



Search Medium

Write



Similarity Search, Part 1: kNN & Inverted File Index

Introduction to similarity search with kNN and its acceleration with inverted file.

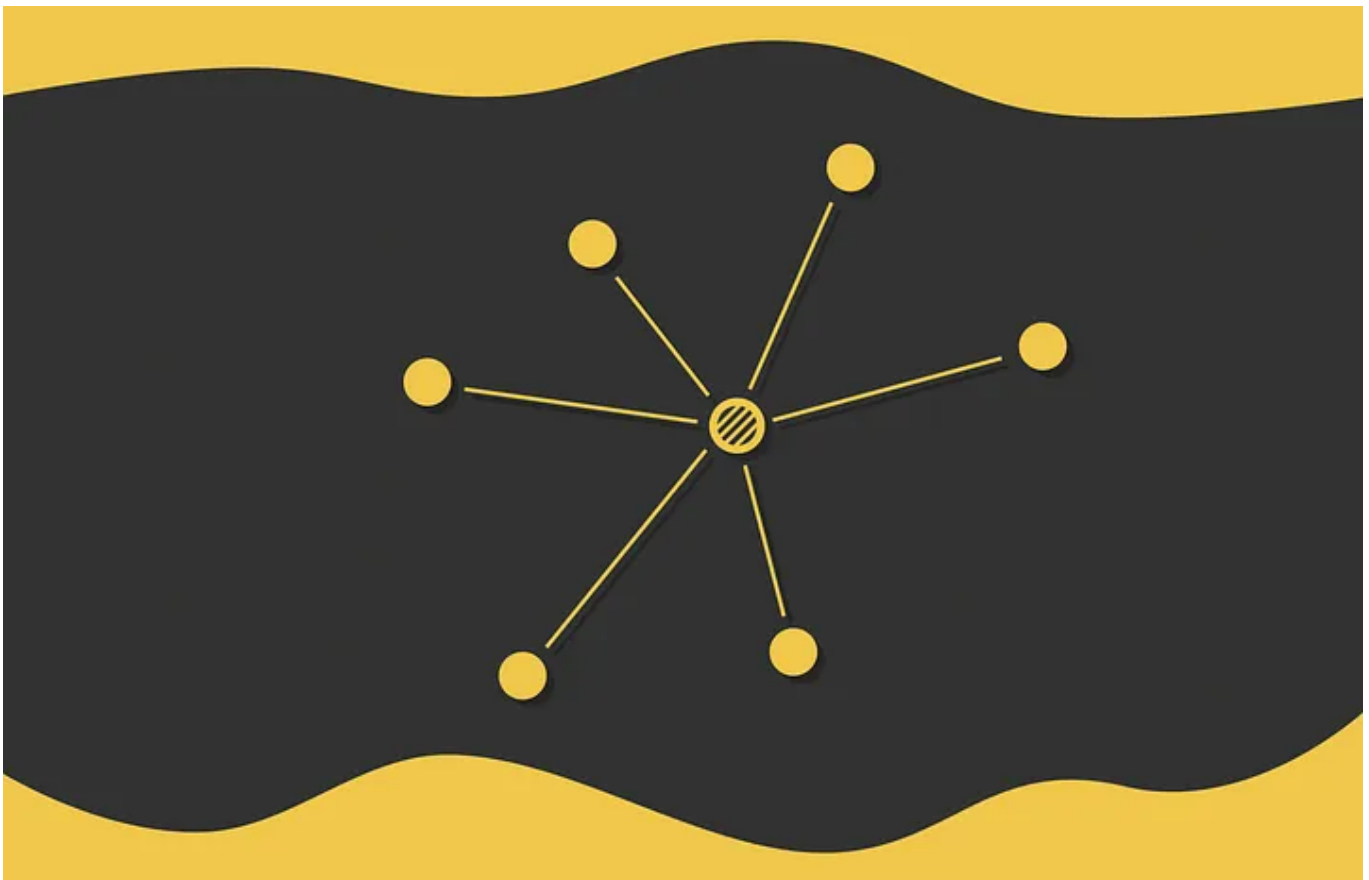


Vyacheslav Efimov · Following

Published in Towards Data Science · 9 min read · Apr 28



213



Similarity search is a problem where given a query the goal is to find the most similar documents to it among all the database documents.

Introduction

In data science, similarity search often appears in the NLP domain, search engines or recommender systems where the most relevant documents or items need to be retrieved for a query. Normally, documents or items are represented in the form of texts or images. However, machine learning algorithms cannot directly work with raw texts or images, which is why documents and items are usually preprocessed and stored as **vectors** of numbers.

Sometimes each component of a vector can store a semantic meaning. In this case, these representations are also called **embeddings**. Such embeddings can have hundreds of dimensions and their quantity can reach up to millions! Because of such huge numbers, any information retrieval system must be capable of rapidly detecting relevant documents.

| *In machine learning, a vector is also referred to as an **object** or **point**.*

Index

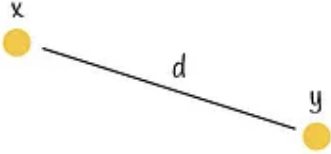
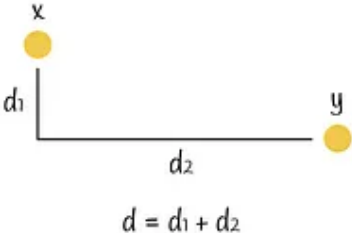
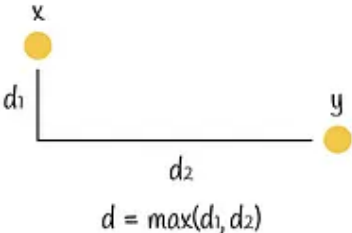
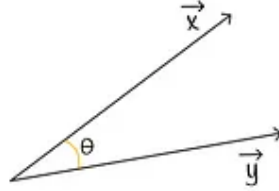
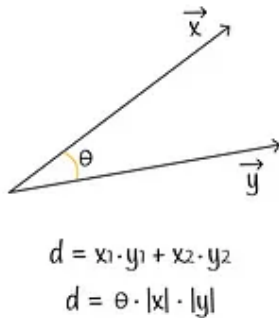
To accelerate the search performance, a special data structure is built on top of dataset embeddings. Such a data structure is called **index**. There has been a lot of research in this field and many types of indexes have been evolved. Before choosing an index to apply for a certain task, it is necessary to understand how it operates under the hood since each index serves a different purpose and comes with its own pros and cons.

In this article, we will have a look at the most naive approach — **kNN**. Based on kNN, we will switch to **inverted file** — an index used for a more scalable search that can accelerate the search procedure several times.

kNN

kNN is the simplest and the most naive algorithm for similarity search.

Consider a dataset of vectors and a new query vector Q . We would like to find the top k dataset vectors which are the most similar to Q . The first aspect to think about is how to measure a similarity (distance) between two vectors. In fact, there are several similarity metrics to do it. Some of them are illustrated in the figure below.

Metric	Formula	Interpretation
Euclidean distance	$d = \sqrt{\sum_i^n (x_i - y_i)^2}$	
Manhattan distance	$d = \sum_i^n x_i - y_i $	
Chebyshev distance	$d = \max_i (x_i - y_i)$	
Cosine distance	$\theta = \frac{x \cdot y}{ x \cdot y }$	
Inner product	$d = \sum_i^n x_i \cdot y_i$	

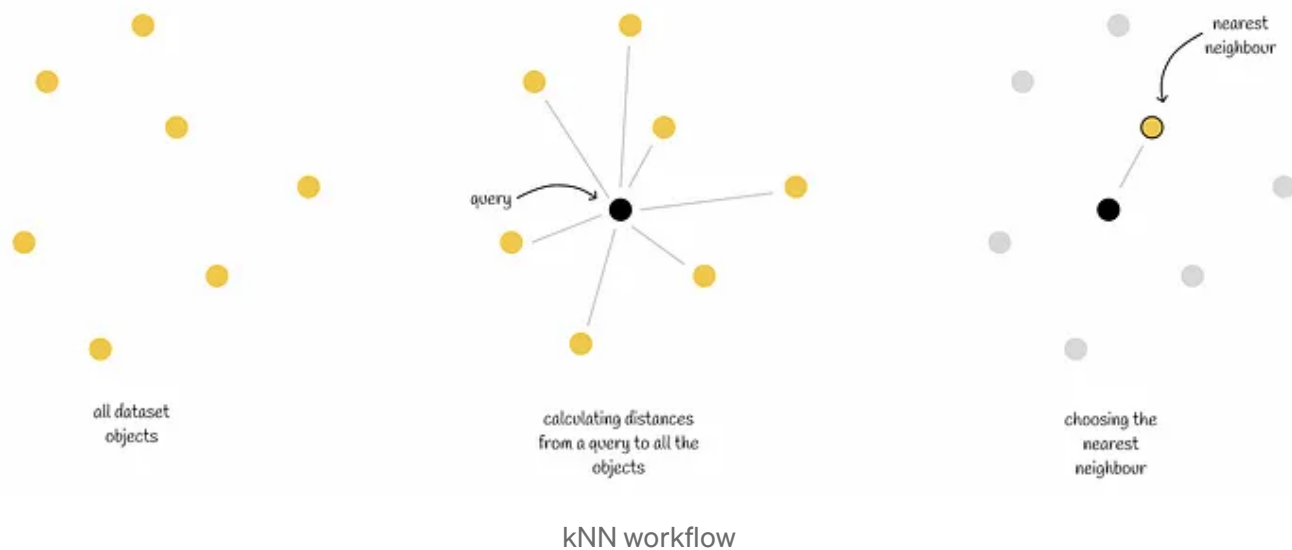
Similarity metrics

Training

kNN is one of the few algorithms in machine learning that does not require a training phase. After choosing an appropriate metric, we can directly make predictions.

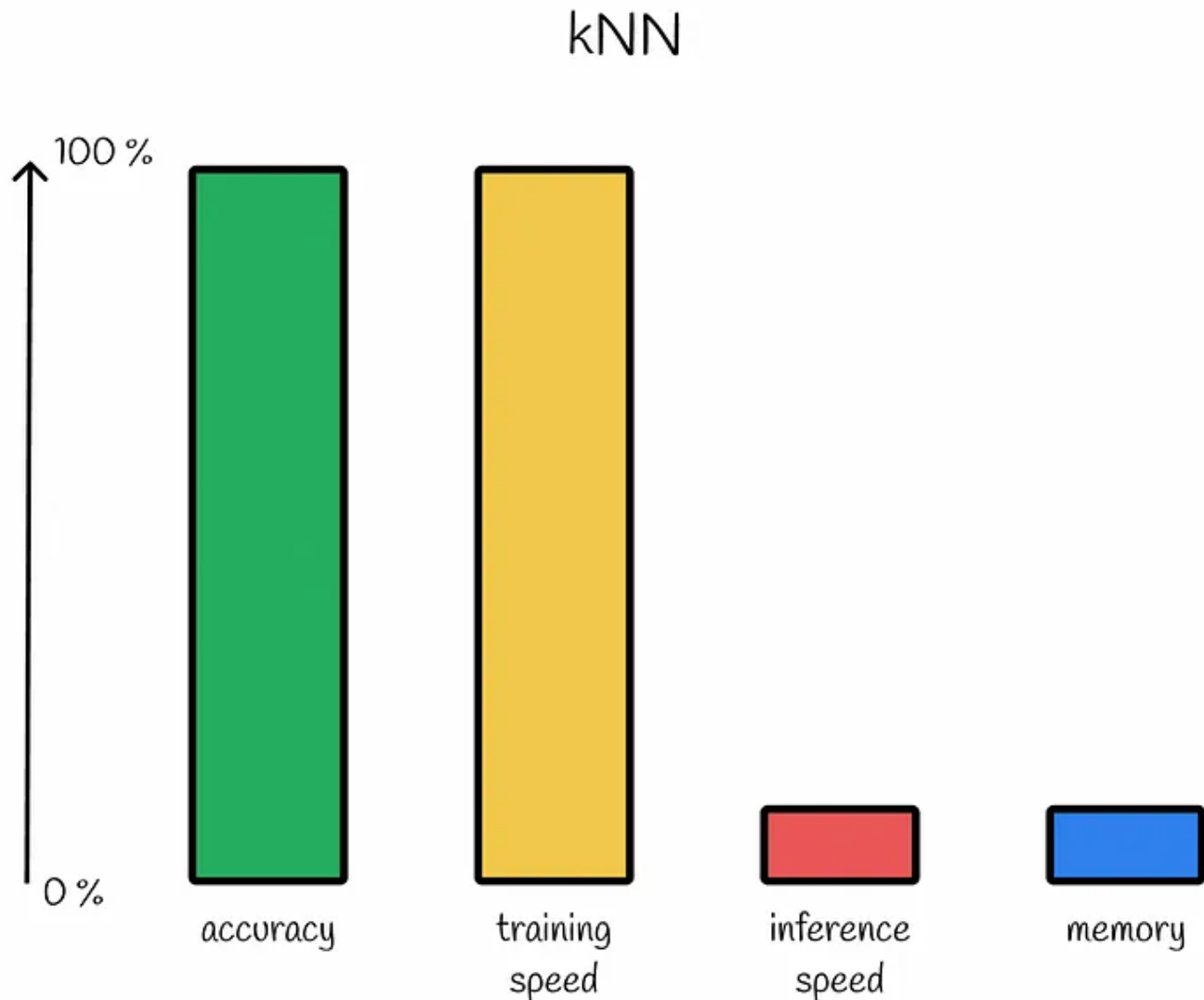
Inference

For a new object, the algorithm exhaustively calculates distances to all the other objects. After that, it finds k objects with the smallest distances and returns them as a response.



Obviously, by checking distances to all the dataset vectors, kNN guarantees 100% accurate results. However, such brute force approach is very inefficient in terms of time performance. If a dataset consists of n vectors with m dimensions, then for each of n vectors $O(m)$ time is required to calculate the distance to it from a query Q which results in $O(mn)$ total time complexity. As we will see later, there exist more efficient methods.

Moreover, there is no compression mechanism for the original vectors. Imagine a dataset with a billions of objects. It would probably be impossible to store all of them in RAM!



kNN performance. Having 100% accuracy and no training phase results in exhaustive search during the inference and no-memory compression of vectors. Note: this type of diagram shows relative comparison of different algorithms. Depending on a situation and chosen hyperparameters, the performance may vary.

Application

kNN has a limited application scope and should be used only in one of the following scenarios:

- The dataset size or embedding dimensionality is relatively small. This aspect makes sure that the algorithm still performs fast.
- The required accuracy of the algorithm must be 100%. In terms of accuracy, there is no other nearest neighbours algorithm which can outperform kNN.

Detection of a person based on his or her fingerprints is an example of a problem where 100% accuracy is required. If the person has committed a crime and left his fingerprints, it is vital to retrieve only the correct results. Otherwise, if the system is not 100% reliable, then another person can be found guilty of a crime which is a very critical error.

Basically, there are two main ways to improve kNN (which we will discuss later):

- Reduce the search scope.
- Reduce the vectors' dimensionality.

When using one of these two approaches, we will not be performing an exhaustive search again. Such algorithms are called **approximate nearest neighbours** (ANN) because they do not guarantee 100% accurate results.

Inverted File Index

“Inverted index (also referred to as a postings list, postings file, or inverted file) is a database index storing a mapping from content, such as words or numbers, to its locations in a table, or in a document or a set of documents” — Wikipedia

When performing a query, the hash function of the query is computed and mapped values from the hash table are taken. Each of these mapped values contains its own set of potential candidates which then are fully checked on

a condition to be the nearest neighbour for the query. By doing so, the search scope of all database vectors is reduced.

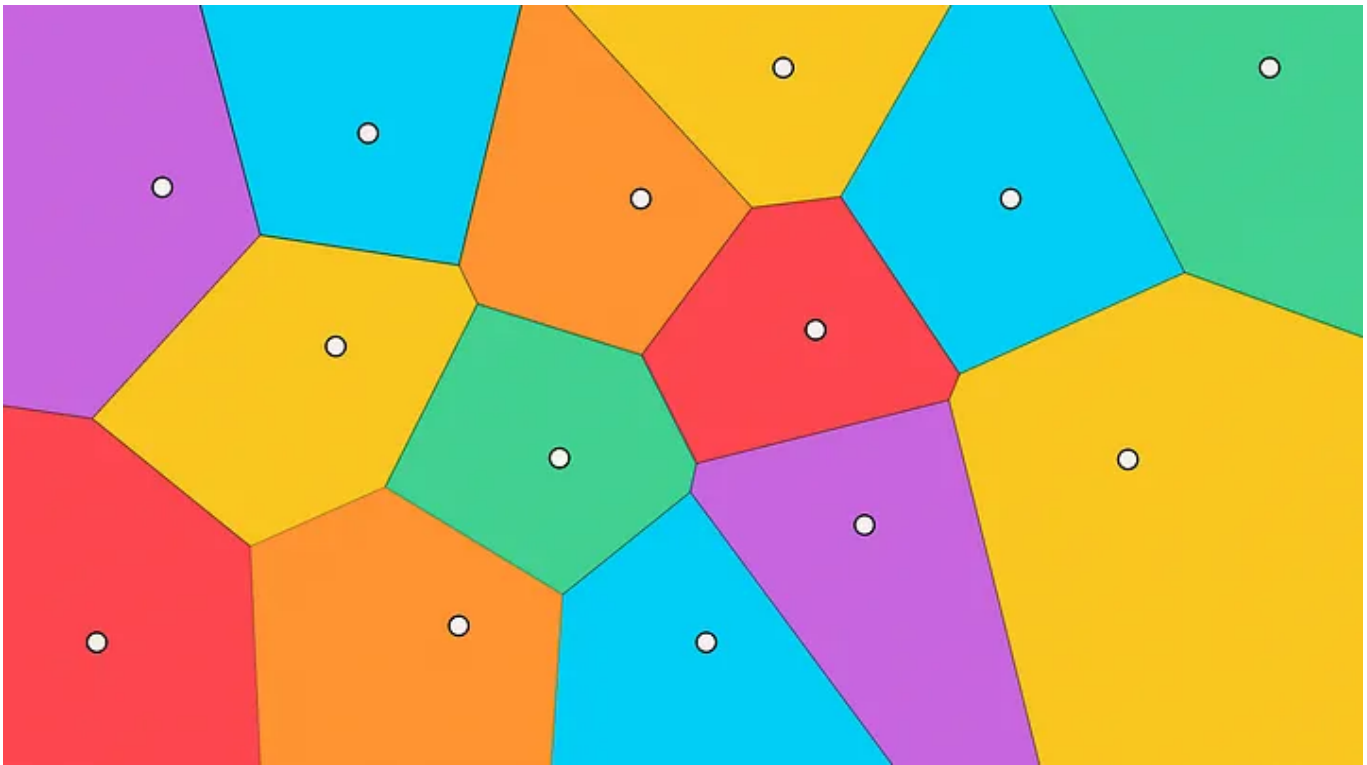


There are different implementations of this index depending on how hash functions are computed. The implementation we are going to look at is the one that uses **Voronoi diagrams** (or **Dirichlet tessellation**).

Training

The idea of the algorithm is to create several non-intersecting regions to which each dataset point will belong. Each region has its own centroid which points to the center of that region.

Sometimes Voronoi regions are referred to as cells or partitions.

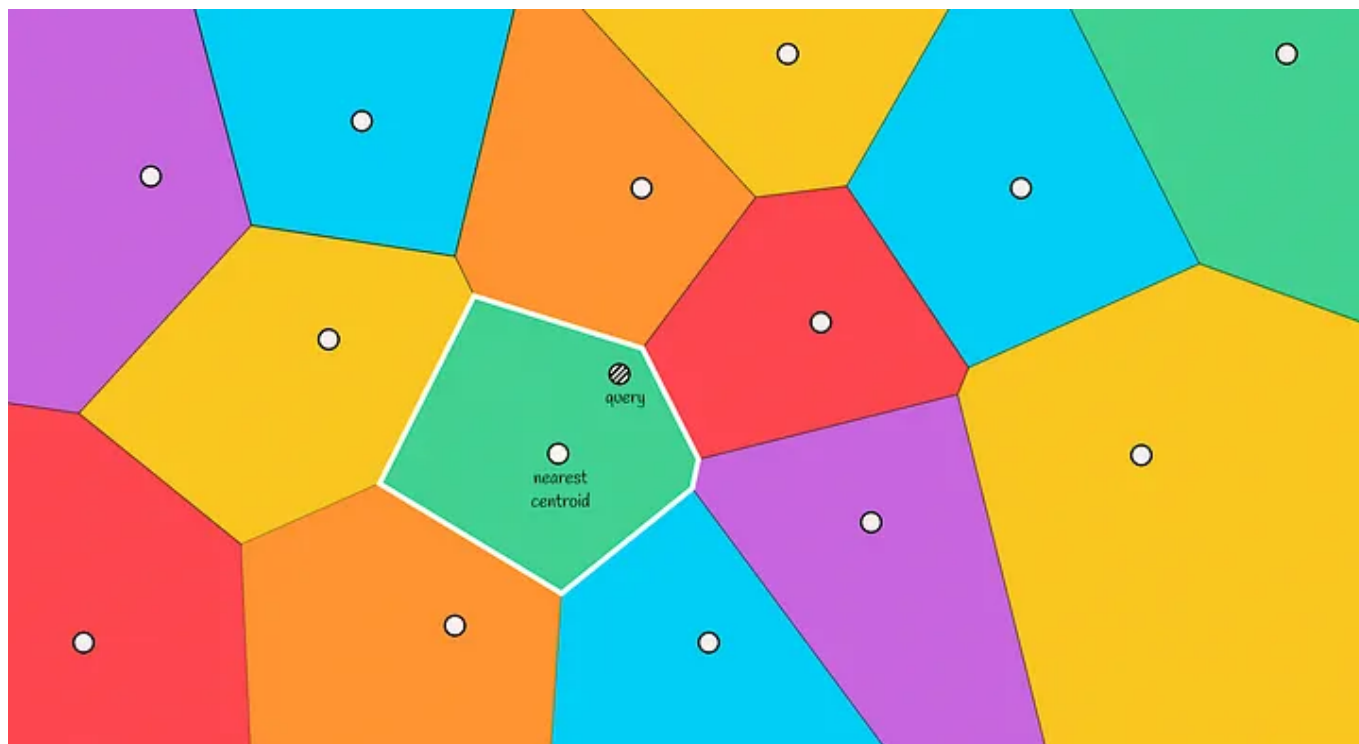


Example of a Voronoi diagram. White points are centers of respective partitions which contain a set of candidates.

The main property of Voronoi diagrams is that the distance from a centroid to any point of its region is less than the distance from that point to another centroid.

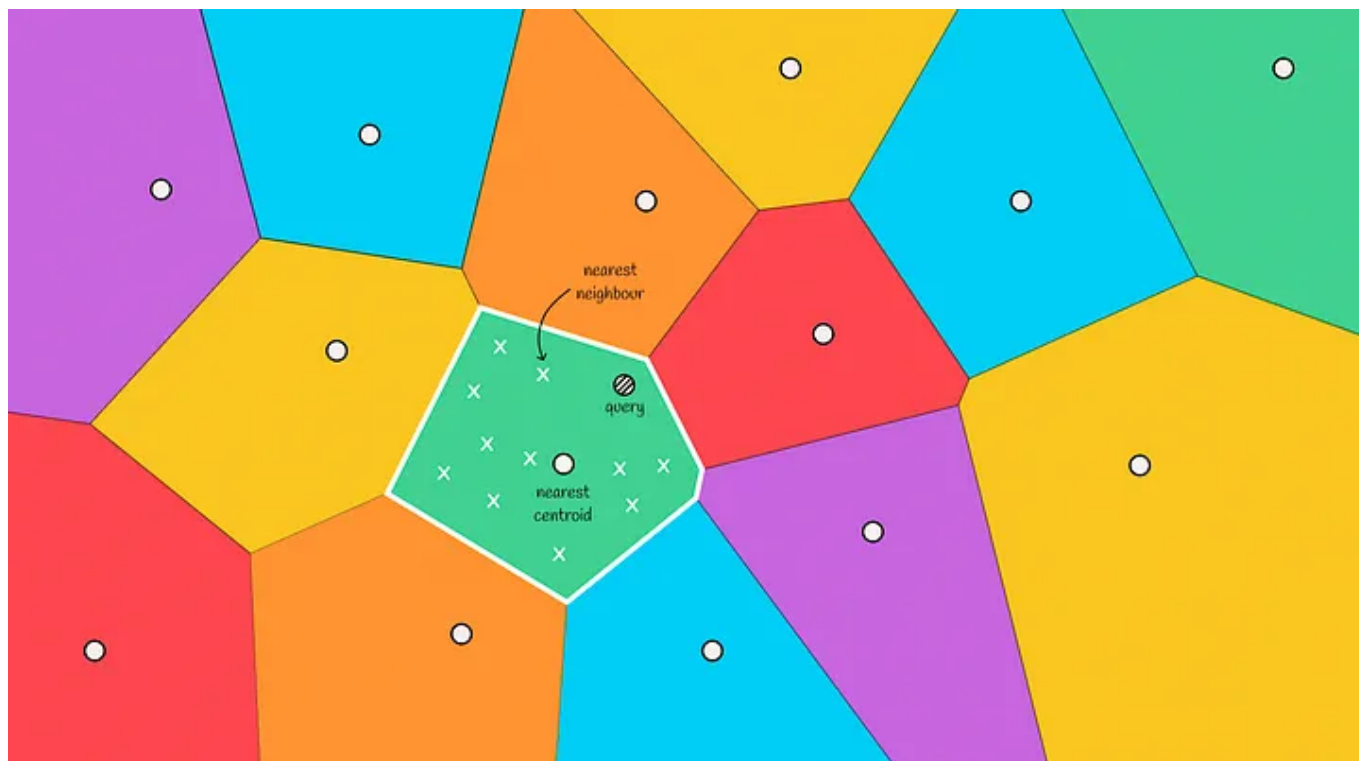
Inference

When given a new object, distances to all the centroids of Voronoi partitions are calculated. Then the centroid with the lowest distance is chosen and vectors contained in this partition are then taken as candidates.



By a given query we search for the nearest centroid (located in the green zone)

Ultimately, by computing the distances to the candidates and choosing the top k nearest of them, the final answer is returned.



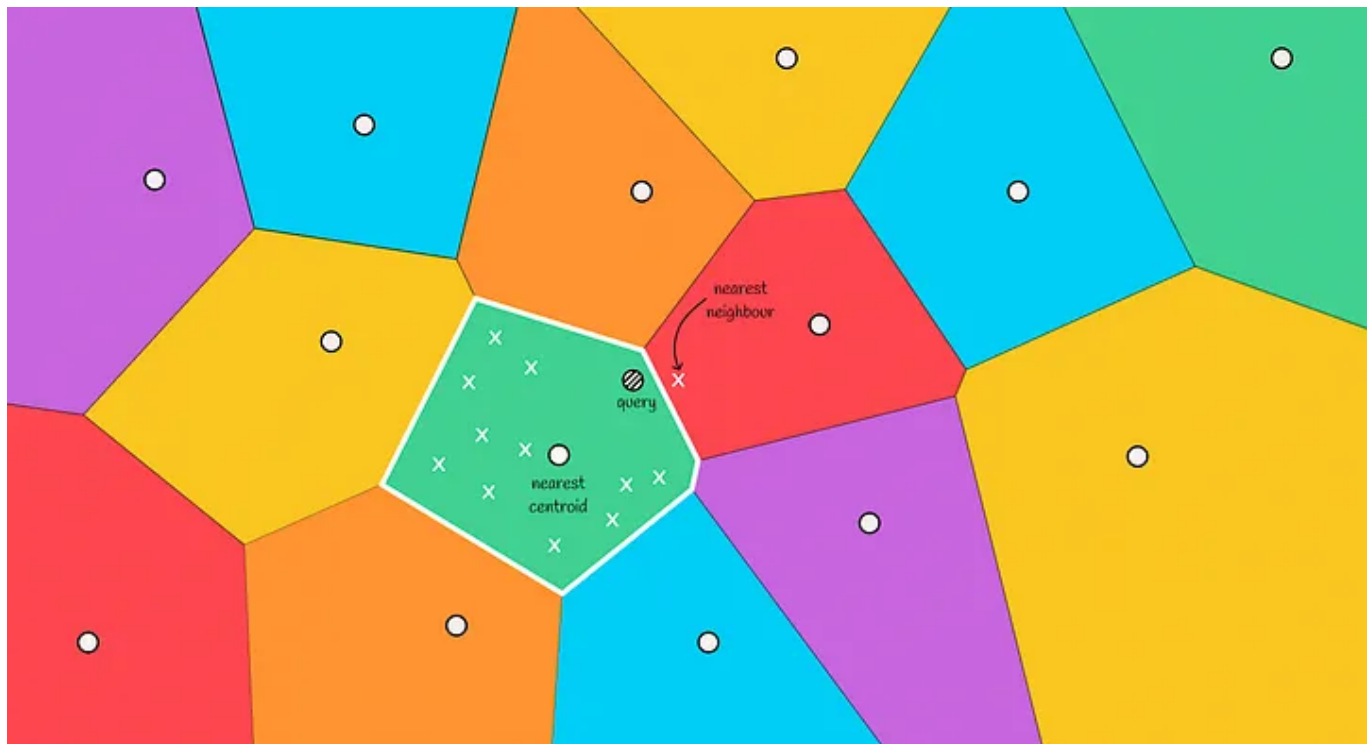
Finding the nearest neighbour in the selected region

As you can see, this approach is much faster than the previous one as we do not have to look through all the dataset vectors.

Edge problem

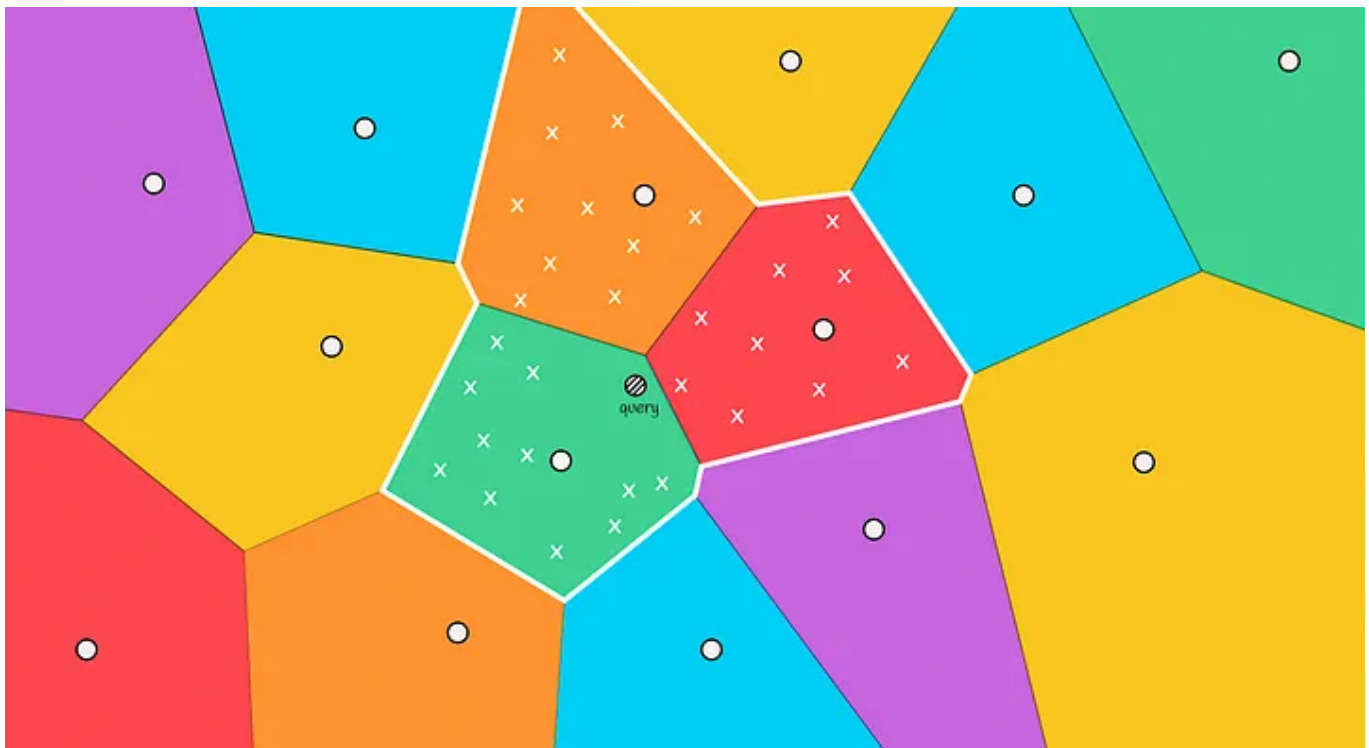
With the increase in search speed, inverted file comes with a downside: it does not guarantee that the found object will always be the nearest.

In the figure below we can see such a scenario: the actual nearest neighbour is located in the red region but we are selecting candidates only from the green zone. Such a situation is called the **edge problem**.



Edge problem

This case typically occurs when the queried object is located near the border with another region. To reduce the number of errors in such cases, we can increase the search scope and choose several regions to search for candidates based on the top m closest centroids to the object.



Searching for nearest neighbours within several regions ($m = 3$)

The more regions are explored, the more accurate results are and the more time it takes to compute them.

Application

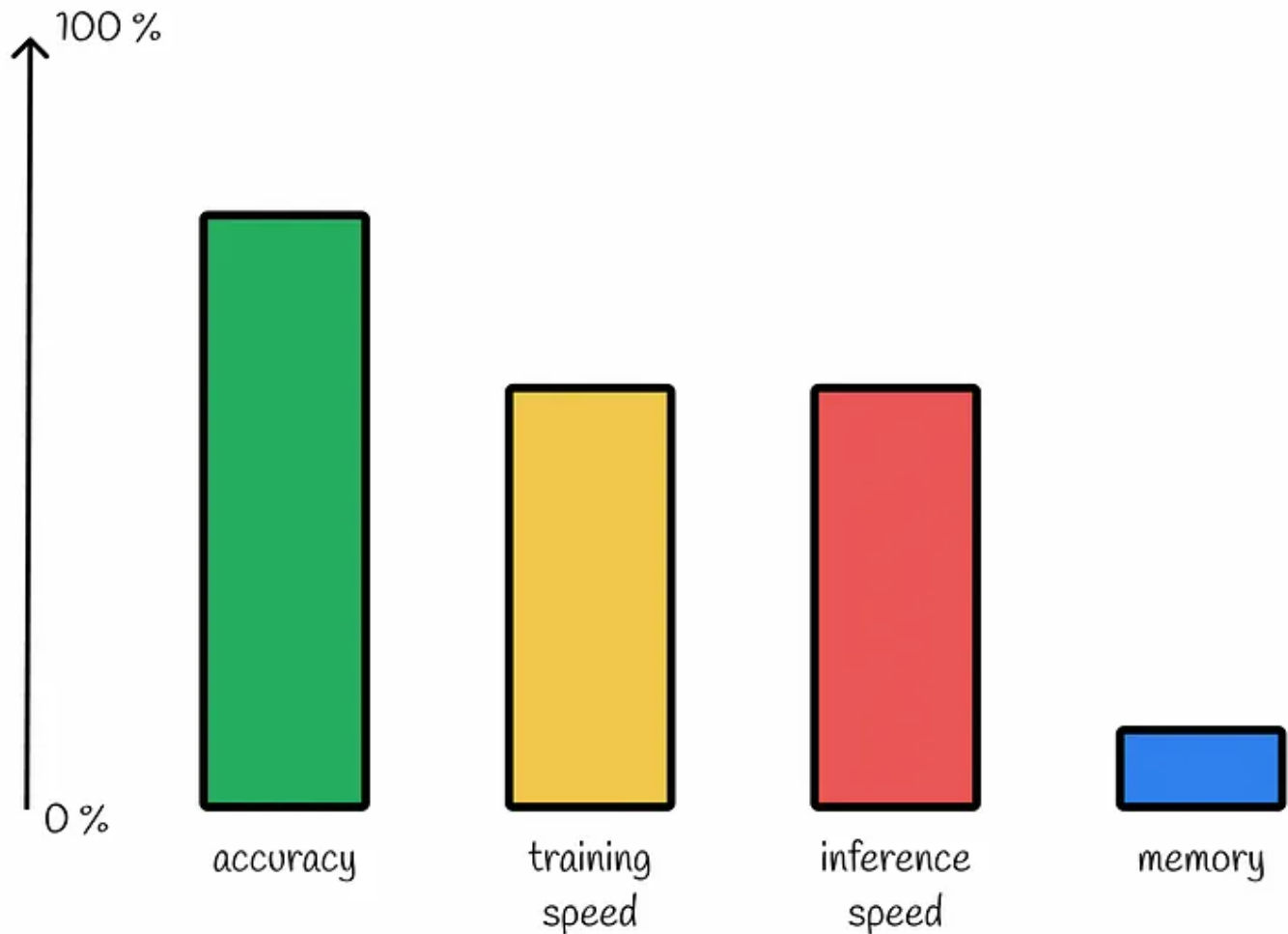
Despite the edge problem, inverted file shows decent results in practice. It is perfect to utilize in cases where we want to trade-off a little decrease in the accuracy for achieving speed growth several times.

One of the use-case examples is a content-based recommender system. Imagine it recommends a movie to a user based on other movies he watched in the past. The database contains a million movies to choose from.

- By using kNN, the system indeed chooses the most relevant movie for a user and recommends it. However, the time required to perform the query is very long.
- Let us assume that with inverted file index, the system recommends the 5-th most relevant movie which is probably the case in real life. The search time is 20 times faster than kNN.

From the user experience, it will be very hard to distinguish between the quality result of these two recommendations: the 1-st and the 5-th most relevant results are both good recommendations from a million of possible movies. The user will probably be happy with any of these recommendations. From the time perspective, inverted file is, obviously, the winner. That is why in this situation it is better to use the latter approach.

Inverted File Index



Inverted file index performance. Here we slightly reduce the accuracy for achieving a higher speed during the inference.

Faiss implementation

***Faiss** (Facebook AI Search Similarity) is a Python library written in C++ used for optimised similarity search. This library presents different types of indexes which are data structures used to efficiently store the data and perform queries.*

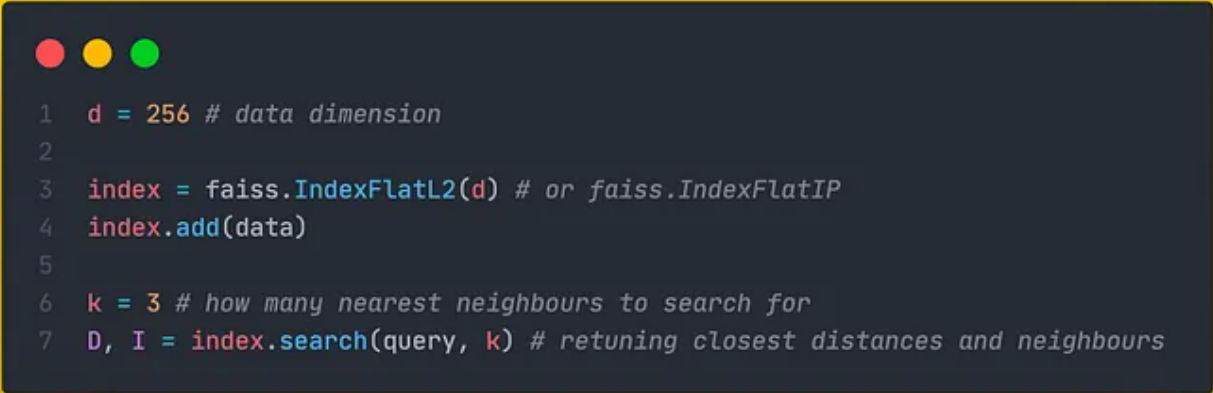
Based on the information from the [Faiss documentation](#), we will see how indexes are created and parametrized.

kNN

Indexes implementing the kNN approach are referred to as **flat** in Faiss because that they do not compress any information. They are the only indexes that guarantee the correct search result. Actually there exist two types of flat indexes in Faiss:

- *IndexFlatL2*. The similarity is calculated as Euclidean distance.
- *IndexFlatIP*. The similarity is calculated as an inner product.

Both of these indexes require a single parameter **d** in their constructors: the data dimension. These indexes do not have any tunable parameters.



```
1 d = 256 # data dimension
2
3 index = faiss.IndexFlatL2(d) # or faiss.IndexFlatIP
4 index.add(data)
5
6 k = 3 # how many nearest neighbours to search for
7 D, I = index.search(query, k) # returning closest distances and neighbours
```

Faiss implementation of IndexFlatL2 and IndexFlatIP

4 bytes are required to store a single component of a vector. Therefore, to store a single vector of dimensionality d , $4 * d$ bytes are required.

$$\text{bytes} = 4 * d$$

Inverted file index

For described inverted file, Faiss implements the class *IndexIVFFlat*. As in the case with kNN, the word “*Flat*” indicates that there is no decompression of original vectors and they are fully stored.

To create this index, we first need to pass a quantizer — an object that will determine how database vectors will be stored and compared.

IndexIVFFlat has 2 important parameters:

- **nlist**: defines a number of regions (Voronoi cells) to create during training.
- **nprobe**: determines how many regions to take for the search of candidates. Changing nprobe parameter does not require retraining.

```
1 d = 256 # data dimension
2 nlist = 50 # number of Voronoi partitions
3
4 quantizer = faiss.IndexFlatL2(d) # how vectors will be compared
5 index = faiss.IndexIVFFlat(quantizer, d, nlist) # creating an index
6 index.train(data) # creating regions
7 index.add(data) # adding data to regions
8
9 k = 3 # how many nearest neighbours to search for
10 index.nprobe = 10 # how many Voronoi partitions to use to search for candidates
11 D, I = index.search(query, k) # returning closest distances and neighbours
```

Faiss implementation of IndexIVFFlat

As in the previous case, we need $4 * d$ bytes to store a single vector. But now we also have to store information about Voronoi regions to which dataset vectors belong to. In Faiss implementation, this information takes 8 bytes per vector. Therefore, the memory required to store a single vector is:

$$\text{bytes} = 4 * d + 8$$

Conclusion

We have gone through two base algorithms in similarity search. Effectively naive kNN should almost be never used for machine learning applications because of its bad scalability except for specific cases. On the other hand, inverted file provides good heuristics for accelerated search whose quality can be improved by tuning its hyperparameters. The search performance can still be enhanced from different perspectives. In the next part of this article series, we will have a look at one of such methods designed to compress dataset vectors.

Similarity Search, Part 2: Product Quantization

Learn a powerful technique to effectively compress large data

towardsdatascience.com

Resources

- [Inverted Index](#) | [Wikipedia](#)
- [Faiss documentation](#)
- [Faiss repository](#)
- [Summary of Faiss indexes](#)
- [Guideline for choosing an index](#)

All images unless otherwise noted are by the author.

Similarity Search

Knn

Inverted Index

Machine Learning

Getting Started

More from the list: "NLP"

Curated by Himanshu Birla



Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

★ · 11 min read · Sep 4, 2021



Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

★ · 6 min read · Sep 3, 2021



Jon Gi... in

The Word2vec

★ · 15 min read

[View list](#)

Written by Vyacheslav Efimov

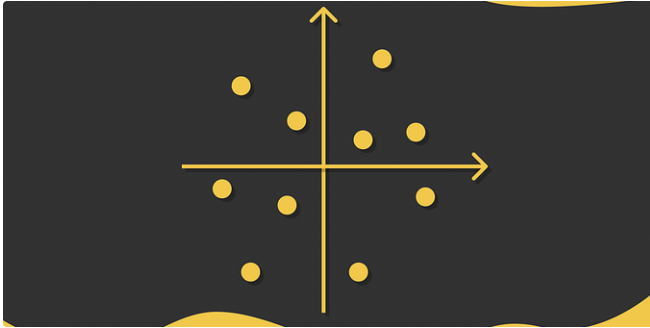
470 Followers · Writer for Towards Data Science

BSc in Software Engineering. Passionate machine learning engineer. Writer at Towards Data Science.

Following



More from Vyacheslav Efimov and Towards Data Science



Vyacheslav Efimov in Towards Data Science

Similarity Search, Part 3: Blending Inverted File Index and Product...

In the first two parts of this series we have discussed two fundamental algorithms in...

8 min read · May 19



128



Antonis Makropoulos in Towards Data Science

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and...

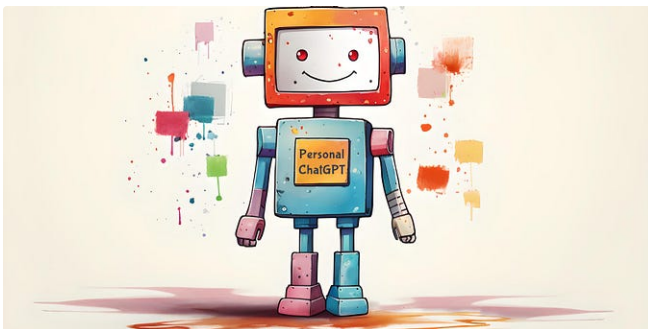
10 min read · Sep 17



549



11

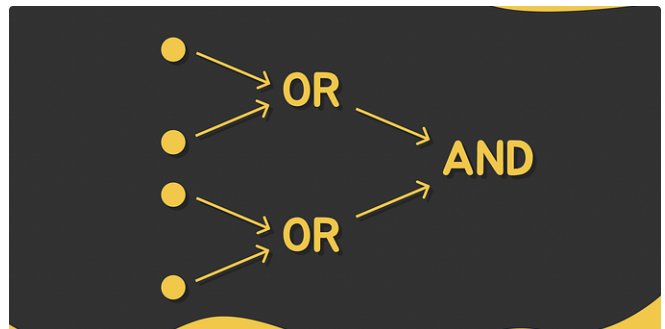


Robert A. Gonsalves in Towards Data Science

Your Own Personal ChatGPT

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...

★ · 15 min read · Sep 8



Vyacheslav Efimov in Towards Data Science

Similarity Search, Part 7: LSH Compositions

Dive into combinations of LSH functions to guarantee a more reliable search

11 min read · Jul 24

595

7

43

See all from Vyacheslav Efimov

See all from Towards Data Science

Recommended from Medium

Core principle	Keyword based Search	Semantic Search
Applicable for	Fully /Semi Structured data	Unstructured data
Preferable with	Advance Search Filters	Natural language Interface
Best suited to capture	Explicit context in the query	Implicit context in the query
Frameworks	ElasticSearch, Solr, Pinecone etc	FAISS, Vespa, Solr, Pinecone etc
Hybrid Search		
Core principle	Semantic Search	
Applicable for	Fully / Semi Structured as well as Unstructured data	
Preferable with	Natural language interface alongside Advanced search filters	



Maithri Vm

Hybrid Search—Amalgamation of Sparse and Dense vector...

— Uniting meaning of data with metadata to leverage deeper context

13 min read · May 14

16

71

1

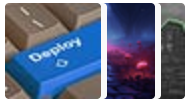
Alyx

Semantic Search with FAISS

HuggingFace get_neareast_example and Cosine Similarity Search

9 min read · Jul 15

Lists



Predictive Modeling w/ Python

20 stories · 452 saves



Practical Guides to Machine Learning

10 stories · 519 saves



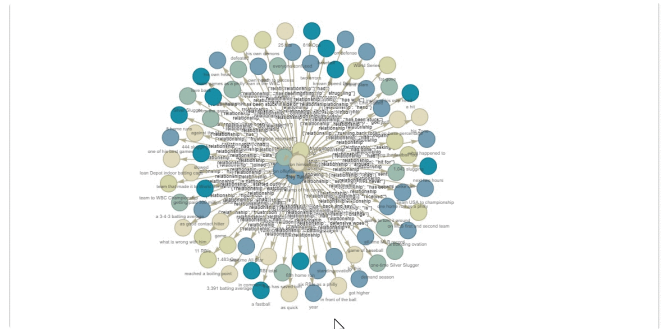
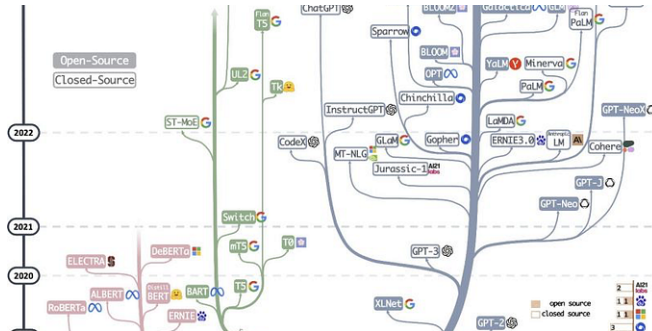
Natural Language Processing

669 stories · 283 saves



The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves

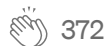


Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14



372



Wenqi Glantz in Better Programming

7 Query Strategies for Navigating Knowledge Graphs With...

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies



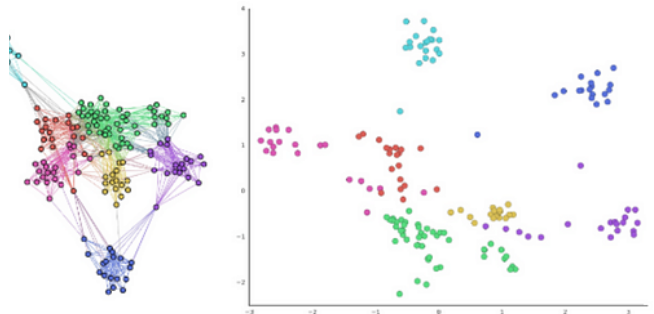
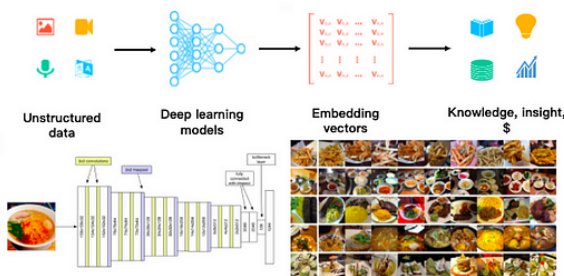
17 min read · 4 days ago



501



4



Jayita Bhattacharyya in GoPenAI



Artem Shlezinger

Primer on Vector Databases and Retrieval-Augmented Generation...

Vector Databases Generation (RAG)
Langchain Pinecone HuggingFace Large...

9 min read · Aug 16



228



1



9



1



See more recommendations