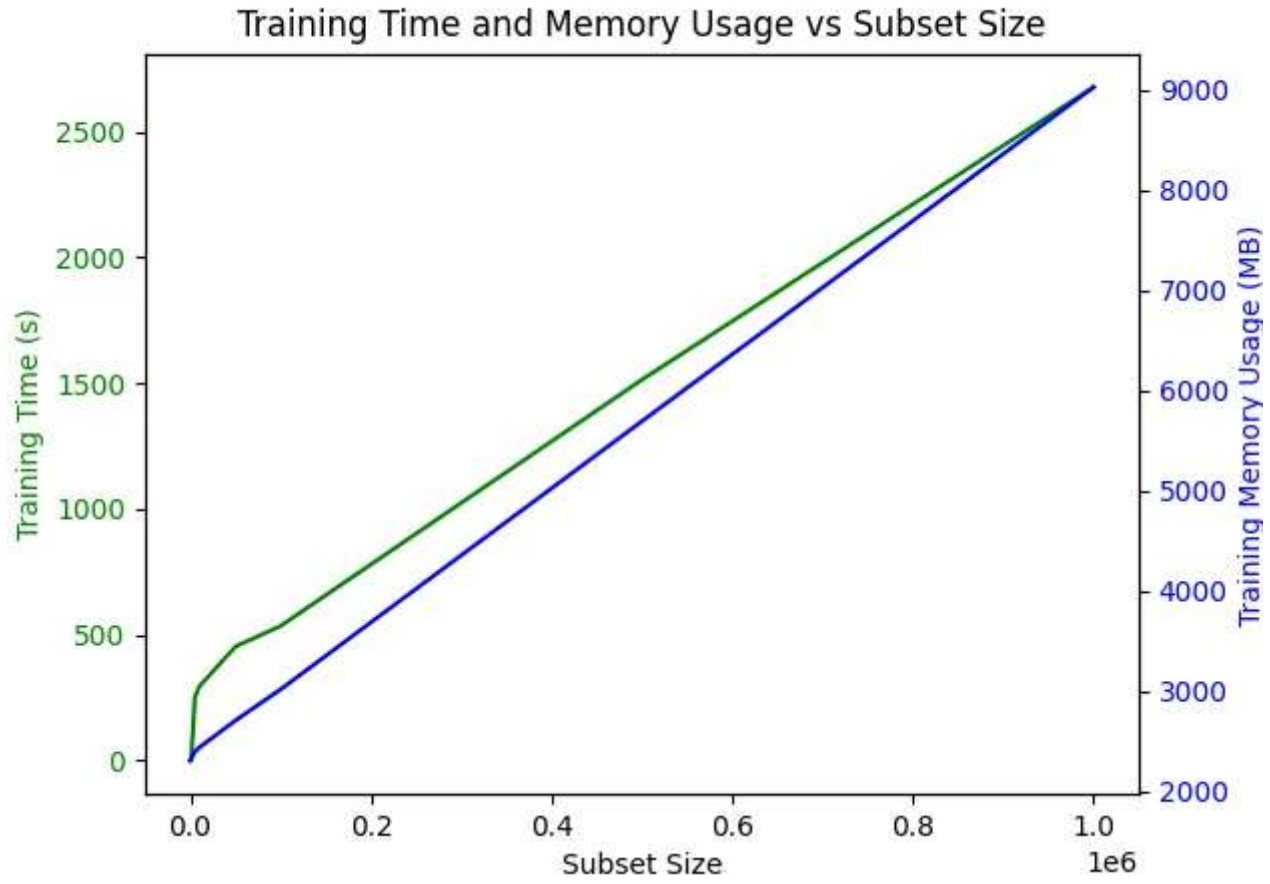


Figure 2: Training time and memory usage for different dataset sizes

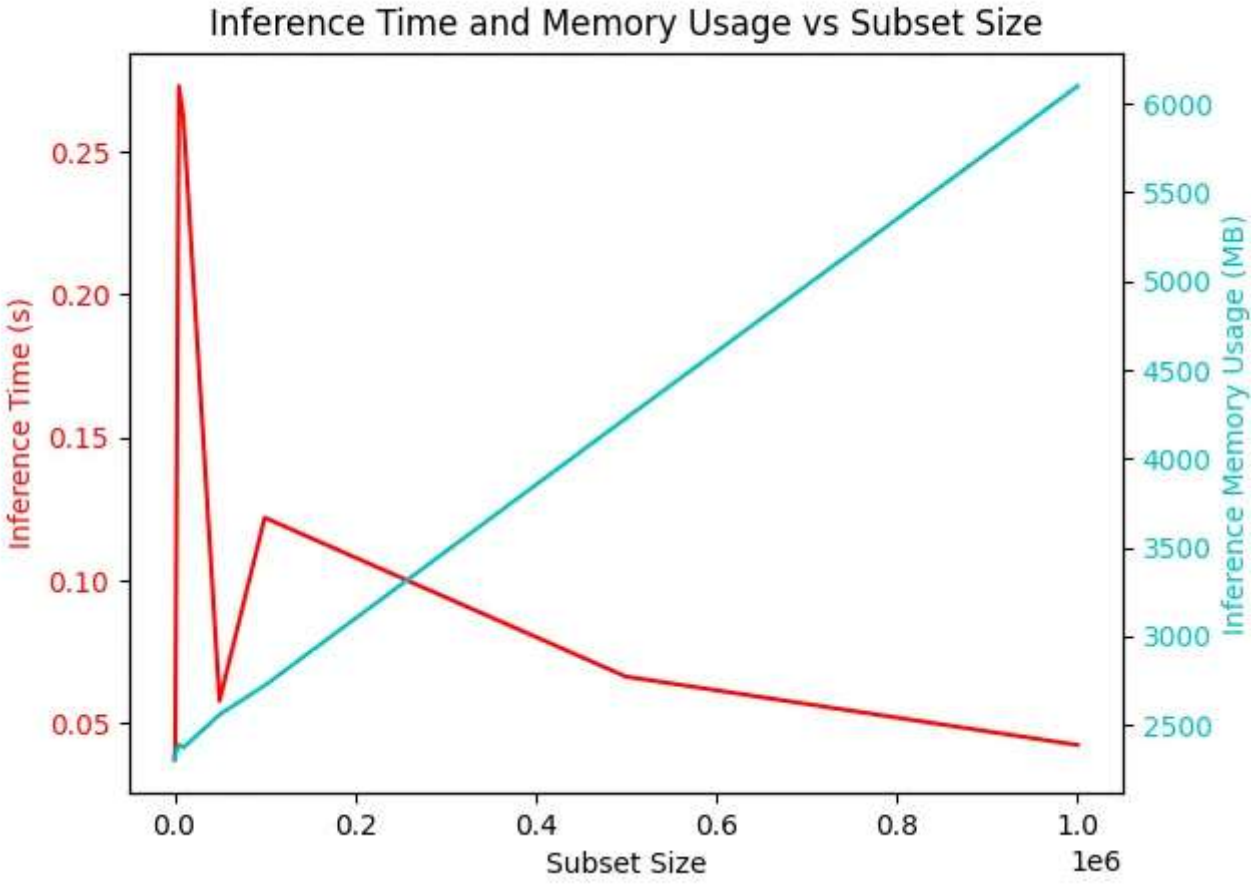


Figure 3: Inference time and memory usage for different dataset sizes

Exact Search using L2 distance

Exact Search using L2 distance is an exhaustive search method that performs a linear scan to find the nearest neighbors.

Theoretical Complexity

- Time Complexity: O(n) — Because it needs to compare the query vector with every vector in the dataset.
- Memory Complexity: O(n) — It stores all the vectors in the dataset.

Observed Complexity

- From the plots above, we can observe that for less than 1500 documents, both the training time and memory usage increase linearly with the number of documents, which matches the expected theoretical complexity.

Approximate Search using Product Quantization and L2 distance

Approximate Search using Product Quantization and L2 distance is an approximate nearest neighbor search method that employs product quantization and an inverted

file structure for improved efficiency. The number of clusters (k) is an essential factor in this method, and it is calculated using the formula: max(2, min(n_data_points, int(np.sqrt(n_data_points)))).

In simpler terms, this formula ensures that:

1. There are at least 2 clusters, providing a minimum level of partitioning.
2. The number of clusters doesn't exceed the number of data points.
3. As a heuristic, the square root of the number of data points is used to balance search accuracy and computational efficiency.

Theoretical Complexity

- Time Complexity (Training): O(n * k) — The complexity of the k-means clustering algorithm used in the training phase.
- Memory Complexity (Training): O(n + k) — It stores the centroids of clusters and residual codes.
- Time Complexity (Inference): O(k + m) — Where m is the number of nearest clusters to be searched. It is faster than linear search due to the hierarchical structure and approximation.
- Memory Complexity (Inference): O(n + k) — It requires storing the inverted file and the centroids.

Observed Complexity

From the plots above, we can observe that for more than 1500 documents:

- Training time complexity: The growth is faster than linear, which matches the expected theoretical complexity of O(n * k) since k grows with the number of data points (n).
- Training memory complexity: The memory usage increases non-linearly with the number of documents, which matches the expected theoretical complexity of O(n + k).
- Inference time complexity: The execution time remains almost constant, which is consistent with the expected theoretical complexity of O(k + m), as m is

usually much smaller than n.

- Inference memory complexity: The memory usage increases linearly with the number of documents, which matches the expected theoretical complexity of O(n + k).

We could also evaluate the accuracy and recall of the search engine by comparing the top results against a manually curated set of ground truth results for a given query. We can calculate the average accuracy and recall for the entire dataset by iterating over various queries and comparing the results.

Conclusion

The demonstrated approach highlights the scalability of semantic search using FAISS and Sentence Transformers while revealing enhancement opportunities. For instance, integrating advanced transformer models for encoding sentences or testing alternative FAISS configurations could speed up the search process. Additionally, investigating state-of-the-art models like GPT-4 or BERT variants might improve semantic search tasks' performance and accuracy.

Several potential applications for the scalable semantic search engine include:

- Retrieving documents in extensive knowledge bases
- Answering questions in automated systems
- Providing personalized recommendations
- Generating chatbot responses

Taking advantage of FAISS and Sentence Transformers, we developed a scalable semantic search engine capable of efficiently processing billions of documents and delivering accurate search results. This innovative approach can significantly influence the future of semantic search and its impact across various industries and applications.

As digital data grows, the demand for efficient and accurate semantic search engines becomes more critical. Based on FAISS and Sentence Transformers, the scalable semantic search engine lays a strong foundation for overcoming these challenges and revolutionizing how we search for and access relevant information.

Future advancements involve incorporating more advanced natural language processing and machine learning techniques to enhance search engine capabilities. These improvements could encompass unsupervised learning methods for better understanding context, intent, and relationships between query words and phrases and approaches for handling ambiguity and variations in language use.

Keep in touch: LinkedIn

Data Science | Machine Learning | Artificial Intelligence | Python | Deep Learning

Follow

Written by Luís Roque

1.4K Followers · Writer for Towards Data Science

Head of Data @ Marley Spoon | Ph.D. Researcher AI @ LIACC | Coordinator DS Masters @ NDS | CoFounder & ex-CEO @ HUUB

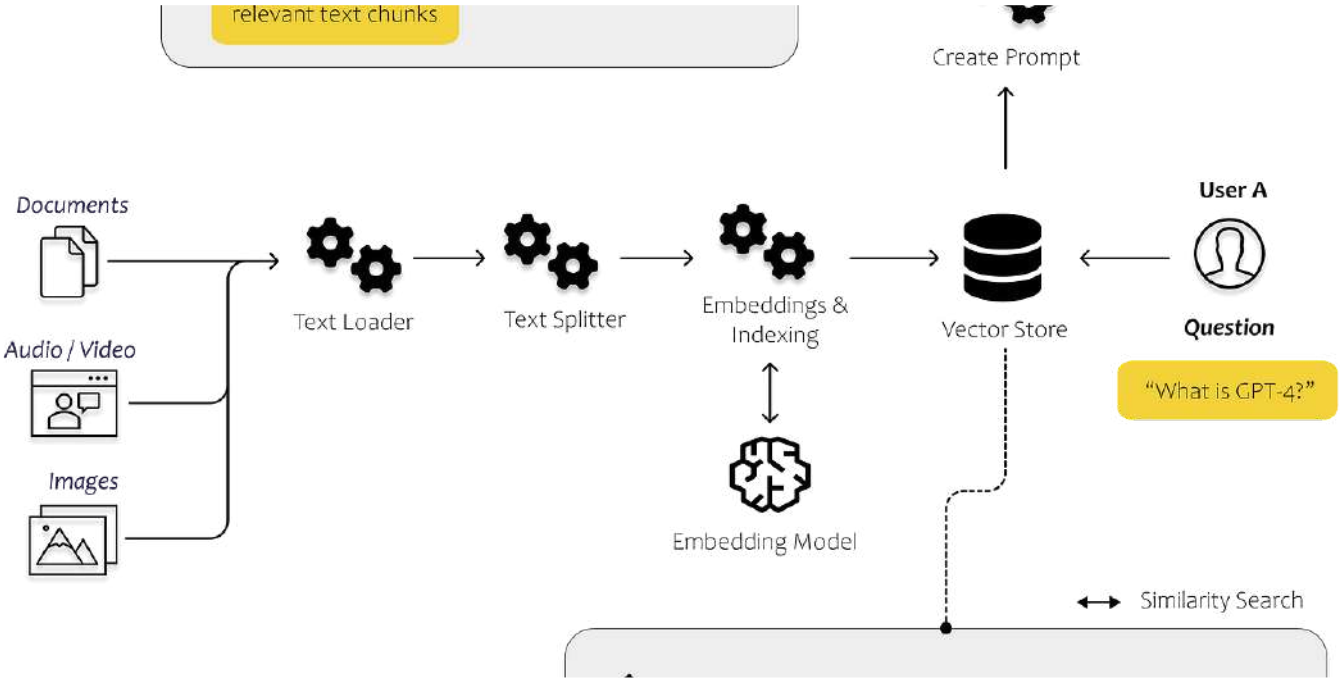
More from Luís Roque and Towards Data Science

Luís Roque in Towards Data Science

Document-Oriented Agents: A Journey with Vector Databases, LLMs, Langchain, FastAPI, and Docker

Leveraging ChromaDB, Langchain, and ChatGPT: Enhanced Responses and Cited Sources from Large Document Databases

· 11 min read · Jul 5



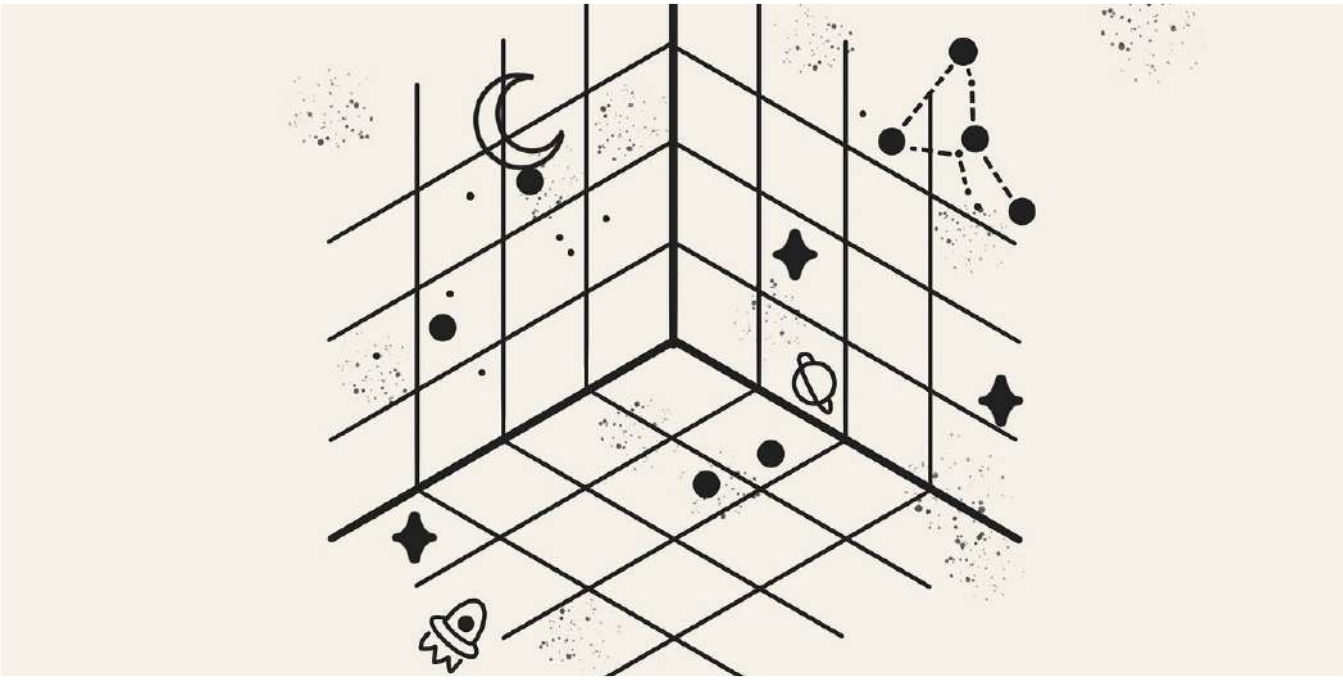
Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

✦ · 26 min read · Jun 22

3.7K 34



Leonie Monigatti in Towards Data Science

Explaining Vector Databases in 3 Levels of Difficulty

From noob to expert: Demystifying vector databases across different backgrounds

✦ · 8 min read · Jul 4

1.92K 21

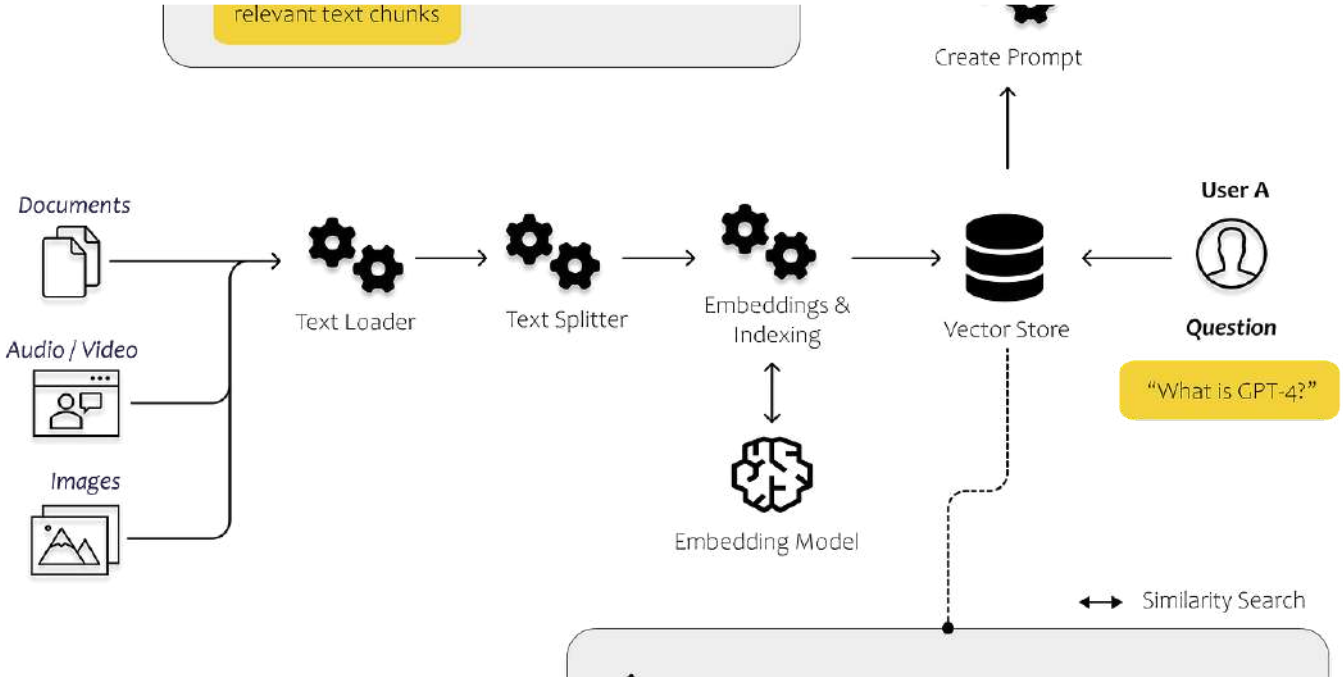
Luís Roque in Towards Data Science

Leveraging Llama 2 Features in Real-world Applications: Building Scalable Chatbots with FastAPI...

An In-Depth Exploration: Open vs Closed Source LLMs, Unpacking Llama 2's Unique Features, Mastering the Art of Prompt Engineering, and...

· 14 min read · Jul 24

352 5



Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

✦ · 26 min read · Jun 22

3.7K 34

Ryan Nguyen in How AI Built This

Zero to One: A Guide to Building a First PDF Chatbot with LangChain & LlamaIndex—Part 1

Welcome to Part 1 of our engineering series on building a PDF chatbot with LangChain and LlamaIndex. Don't worry, you don't need to be a...

15 min read · May 11

81 3

Lists

Predictive Modeling w/ Python

18 stories · 195 saves

Practical Guides to Machine Learning

10 stories · 211 saves

Natural Language Processing

455 stories · 89 saves

ChatGPT

21 stories · 79 saves

Lucas McGregor

No One Wants To Talk To Your Chatbot

5 min read · Jul 16

285 5

Sami Maameri in Better Programming

Private LLMs on Your Local Machine and in the Cloud With LangChain, GPT4All, and Cerebrium

The idea of private LLMs resonates with us for sure. The appeal is that we can query and pass information to LLMs without our data or...

17 min read · Jun 15

967 8

CUSTOMER DETAILS

Billing		Delivery	
Nick Bert	P:0401 320 816	Nick Bert	M:0401 320 816
134 Barker Street	M:0401 320 816	134 Barker Street	
NEW FARM Queensland 4005	Account#: WW-833332	NEW FARM Queensland 4005	
Australia		Australia	

DESCRIPTION:	QTY:	UNIT PRICE: (INC TAX)	TOTAL: (EX TAX)	TOTAL: (INC TAX)
Adreno Dive Trip - Wreck of the Marietta Dal & Smiths Rock Double Dive Trip (ID: 179285, AD-DiveMariettaSmiths,)	1	$210.00	$190.91	**$210.00**
Scuba Kit Hire (Tank, Weights, BCD, Regulator & Computer) (ID: 172605, AD-SCUBA-SET,)	1	$70.00	$63.64	**$70.00**
Adreno Wetsuit Hire	1	$30.00	$27.27	**$30.00**

in box

Use of Generative A.I. to achieve better OCR results than traditional tools

As generative AI advances, models like ChatGPT have proven adept at extracting text from unstructured data. These models can grasp the...

3 min read · Apr 12

95

Sushil Khadka

Self-Attention in Transformers

A Beginner-Friendly Guide to Self-Attention Mechanism

11 min read · May 15

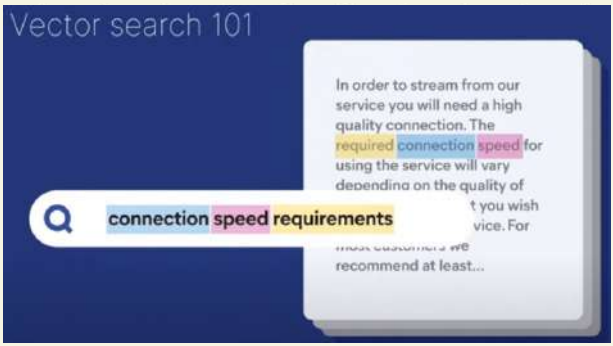
93 2

Vector Search Isn't Enough | BRKFP301H

Vector search:

-> Consider 3 words connection, speed requirement,

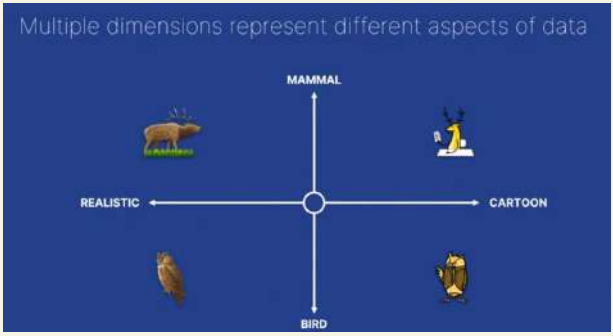
->

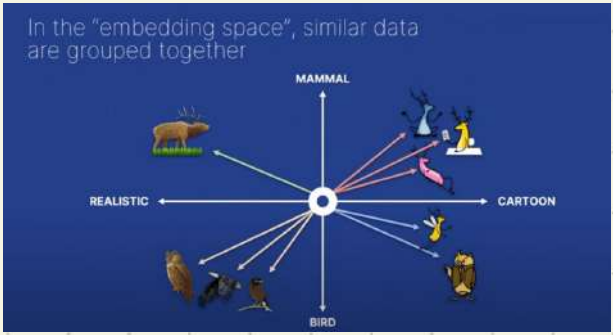


-> These 3 words are searched in document & here we have above 1-to-1 matched document

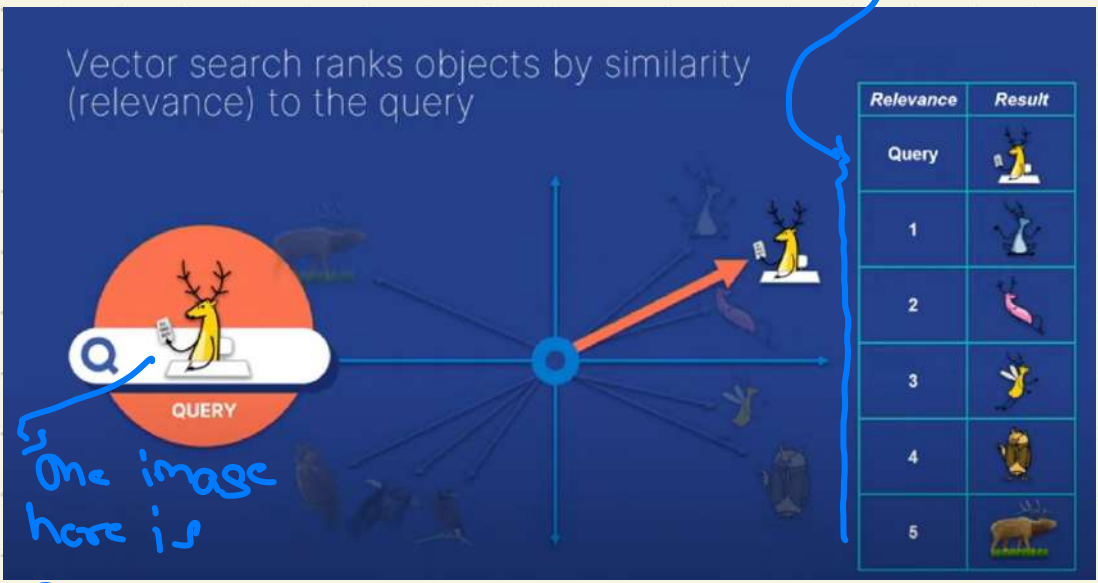
-> In vector search, we will find meaning of query & then do search e.g. above meaning from these 3 words together like "high-speed internet" or "is my connection fast enough?" such meanings are searched through document where words query given by user need not to be compulsory matched

-> Dimensions in data

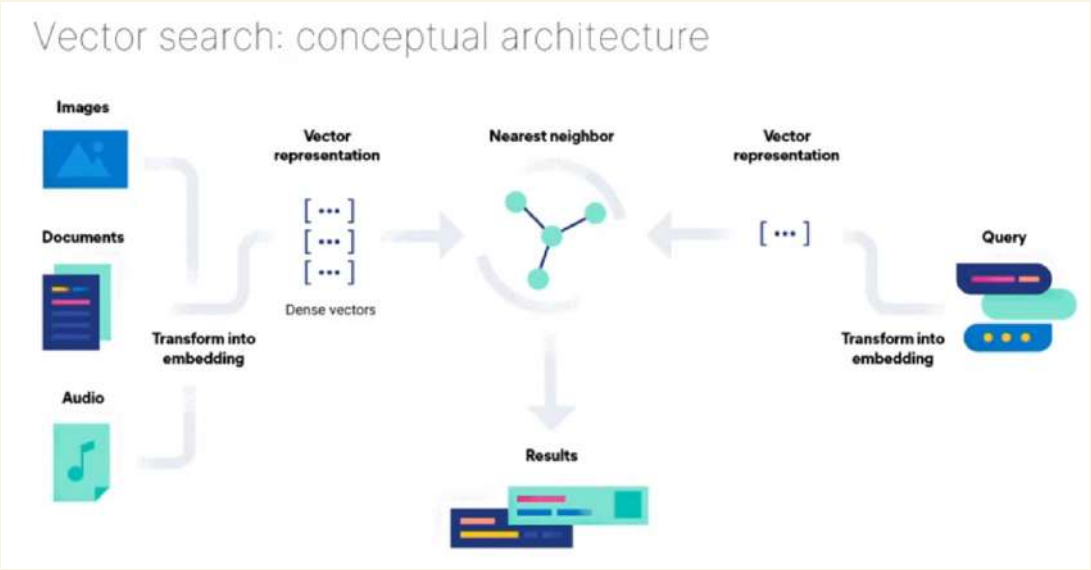


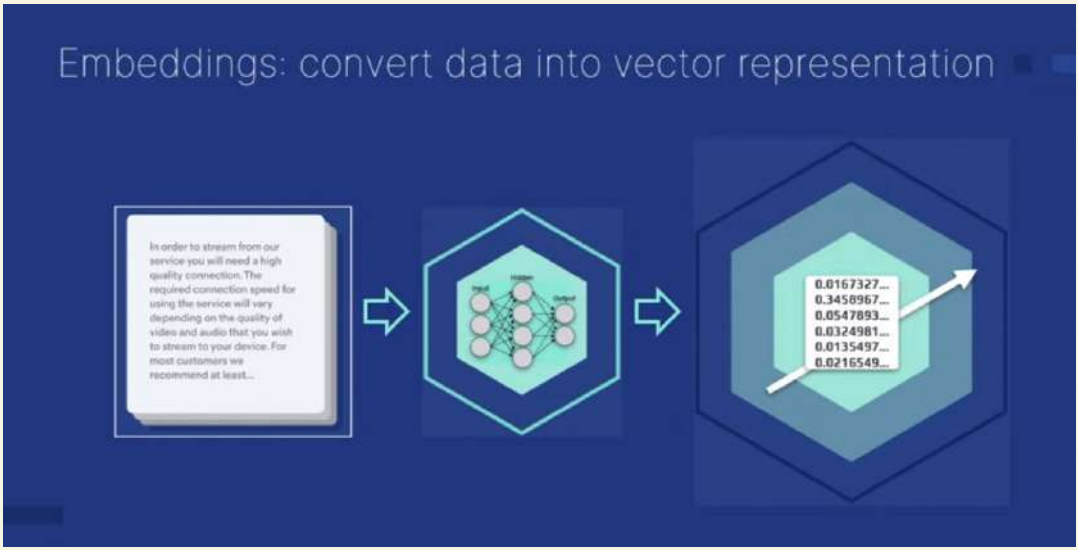


Ranking given

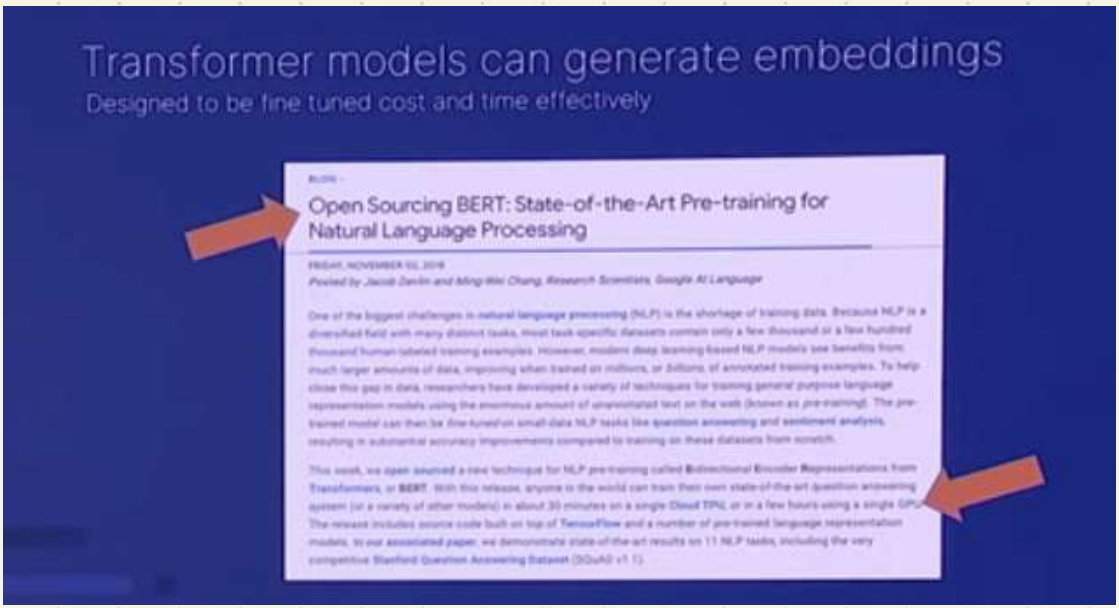


One image here is query



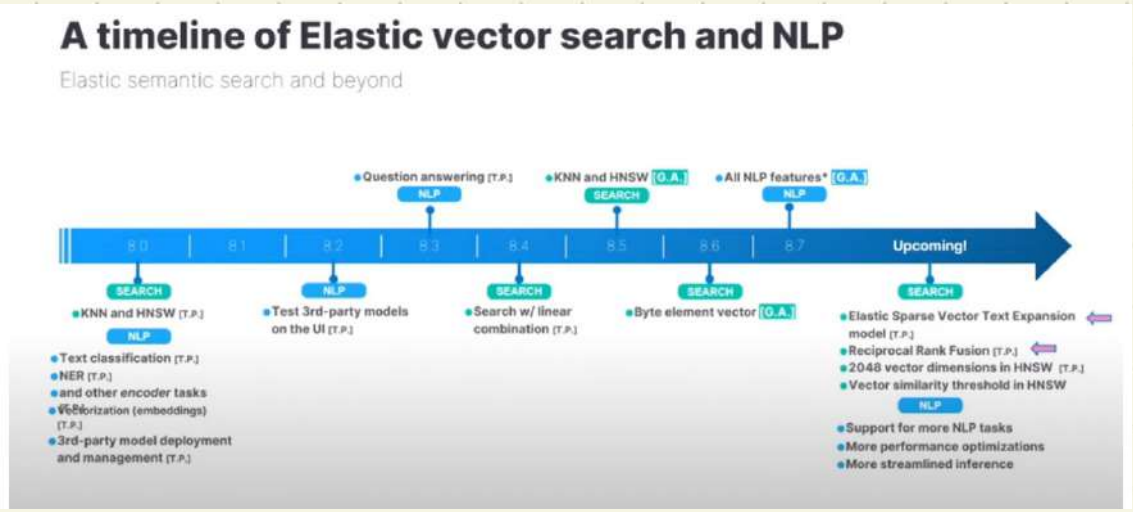


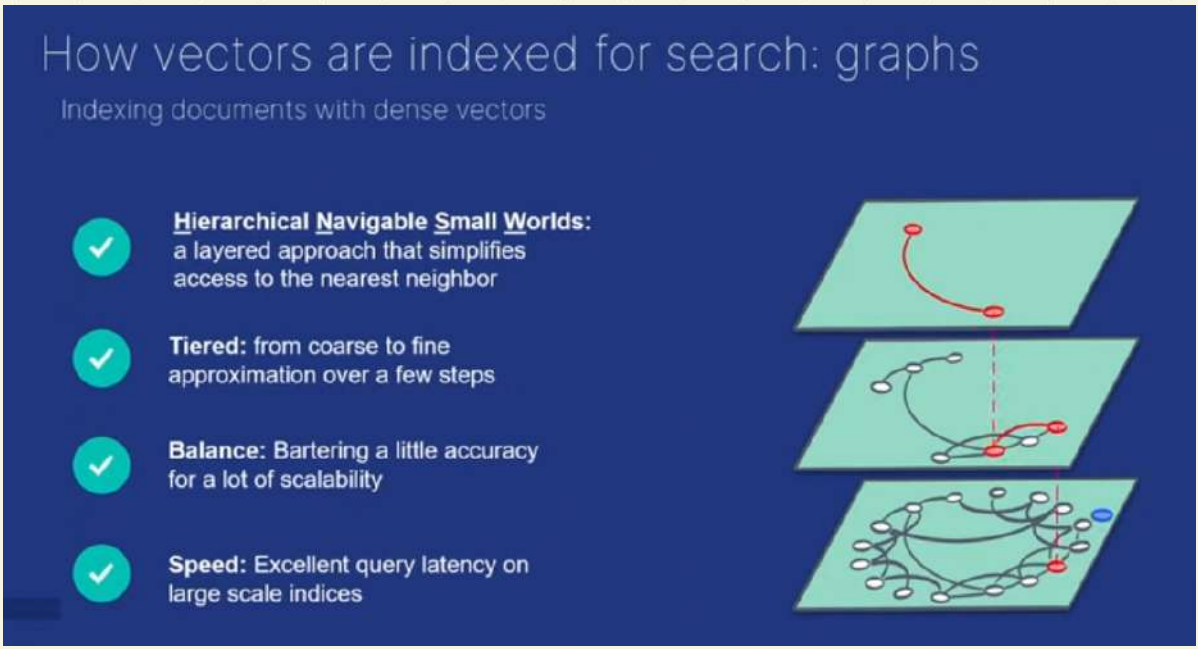
→ Normal unstructured data cannot be stored.
→ So embeddings are used



→ For unstructured text, converting to embeddings is done by BERT

→ BERT models are designed to trained in layers. Meaning a machine understanding what is really a language it is trained in layers



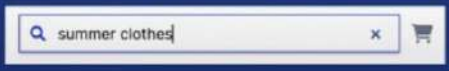


HNSW: Layered approach to help you get to nearest neighbor

Find nearest neighbor, approximately

You can also score similarity exactly

Query is submitted to the search-powered application:

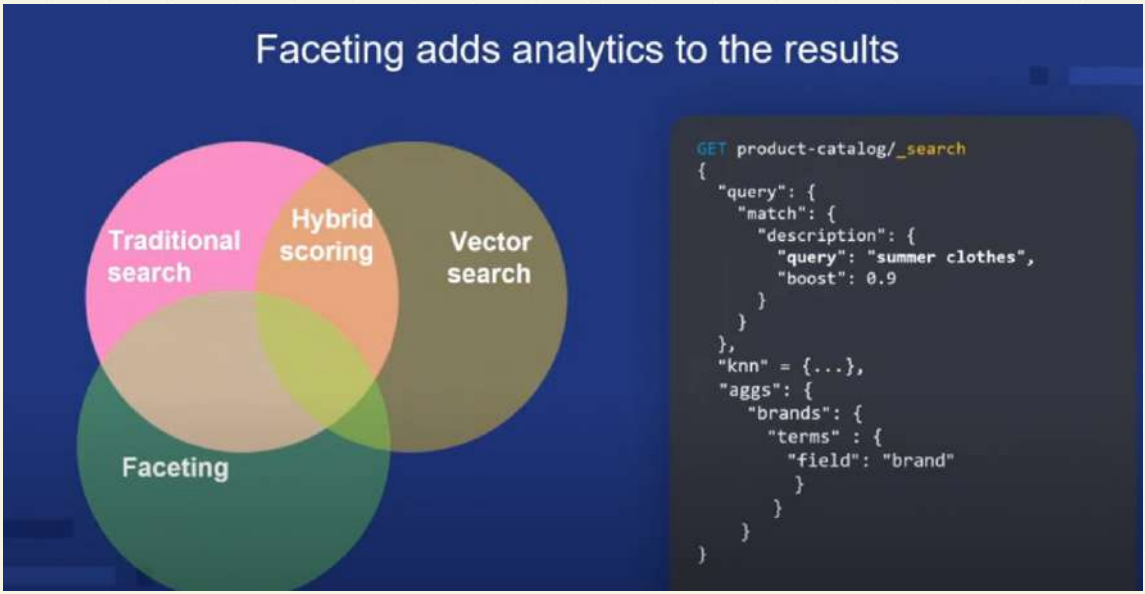


Issue kNN query using the _search endpoint

```
GET product-catalog/_search

{
"knn" :{
        "field": "title-vector",
        "k": 1,
        "num_candidates": 20,
        "query_vector_builder": {
            "text_embedding": {
                "model_id": "sentence-
transformers__all-distilroberta-v1",
                "model_text": "summer clothes"
            }
        },
        "boost": 2
    }
}
```

Hybrid scoring gets you the best of both worlds


```
GET product-catalog/_search
{
  "query": {
    "match": {
      "description": {
        "query": "summer clothes",
        "boost": 0.9
      }
    }
  },
  "knn": {
        "field": "title-vector",
        "k": 1,
        "num_candidates": 20,
        "query_vector_builder": {
            "text_embedding": {
                "model_id": "sentence-
transformers__all-distilroberta-v1",
                "model_text": "summer clothes"
            }
        },
        "boost": 2
    }
}
```



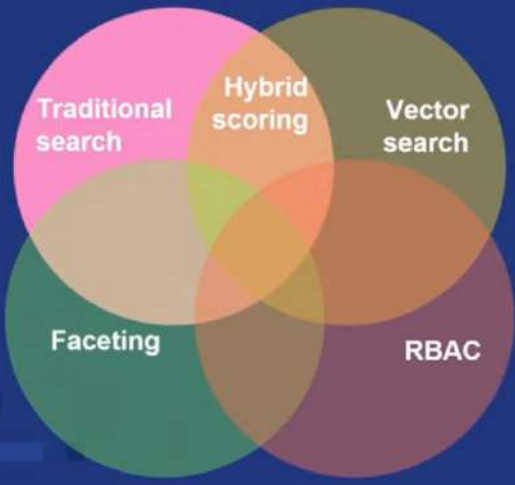
→ Dynamically adding additional results based on search result

→ Suppose user has searched summer clothes, you get the result, now you visit one product

→ On that visited product page additional products are also shown based on attributes of product you choosed to visit like brand or color or quality of clothes

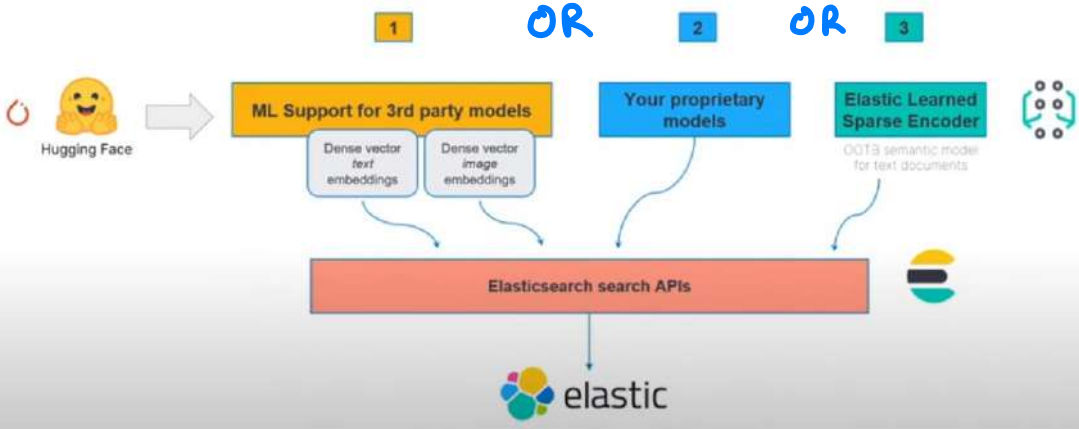
→ These are additional results shown are called faceting

Role Based Access Control secures sensitive data




```
GET employee-database/_search
{
  "query": {...},
  "knn" = {...},
  "aggs": {...}
}


POST /_security/role/employee_role
{
  "indices": [
    {
      "names": [ "employee-database" ],
      "privileges": [ "read" ],
      "field_security" : {
        "grant" : [ "state", "name.first",
"name.last" ]
      }
    }
  ]
}
```

Model management with Elastic



Use Elastic with a range of supported NLP models

Fill mask model
Mask some of the words in a sentence and predict words that replace masks

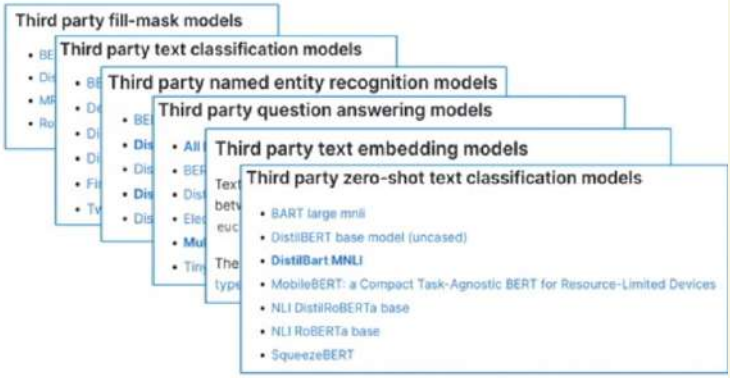
Named entity recognition model
NLP method that extracts information from text

Text embedding model
Represent individual words as numerical vectors in a predefined vector space

Text classification model
Assign a set of predefined categories to open-ended text

Question answering model
Model that can answer questions given some or no context

Zero-shot text classification model
Model trained on a set of labeled examples, that is able to classify previously unseen examples



For full list of Elastic supported models, use QR code or go to: ela.st/nlp-supported-models

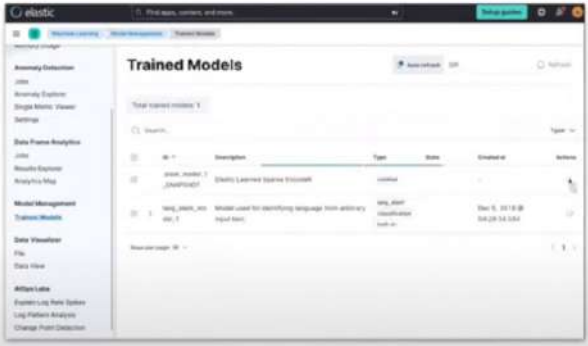
AI-powered search: Elastic Learned Sparse Encoder

Drivers

- You know, for... semantic search!
- Elastic's new relevance engine
- Democratise AI-powered search

Value

- Superior semantic relevance: Search based on ***meanings***, instead of just terms
- State-of-the-art, beats competition in industry benchmarks
- Start using it OOTB with one click with our streamlined search APIs



Native Semantic Search OOTB

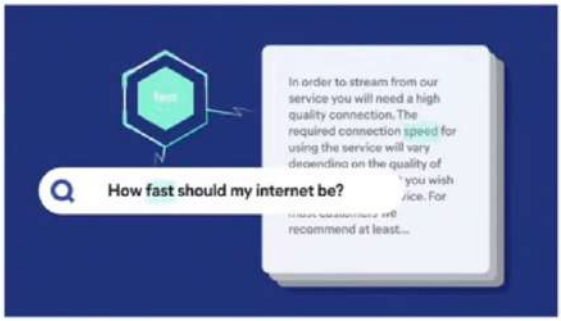
AI-powered search: Elastic Learned Sparse Encoder

The insurmountable barrier to AI-search

- Significant expertise and resources
- Processes significantly different than conventional software productization

Enter Elastic Learned Sparse Encoder

- Out-of-domain -> Out-of the box
- Solves the vocabulary mismatch problem
- More interpretable than dense vector search results
- State of the art: outperforms SPLADE, the previous champ in the category



Superior relevance across domains, without training

- Elastic Learned Sparse Encoder outperforms BM25 in **11** out of the 12 benchmarks
- Elastic Learned Sparse Encoder outperforms SPLADE in **8** out of the 12 benchmarks
- More granular statistics also on the positive side and expecting further improvements

Data sets (BEIR benchmark)

Search results ranking models

	Average	TREC-COVID	NFCorpus	NQ	HotpotQA	FIQA	ArguAna	Touche-2020	DBPedia	SCIDOCS	FEVER	Climate-FEVER	SciFact
BM25	0.416	0.688	0.327	0.326	0.602	0.254	0.472	0.347	0.297	0.165	0.649	0.186	0.69
RRF (BM25/Dense)	0.449	0.697	0.331	0.446	0.611	0.319	0.474	**0.354**	0.353	0.169	0.746	0.238	0.671
Linear (BM25/Dense)	0.471	0.787	0.335	0.485	0.62	0.347	0.444	0.346	0.378	0.164	**0.778**	**0.272**	0.698
SPLADE	N/A	0.726	0.347	**0.537**	0.687	**0.347**	**0.526**	0.249	**0.436**	0.158			0.703
Elastic Learned Sparse Encoder	0.471	0.747	**0.351**	0.524	0.67	0.335	0.5	0.363	0.415	0.159	0.727	0.238	0.696
RRF (BM25 / Elastic Learned Sparse Encoder)	**0.478**	**0.797**	**0.352**	0.488	0.674	0.31	0.497	0.347	0.41	**0.166**	0.762	0.24	**0.712**

Search quality *worse* than benchmark (BM25) — Search quality *better* than benchmark (BM25)

BM25: default similarity ranking algorithm for Elasticsearch
RRF: Reciprocal Rank Fusion
SPLADE: Sparse Lexical and Expansion Model

elastic

Reciprocal Rank Fusion

An alternative to linear hybrid scoring!

Problem: linear combination works well only if scores are normalized

Solution: RRF!

- Based on ranking, not on scores
- Great results without normalization and tuning

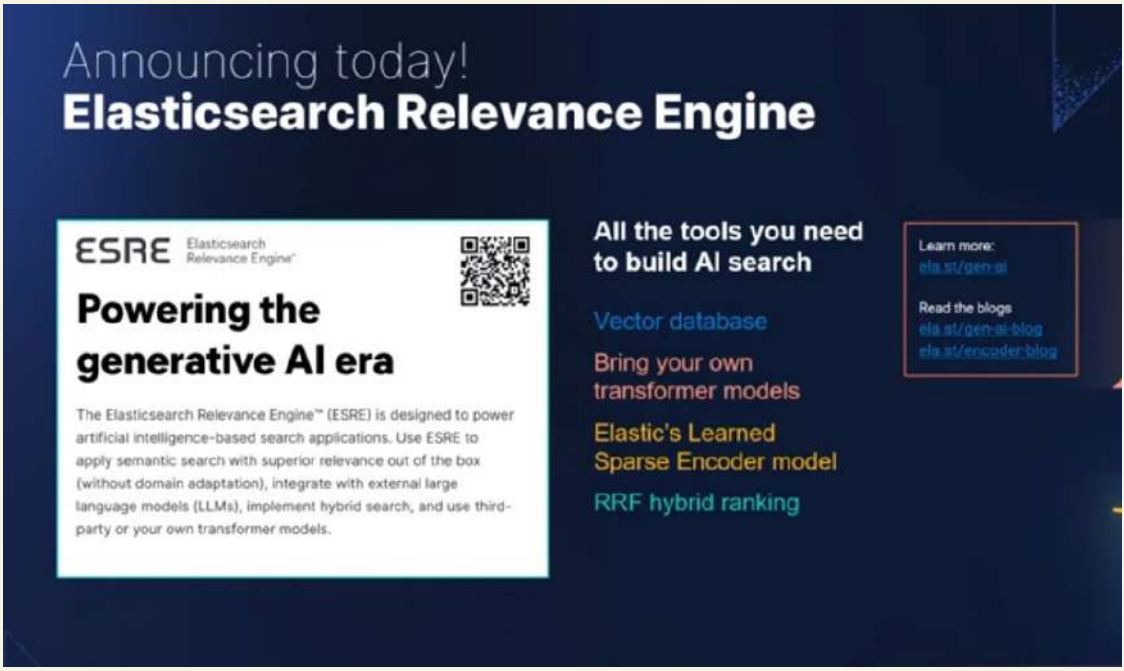
$$RRFscore(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

D - set of docs
R - set of rankings as permutation on 1..|D|
K - typically set to 60 by default

Ranking Algorithm 1		
Doc	Score	r(d)
A	1	1
B	0.7	2
C	0.5	3
D	0.2	4
E	0.01	5

Ranking Algorithm 2		
Doc	Score	r(d)
C	1,341	1
A	739	2
F	732	3
G	192	4
H	183	5

Doc
A
C
B
F
D



Elastic: search experiences are rapidly evolving

Q&A more intuitive search

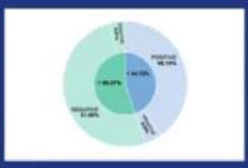
"What are the troubleshooting steps for ___?"

Image search

"Do you sell black v-neck shirts that look like this?"

Personalization

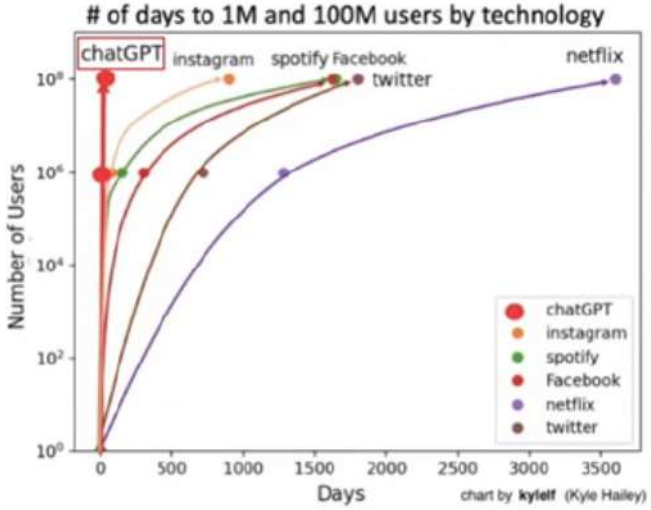
"What 70" TVs do you have on sale?"



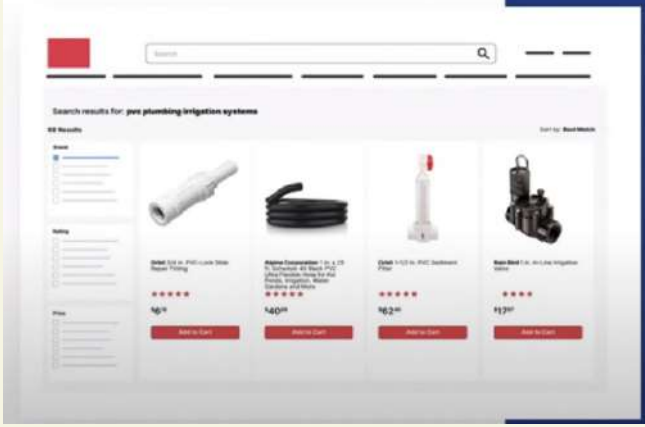
Sentiment analysis

Identify poor customer interactions before they lead to escalations

Unprecedented rapid adoption of AI



Ecommerce search



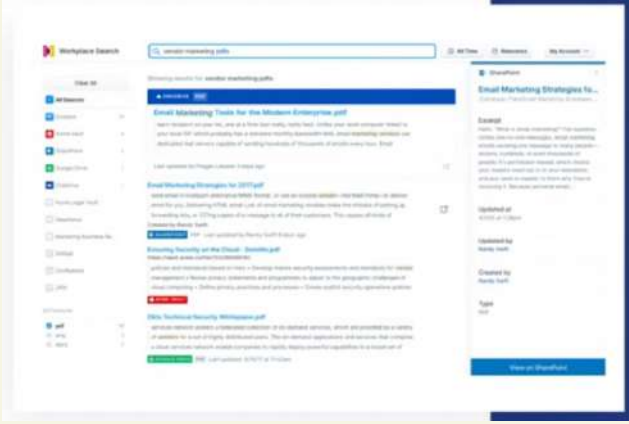
TODAY

pvc plumbing irrigation systems

TOMORROW

What material list and tools do I need to build an irrigation system for my 1 acre back yard in Detroit, MI?

Workplace search



TODAY

401k policy usa

TOMORROW

What are the key aspects of the company's 401k policy for an employee in my location and how do I enroll?

ChatGPT has created a lot of interest in this area

Excels at human-like, iterative content creation

Natural language processing of large data sets to create human-like conversations, write content, and provide code examples

Limited by public training data, generic context

It's only as good as the data it was trained on - and it can hallucinate (confident-sounding but erroneous output) especially when asked domain-specific questions

These limitations in domain-specific topics can be addressed with various techniques

Fine tuning

Further training the base LLM with the domain-specific content that is specific to the use case. Unfortunately, expensive to train, and expensive to host custom models.

Question context windows

Passing additional, relevant content to the LLM, alongside the original question, as a "context window". Straightforward, but context window size is limited.

LLMs require large scale resources

May not fit on your laptop!

Home / Blog / Announcements

Azure previews powerful and scalable virtual machine series to accelerate generative AI

Posted on March 13, 2023

Matt Vegas, Principal Product Manager, Azure HPC + AI

Delivering on the promise of advanced AI for our customers requires supercomputing infrastructure, services, and expertise to address the exponentially increasing size and complexity of the latest models. At Microsoft, we are meeting this challenge by applying a decade of experience in supercomputing and supporting the largest AI training workloads to create AI infrastructure capable of massive performance at scale. The Microsoft Azure cloud, and specifically our graphics processing unit (GPU) accelerated virtual machines (VMs), provide the foundation for many generative AI advancements from both Microsoft and our customers.

"Co-designing supercomputers with Azure has been crucial for scaling our demanding AI training needs, making our research and alignment work on systems like ChatGPT possible."—Greg Brockman, President and Co-Founder of OpenAI

Azure's most powerful and massively scalable AI virtual machine series

Today, Microsoft is introducing the ND H100 v5 VM which enables on-demand in sizes ranging from eight to thousands of NVIDIA H100 GPUs interconnected by NVIDIA Quantum-2 InfiniBand networking. Customers will see significantly faster performance for AI models over our last generation ND A100 v4 VMs with innovative technologies like:

- 8x NVIDIA H100 Tensor Core GPUs interconnected via next gen NVSwitch and NVLink 4.0
- 400 Gb/s NVIDIA Quantum-2 CX7 InfiniBand per GPU with 3.2Tb/s per VM in a non-blocking fat-tree network
- NVSwitch and NVLink 4.0 with 3.6TB/s bisectional bandwidth between 8 local GPUs within each VM
- 4th Gen Intel Xeon Scalable processors
- PCIE Gen5 host to GPU interconnect with 64GB/s bandwidth per GPU
- 16 Channels of 4800MHz DDR5 DIMMs

In 2019, Microsoft and OpenAI entered a partnership, which was extended this year, to collaborate on new Azure AI supercomputing technologies that accelerate breakthroughs in AI, deliver on the promise of large language models and help ensure AI's benefits are shared broadly.

The two companies began working in close collaboration to build supercomputing resources in Azure that were designed and dedicated to allow OpenAI to train an expanding suite of increasingly powerful AI models. This infrastructure included thousands of NVIDIA AI-optimized GPUs linked together in a high-throughput, low-latency network based on NVIDIA Quantum InfiniBand communications for high-performance computing.

The scale of the cloud-computing infrastructure OpenAI needed to train its models was unprecedented – exponentially larger clusters of networked GPUs than anyone in the industry had tried to build, noted Phil Waymouth, a Microsoft senior director in charge of strategic partnerships who helped negotiate the deal with OpenAI.

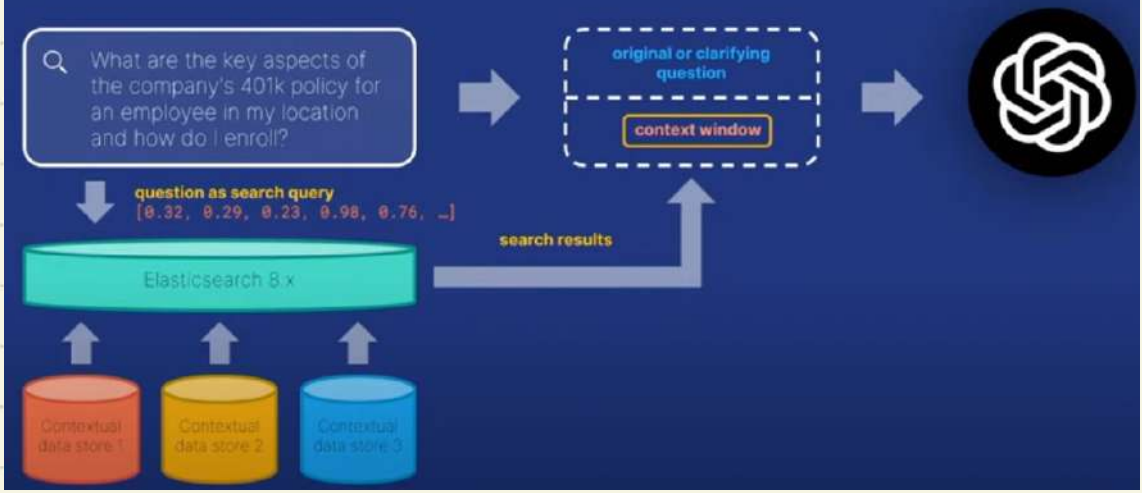
A word on pricing

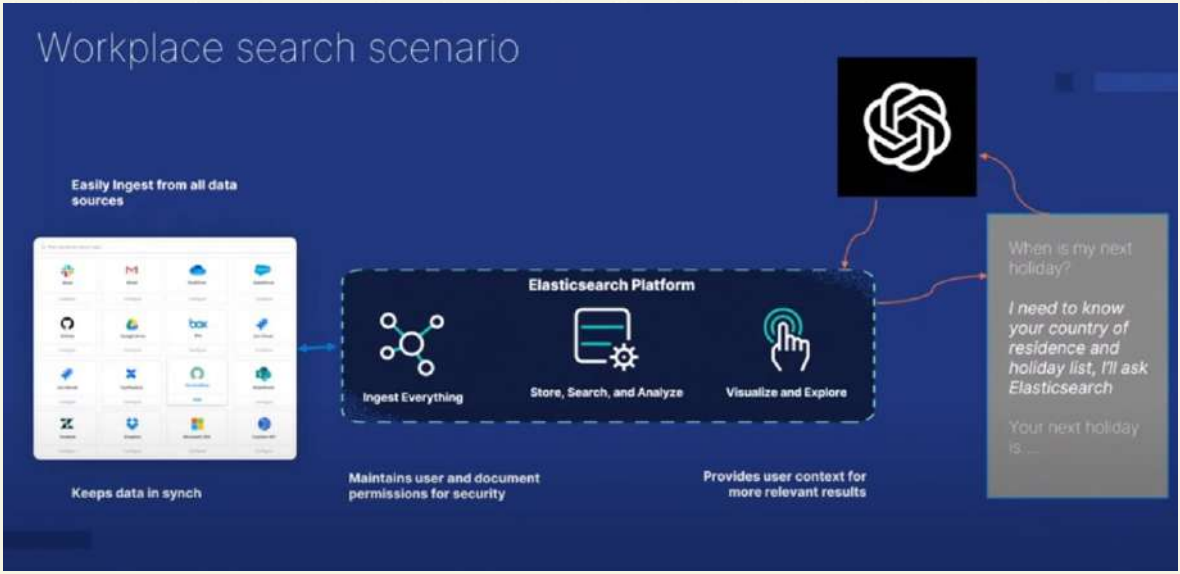
- Go to 👉 openai.com/pricing#language-models
- Multiple models, each with different capabilities and price points, prices are per 1,000 tokens
- Think of tokens as pieces of words (1,000 tokens is about 750 words), text on this slide is ~30 tokens
- gpt-3.5-turbo$0.002 / 1K tokens

A pricing example

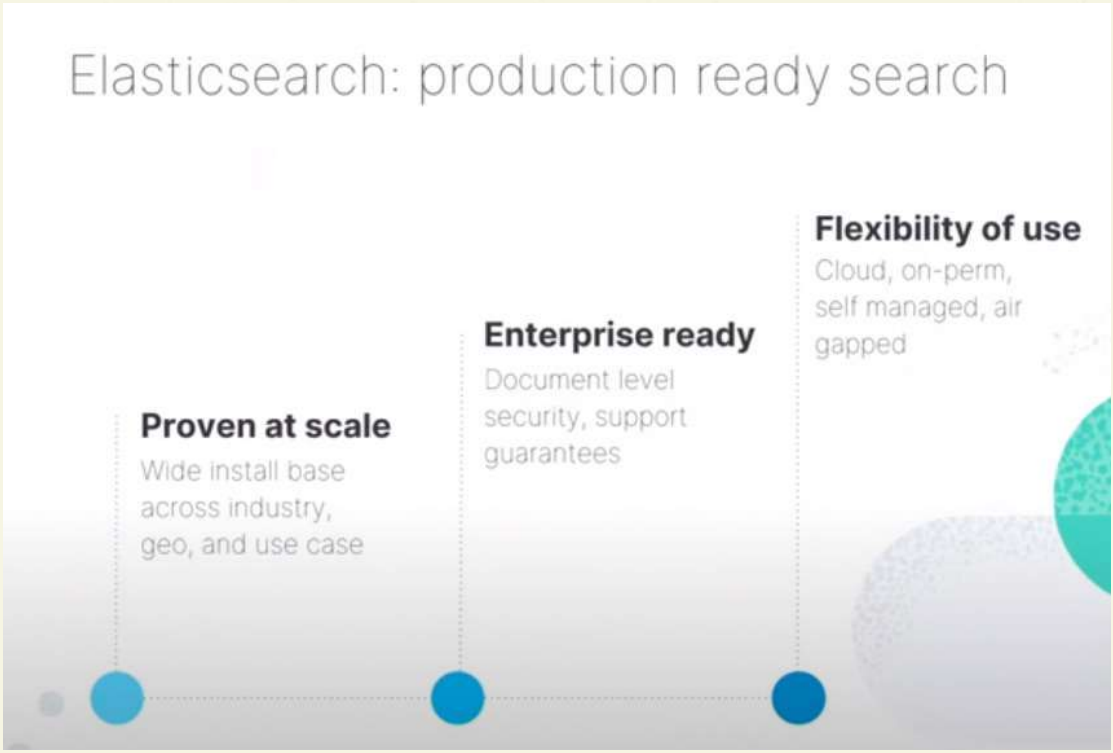
- **War and Peace** (Tolstoy 1865-69) has 587,287 words, it is about 3MB in size
  - ChatGPT processing cost - estimated at $1.56
- **Daily generated information** volume is currently estimated to be 328.77 million terabytes or approximately
  - ChatGPT processing cost - estimated at $170,960,400,000,000 !!

Conceptual context window workflow




Workplace search scenario
Easily Ingest from all data sources
Elasticsearch Platform
Ingest Everything
Store, Search, and Analyze
Visualize and Explore
When is my next holiday?
I need to know your country of residence and holiday list, I'll ask Elasticsearch
Your next holiday is ...
Keeps data in synch
Maintains user and document permissions for security
Provides user context for more relevant results


Several investments that add up
Vector search
Native support for fast similarity in vector space, alongside the existing retrieval models built into Elasticsearch
Native transformer models
Create vector embeddings from natural language text directly in your Elasticsearch deployment
Sophisticated relevance ranking
Semantic search intuitively integrated into _search API and RRF for best-in-class ranking, filtering, document-level permission enforcement


Elasticsearch: production ready search
Proven at scale
Wide install base across industry, geo, and use case
Enterprise ready
Document level security, support guarantees
Flexibility of use
Cloud, on-perm, self managed, air gapped

Elastic Search use cases =>

Machine learning use cases across search, security and observability

Search

- Recommendations
- Personalization
- Semantic similarity
- Image search
- Q&A
- Sentiment analysis
- Vector database
- Hybrid ranking

Security

- Threat intelligence
- Automated alerting
- Security chatbot

Observability

Anomaly detection

AIOps

Interactive root cause Analysis

Integrated case management

Custom Models

- Data engineering
  (Extract, Analyze, Transform)
- Feature engineering
- Modeling & APIs
  (Build, Serve, Learn)
- NLP/supervised/ unsupervised/ vector embedding
- Inference pipelines, model management
- Built-in model

elastic

IR12 Locality Sensitive Hashing

https://www.youtube.com/playlist?list=PLBv09BD7ez_6xoNh_luPdBmDCIHOQ3j7F

LSH.1 Exact duplicates and near-duplicates

https://www.youtube.com/watch?v=356GoYkmYKg&list=PLBv09BD7ez_6xoNh_luPdBmDCIHOQ3j7F

Detecting duplicates

- About 30% of web-pages are duplicates (2003)
  - mirrors, plagiarism, spam
  - also happens in news-feeds, speeches, essays
- Important to detect and remove
  - exact duplicate:
    - the content is identical bit-for-bit
    - non-content elements may differ
  - near-duplicate:
    - very similar but not exact: e.g 90% of content is the same
    - reworded a few sentences, or used different keywords

LSH.2 Duplicate detection: naive approach

https://www.youtube.com/watch?v=4h_cUtXQpzI&list=PLBv09BD7ez_6xoNh_luPdBmDCIHOQ3j7F&index=2

Detecting duplicates Naively

- Compare every document against all others

```
for b = 1..n:
  for a = 1..b-1:
    if doc[a] == doc[b]: b is a duplicate of a
    else: b is not a duplicate
```

- Computational complexity: $O(n^2d)$ [impractical]
  - n … number of documents in a set (10^6 – 10^9)
  - d … size of each document (10^3 words / 10^4 bytes)

LSH.3 Detecting duplicates by hashing

https://www.youtube.com/watch?v=hIBN1fMBHzQ&list=PLBv09BD7ez_6xoNh_luPdBmDCIHOQ3j7F&index=3

Duplicates via Fingerprinting

- Idea: make duplicates fall into same bucket
- For each new document:
  - compute its fingerprint
    - large number, which is:
      - always same for two identical docs
      - never same for two different docs
  - hash [fingerprint]
    - collision → test if duplicate
- Complexity: O(nd)

→ Detect duplicates by hashing and fingerprint

→ Hash table is maintained. And that hash table is indexed by

fingerprints: (usually a number)

→ For every doc, compute fingerprint of it and insert them into bucket of hash table.

→ And if multiple documents falls in same bucket of hash table then they are carrying identical fingerprints and that also means it has identical content

→ Rule:
if two documents are same fingerprint will be same otherwise it will be different.

→ For near duplicates, things will be different. fingerprint will not be same

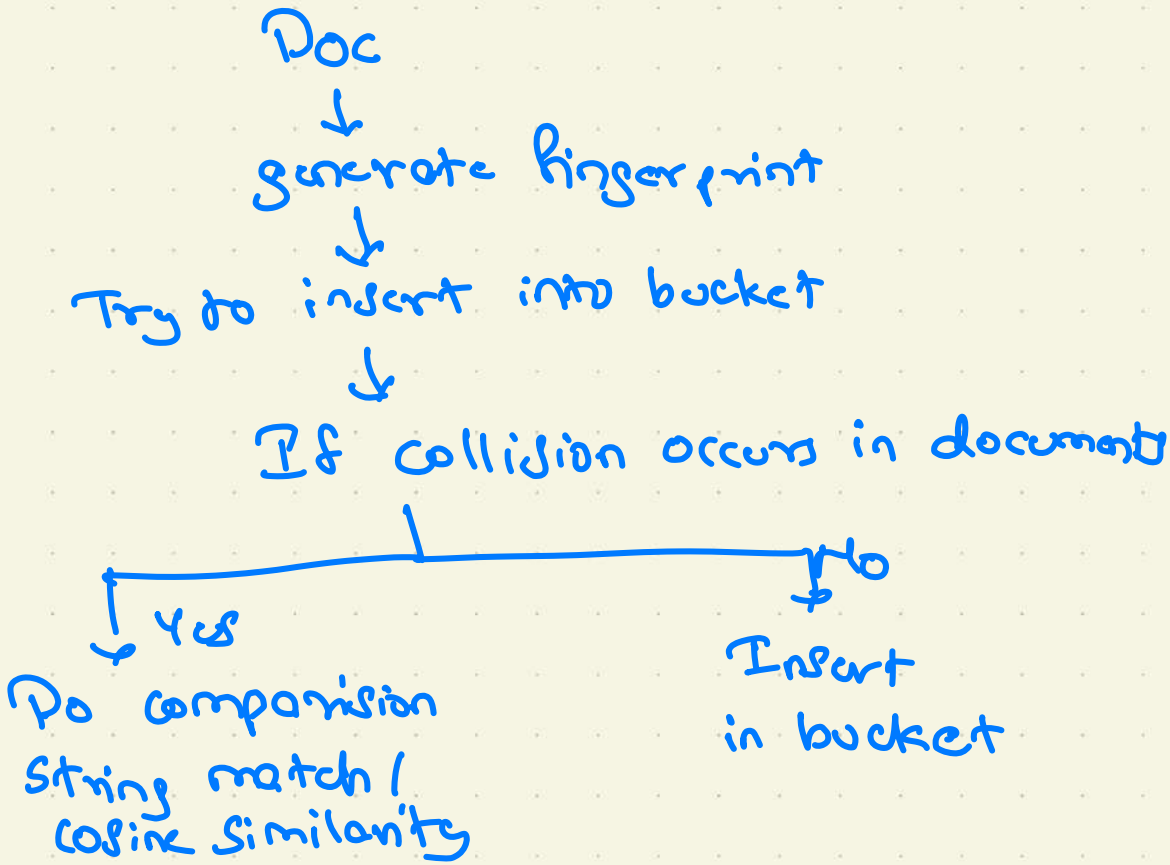
→ Consider 2 fingerprints of two different documents

→ You will be adding fingerprints as keys in hashtable

→ Since hash table has finite number of memory, sooner, later there will be collisions by occurance of 2 fingerprints being same or 2 fingerprints which are slightly different but got landed in same bucket

→ If 2 documents are coming in same bucket then you can test these 2 documents by string comparison or cosine similarity and figure out are they really duplicate or near duplicate. Or it came accidently due to hash table ran out of new slots of memory

-) Flow of insertion in hash table



LSH.4 Adler32 hashcode

https://www.youtube.com/watch?v=BWqH4O7OuyY&list=PLBv09BD7ez_6xoNh_luPdBmDCIHOQ3j7F&index=4

Fingerprinting: Adler32

- Generate a unique fingerprint for each doc
  - change 1 bit in a doc → get a different print
  - maintain low chance of false collision
  - remove non-content (punctuation, tags)
  - add up byte values modulo a prime

```
adler32(str) {
    A = B = 0; m = 65521;
    while (c = *str++) A += (B += c);
    return ((A % m) << 16) | (B % m); }
```

	a		c	a	b	
ASCII	97	32	99	97	98	modulo 13
sum B	97	129	228	325	423	7 **"111"**
sum A	97	226	454	779	1202	6 **"110"**

- Designed for data integrity applications:
  - "magnify" small changes in input
    - transmission error, bad bit in RAM, bad sector on disk

→ fingerprints: can be called as hash of file / md5 / signatures / checksum

[Read video of how Adler32 works. Skipping as it is not my part]

LSH.8 Locality-sensitive hashing: the idea

https://www.youtube.com/watch?v=dgH0NP8Qxa8&list=PLBv09BD7ez_6xoNh_luPdBmDCIHOQ3j7F&index=8&pp=iAQB