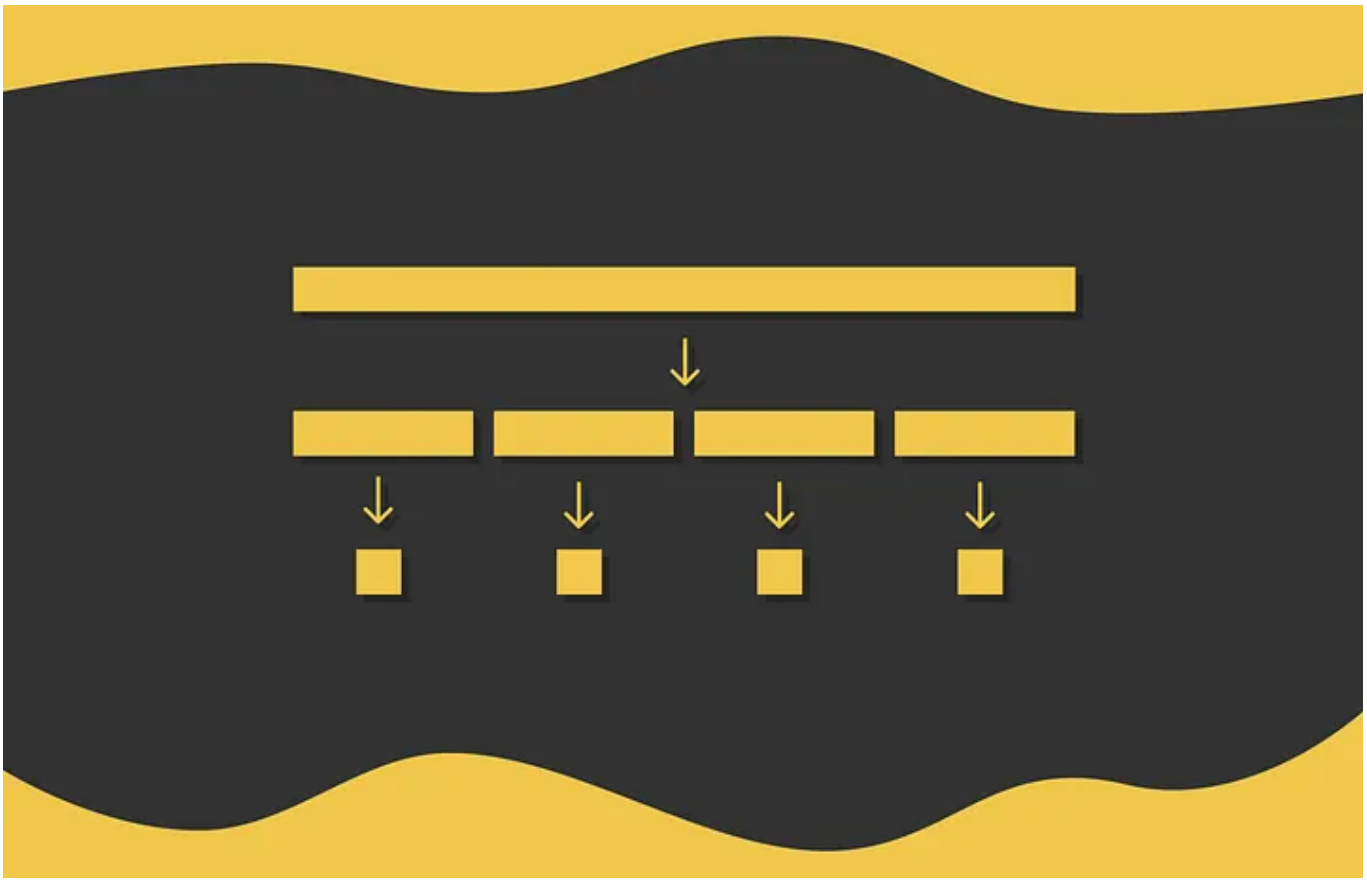
Similarity Search, Part 2: Product Quantization

Learn a powerful technique to effectively compress large data

Vyacheslav Efimov · Following
Published in Towards Data Science · 9 min read · May 10



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. There exists a large variety of different ways to improve search performance in massive volumes of data.

In the first part of this article series, we looked at kNN and inverted file index structure for performing similarity search. As we learned, kNN is the most straightforward approach while inverted file index acts on top of it suggesting a trade-off between speed acceleration and accuracy. Nevertheless, both methods do not use data compression techniques which might lead to memory issues, especially in cases of large datasets and limited RAM. In this article, we will try to address this issue by looking at another method called **Product Quantization**.

Definition

Product quantization is the process where each dataset vector is converted into a short memory-efficient representation (called **PQ code**). Instead of fully keeping all the vectors, their short representations are stored. At the same time, product quantization is a lossy-compression method which

results in lower prediction accuracy but in practice, this algorithm works very well.

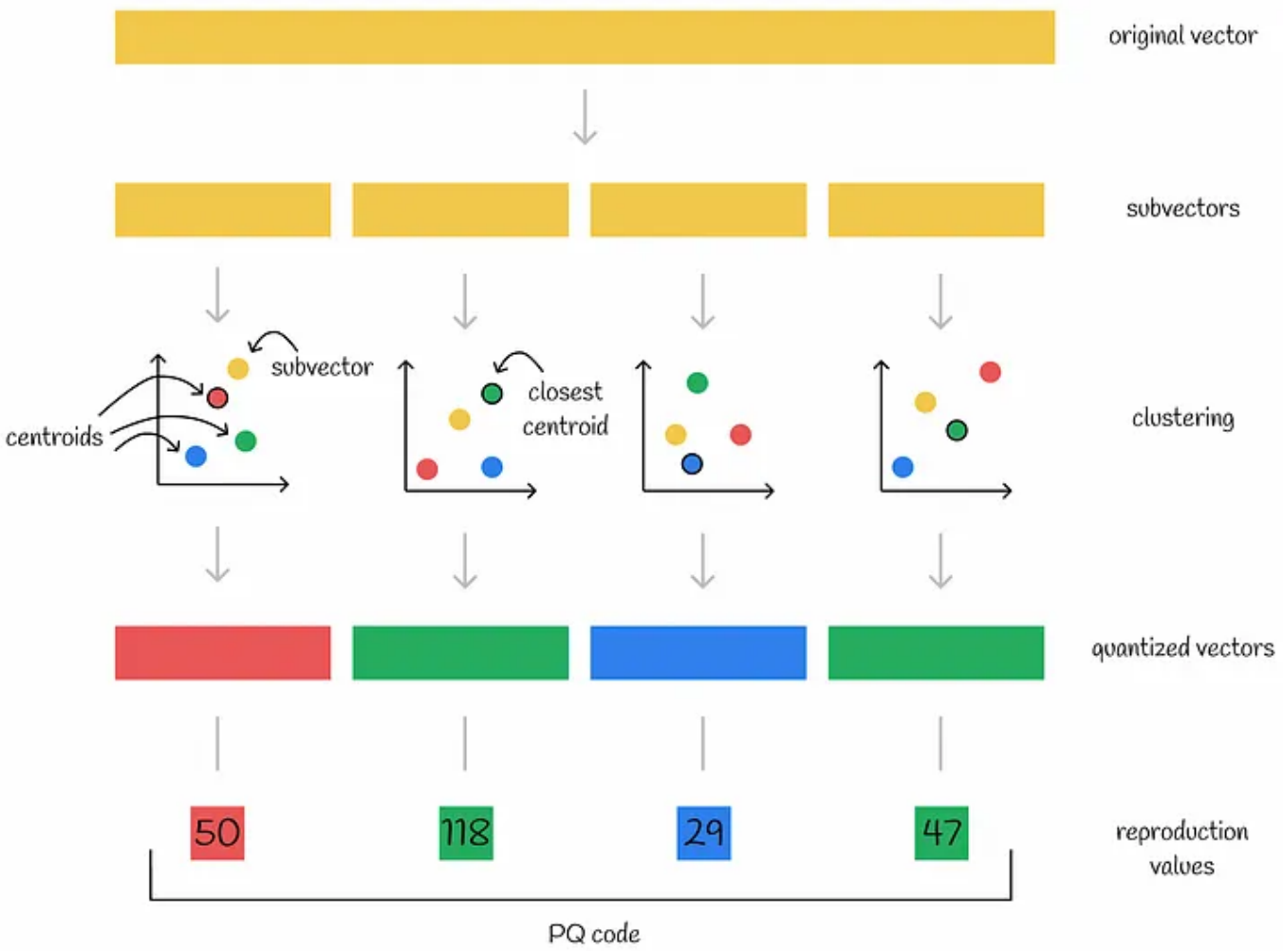
> *In general, quantization is the process of mapping infinite values to discrete ones.*

Training

Firstly, the algorithm divides each vector into several equal parts — **subvectors**. Each of the respective parts of all dataset vectors form independent **subspaces** and is processed separately. Then a clustering algorithm is executed for each subspace of vectors. By doing so, several centroids in each subspace are created. Each subvector is encoded with the ID of the centroid that it belongs to. Additionally, the coordinates of all centroids are stored for later use.

> *Subspace centroids are also called **quantized vectors**.*
>
> *In product quantization, a cluster ID is often referred to as a **reproduction value**.*

Note. In the figures below a rectangle represents a vector containing several values while a square indicates a single number.



Encoding using quantization

As a result, if an original vector is divided into n parts, then it can be encoded by n numbers — IDs of respective centroids for each of its subvectors. Typically, the number of created centroids k is usually chosen as a power of 2 for more efficient memory usage. This way, the memory required to store an encoded vector is $n * log(k)$ bits.

> *The collection of all centroids inside a subspace is called a* ***codebook****. Running n clustering algorithms for all subspaces produces n separate codebooks.*

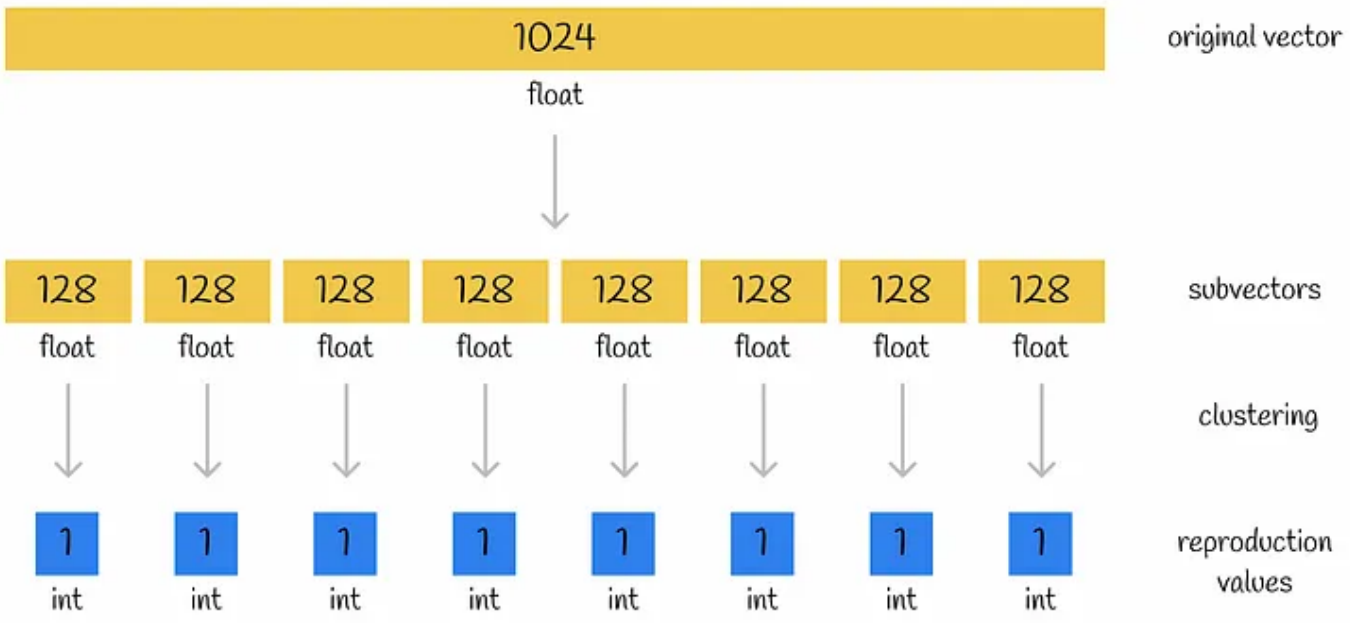
Compression example

Imagine an original vector of size 1024 which stores floats (32 bits) was divided into $n = 8$ subvectors where each subvector is encoded by one of $k =$

256 clusters. Therefore, encoding the ID of a single cluster would require *log(256) = 8* bits. Let us compare the memory sizes for the vector representation in both cases:

- Original vector: 1024 * 32 bits = 4096 bytes.
- Encoded vector: 8 * 8 bits = 8 bytes.

The final compression is 512 times! This is the real power of product quantization.



Quantization example. Numbers in vectors show how many numbers it stores.

Here are some important notes:

- The algorithm can be trained on one subset of vectors (e.g., to create clusters) and be used for another one: once the algorithm is trained, another dataset of vectors is passed where new vectors are encoded by using already constructed centroids for each subspace.

- Typically, k-means is chosen as a clustering algorithm. One of its advantages is that the number of clusters k is a hyperparameter that can be manually defined, according to memory usage requirements.

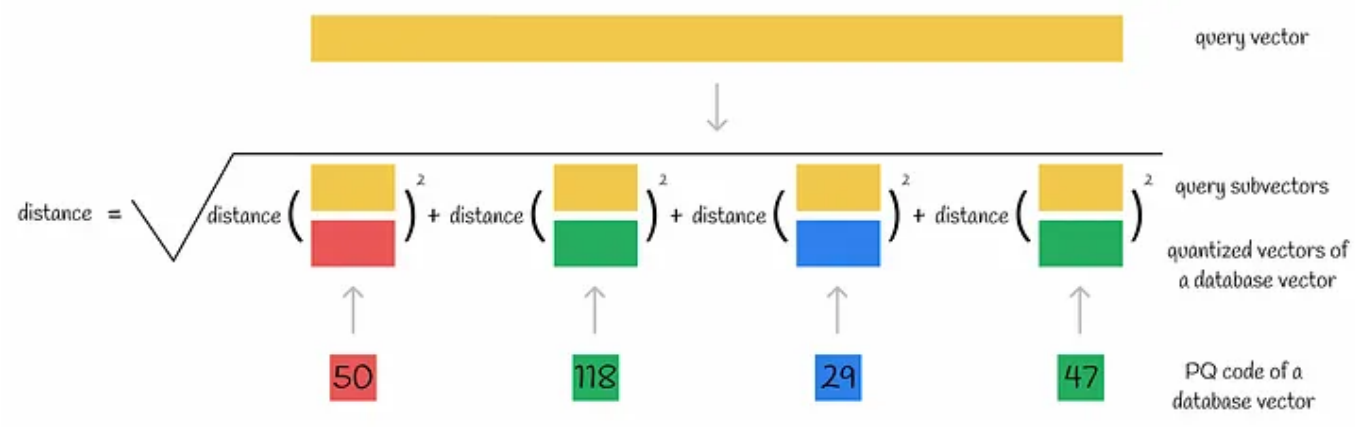
Inference

To get a better understanding, let us first have a look at several naive approaches and find out their downsides. This will also help us realize why they should not be normally used.

Naive approaches

The first naive approach consists of decompressing all vectors by concatenating the corresponding centroids of each vector. After that, the *L2* distance (or another metric) can be calculated from a query vector to all the dataset vectors. Obviously, this method works but it is very time-consuming because the brute-force search is performed and the distance calculation is performed on high-dimensional decompressed vectors.

Another possible way is to split a query vector into subvectors and compute a sum of distances from each query subvector to respective quantized vectors of a database vector, based on its PQ code. As a consequence, the brute-search technique is used again and the distance calculation here still requires a linear time of the original vectors' dimensionality, as in the previous case.

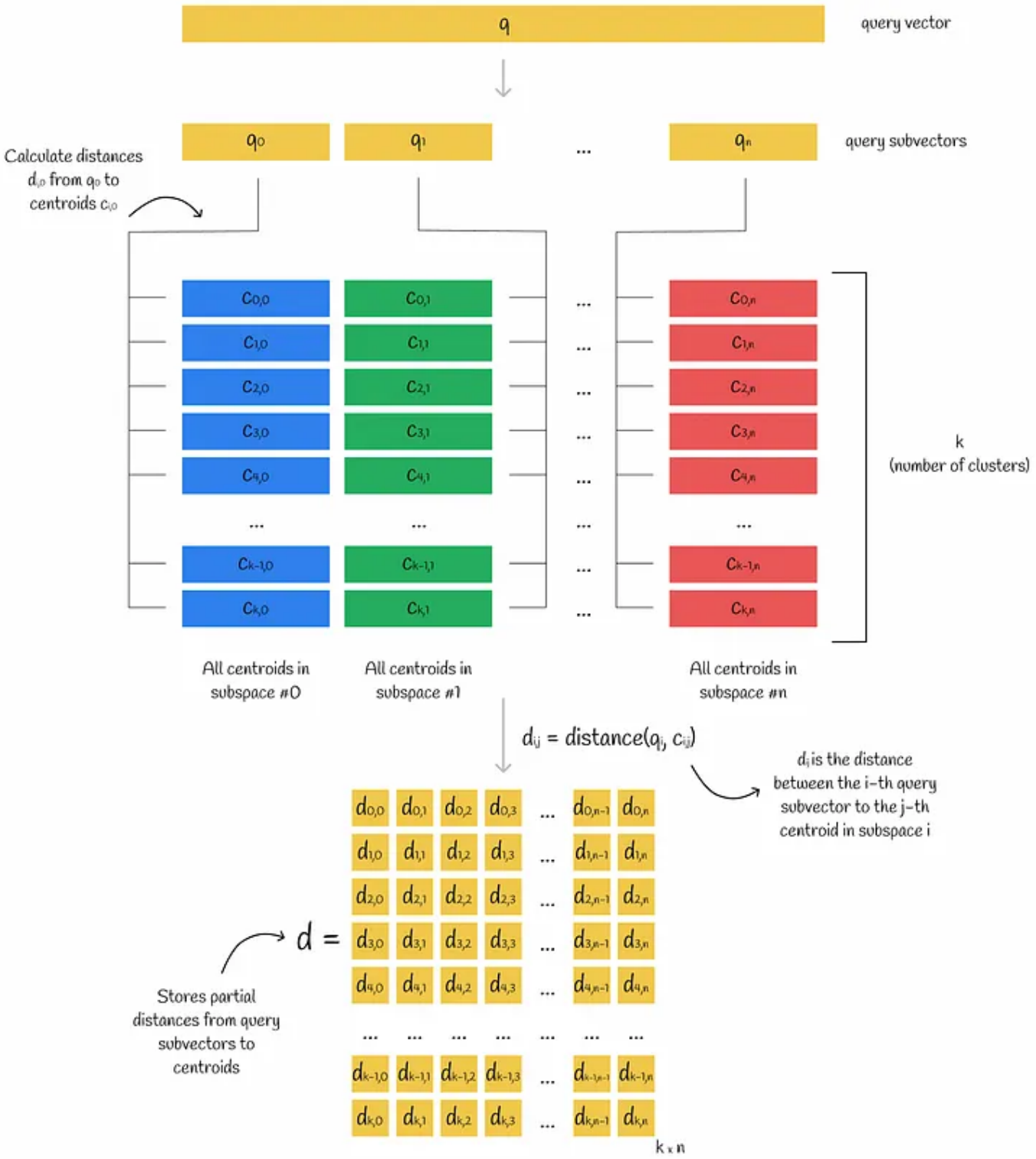


Calculating approximate distance using naive approach. The example is shown for euclidean distance as a metric.

Another possible method is to encode the query vector into a PQ code. Then this PQ code is directly utilized to calculate distances to all other PQ codes. The dataset vector with the corresponding PQ code which has the shortest distance is then considered as the nearest neighbour to the query. This approach is faster than the previous two because the distance is always computed between low-dimensional PQ codes. However, PQ codes are composed by cluster IDs which do not have a lot of semantic meaning and can be considered as a categorical variable explicitly used as a real variable. Clearly, this is a bad practice and this method can lead to poor prediction quality.

Optimized approach

A query vector is divided into subvectors. For each of its subvectors, distances to all the centroids of the corresponding subspace are computed. Ultimately, this information is stored in table d.

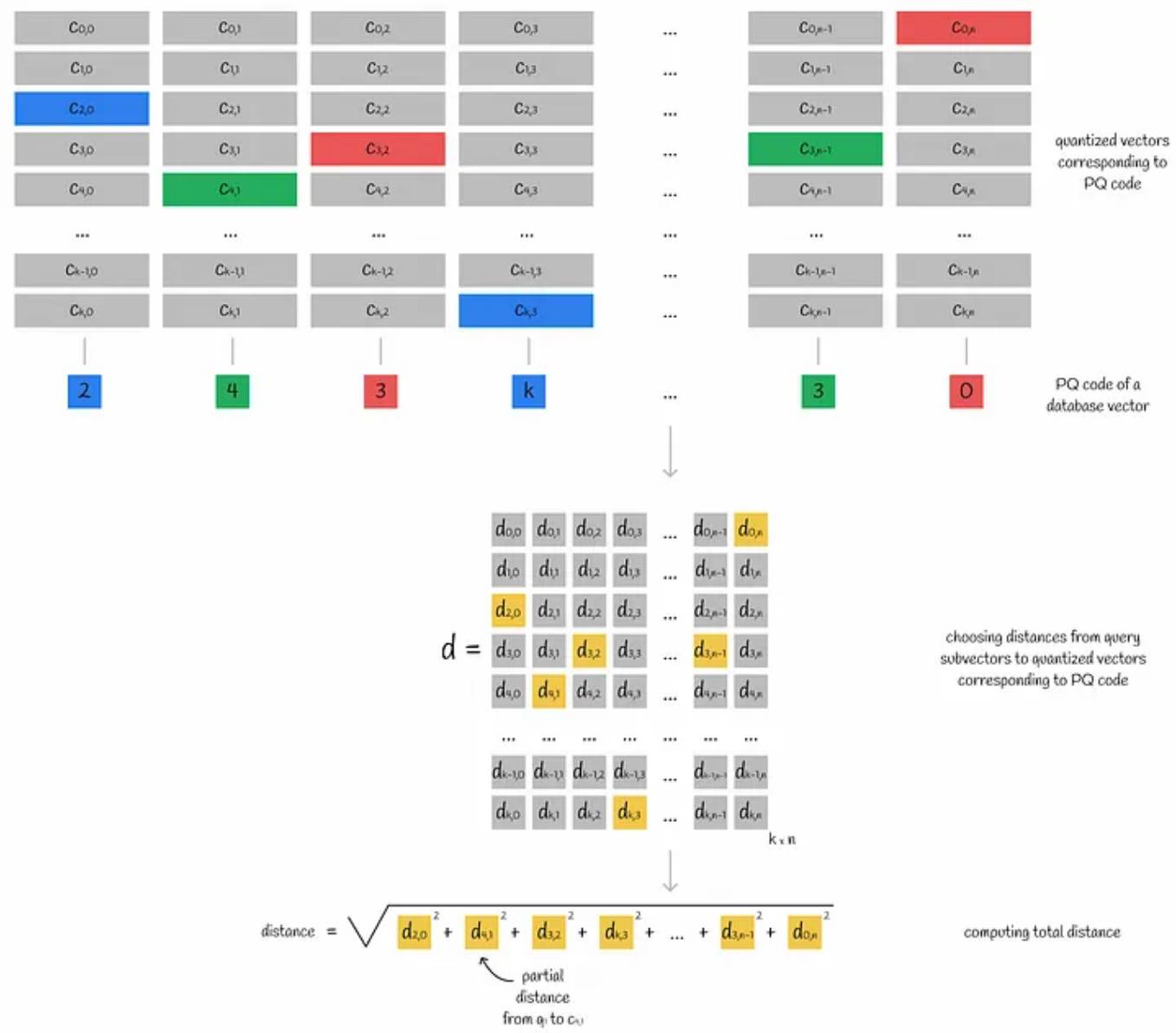


Obtaining a table d storing partial query subvector-to-centroid distances

> *Calculated subvector-to-centroid distances are often referred to as* ***partial distances.***

By using this subvector-to-centroid distance table *d*, the approximate distance from the query to any database vector can be easily obtained by its PQ codes:

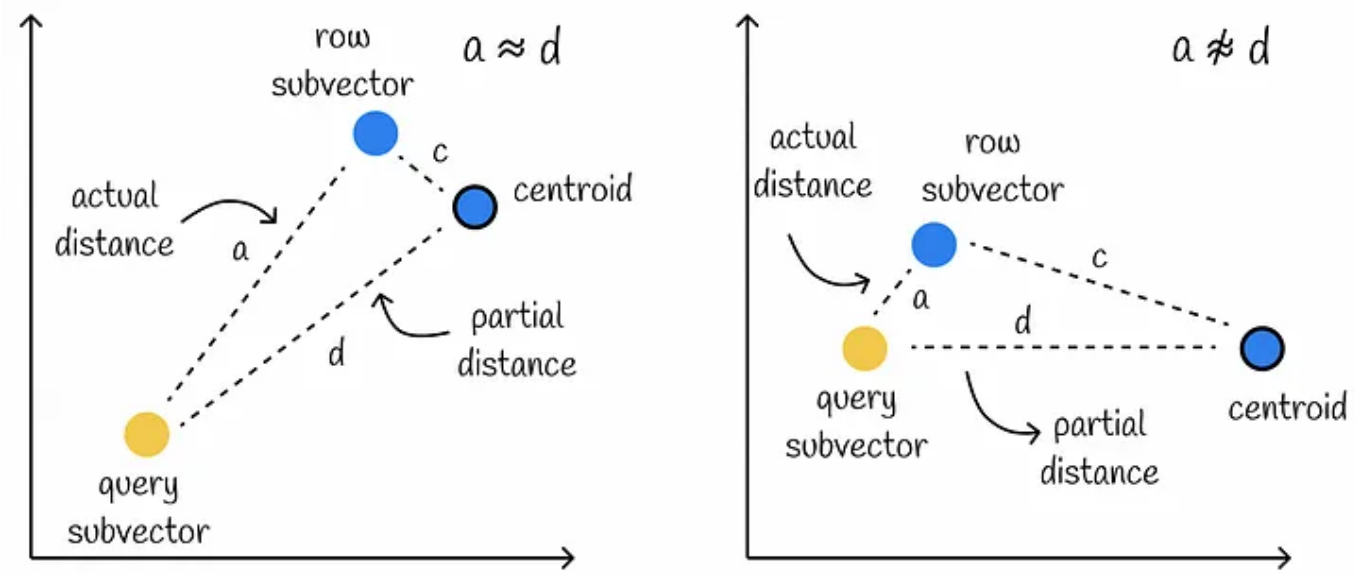
1. For each of subvectors of a database vector, the closest centroid *j* is found (by using mapping values from PQ codes) and the partial distance *d[i][j]* from that centroid to the query subvector *i* (by using the calculated matrix *d*) is taken.
2. All the partial distances are squared and summed up. By taking the square root of this value, the approximate euclidean distance is obtained. If you want to know how to get approximate results for other metrics as well, navigate to the section below *"Approximation of other distance metrics"*.



Computing distance from a query to database vector by using PQ code and distance table

> *Using this method for calculating approximate distances assumes that partial distances* $\boldsymbol{d}$ *are very close to actual distances* $\boldsymbol{a}$ *between query and database subvectors.*

Nevertheless, this condition may not be satisfied, especially when the distance c between the database subvector and its centroid is large. In such cases, calculations result in lower accuracy.



Example on the left shows a good case of approximation when the actual distance is very close to the partial distance (c is small). On the right side, we can observe a bad scenario because the partial distance is much longer than the actual distance (c is large).

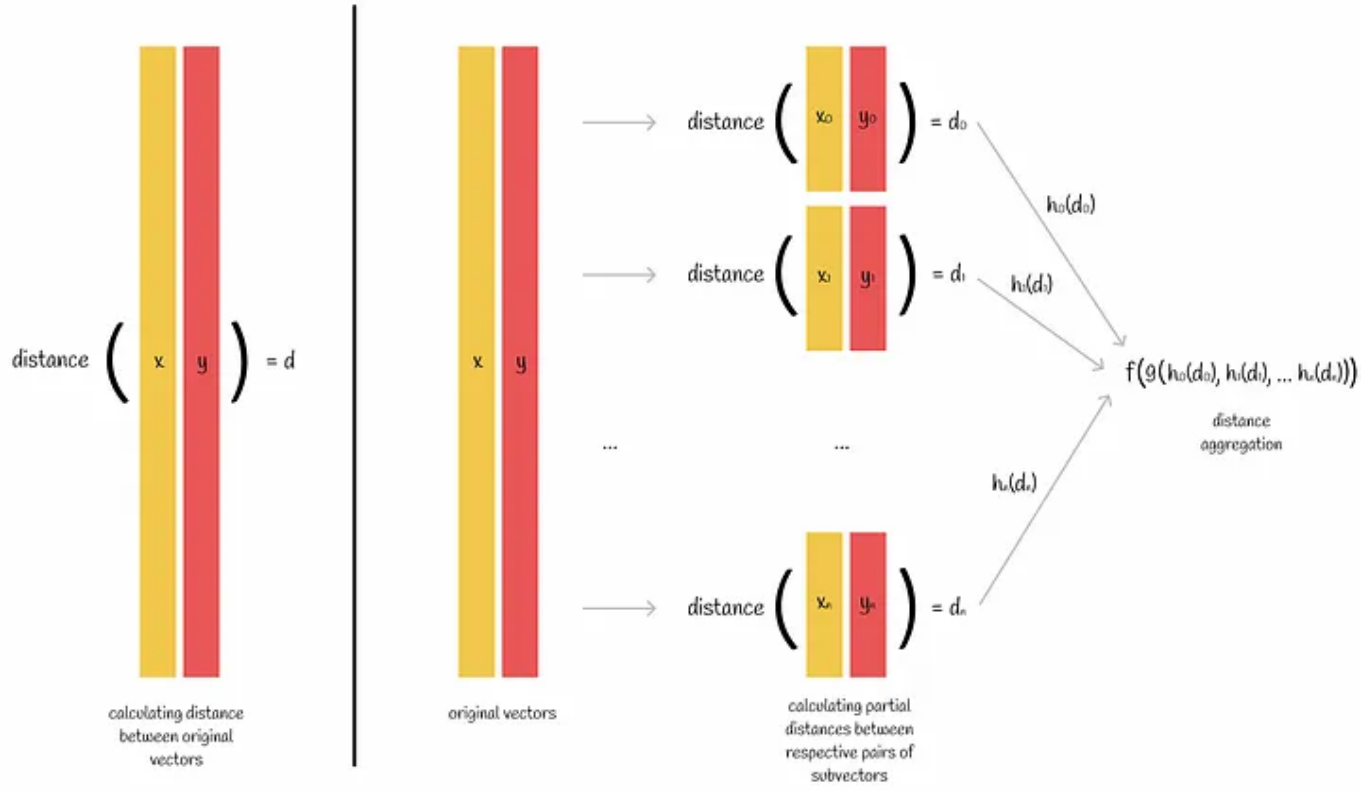
After we have obtained approximate distances for all database rows, we search for vectors with the smallest values. Those vectors will be the nearest neighbours to the query.

Approximation of other distance metrics

So far have looked at how to approximate euclidean distance by using partial distances. Let us generalize the rule for other metrics as well.

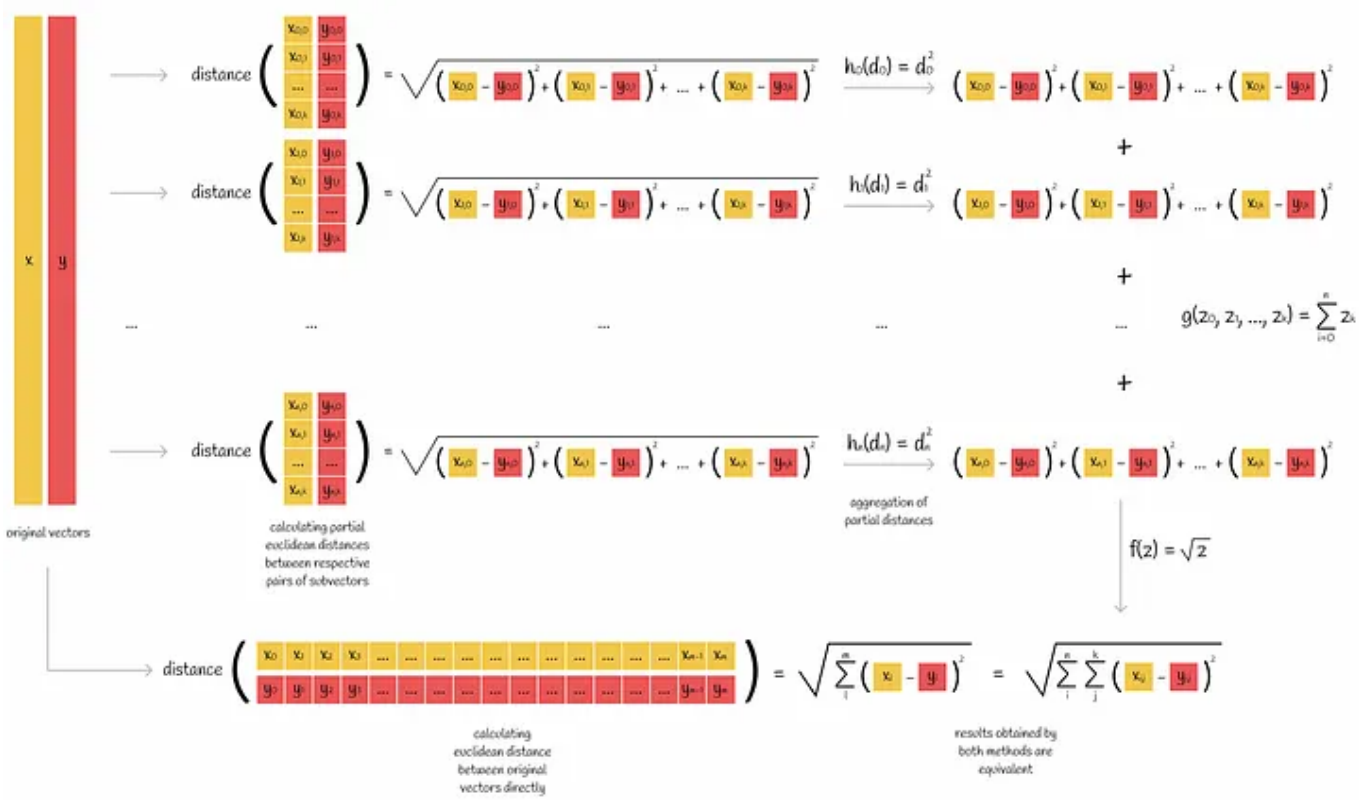
Imagine we would like to calculate a distance metric between a pair of vectors. If we know the metrics' formula, we can directly apply it to get the result. But sometimes we can do it by parts in the following manner:

- Both vectors are divided into n subvectors.
- For each pair of respective subvectors, the distance metric is calculated.
- Calculated n metrics are then combined to produce the actual distance between the original vectors.



The figure shows two ways of calculating a metric. On the left, the metric formula is directly applied to both vectors. On the right, partial distances are calculated for each pair of respective subvectors. Then they are combined by using aggregation functions h, g and f.

Euclidean distance is an example of a metric which can be calculated by parts. Based on the figure above, we can choose the aggregation functions to be $h(z) = z^2$, $g(z_0, z_1, \ldots, z_n) = sum(z_0, z_1, \ldots, z_n)$ and $f(z) = \sqrt{z}$.



Euclidean distance can be calculated by parts

Inner product is another example of such metric with aggregation functions $h(z) = z$, $g(z_0, z_1, \ldots, z_n) = sum(z_0, z_1, \ldots, z_n)$ and $f(z) = z$.

In the context of product quantization, this is a very important property because during inference the algorithm calculates distances by parts. This means that it would be much more problematic to use metrics for product quantization that do not have this property. Cosine distance is an example of such metric.

If there is still a need to use a metric without this property, then additional heuristics need to be applied to aggregate partial distances with some error.

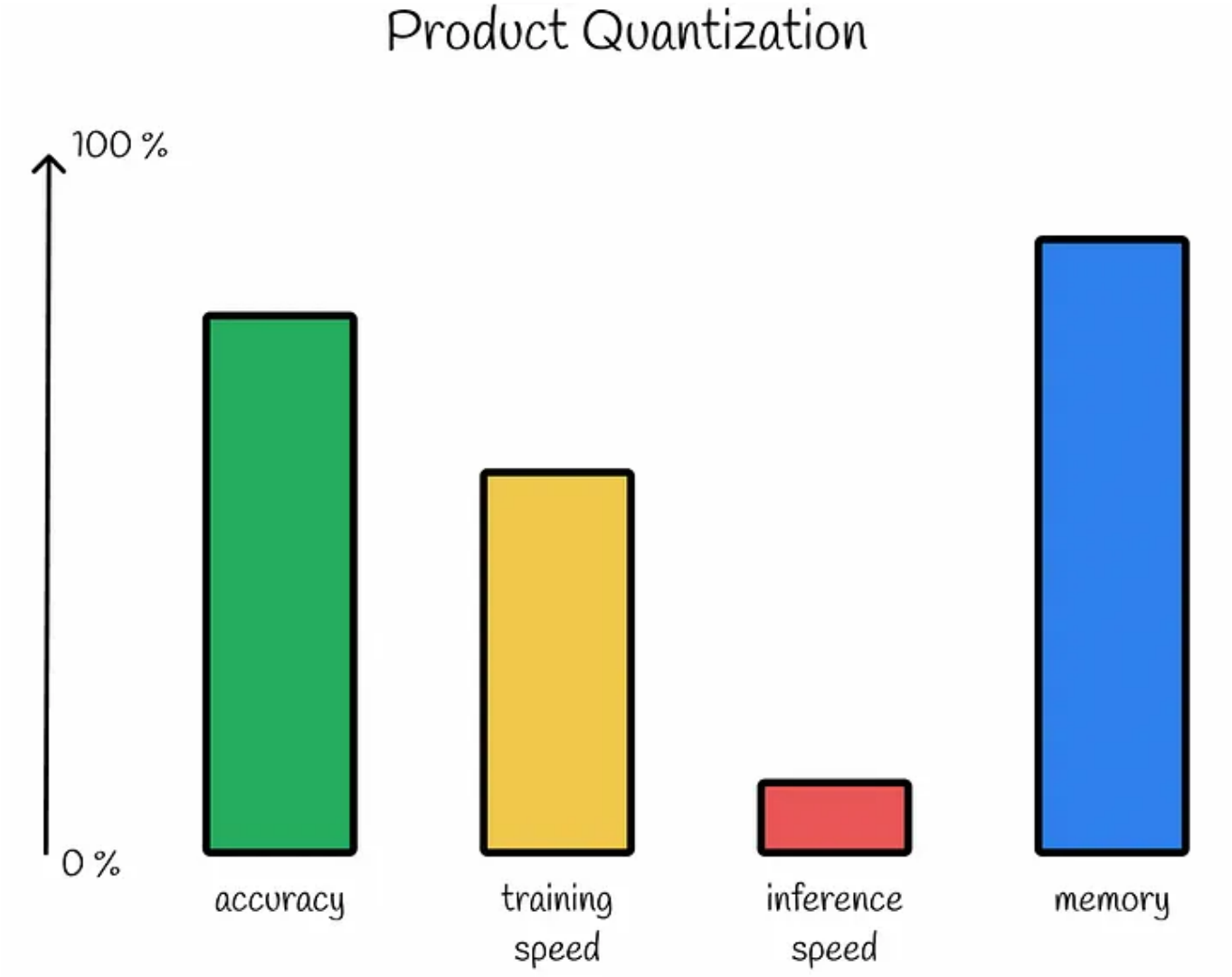
Performance

The main advantage of the product quantization is a massive compression of database vectors which are stored as short PQ codes. For some applications, such compression rate may be even higher than 95%! However, apart from

PQ codes, the matrix d of size k x n containing quantized vectors of each subspace needs to be stored.

> Product quantization is a lossy-compression method, so the higher the compression is, the more likely that the prediction accuracy will decrease.

Building a system for efficient representation requires training several cluster algorithms. Apart from it, during inference, $k * n$ partial distances need to be calculated in a brute-force manner and summed up for each of the database vectors which may take some time.



Product Quantization performance

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 product quantization is utilized.

Product quantization is implemented in the *IndexPQ* class. For initialisation, we need to provide it 3 parameters:

- **d**: number of dimensions in data.
- **M**: number of splits for each vector (the same parameter as n used above).
- **nbits**: number of bits it takes to encode a single cluster ID. This means that the number of total clusters in a single subspace will be equal to $k = 2\text{^}nbits$.

For equal subspace dimensions splitting, the parameter *dim* must be divisible by M.

The total number of bytes required to store a single vector is equal to:

$$bytes = \left\lceil \frac{M * nbits}{8} \right\rceil$$

> *As we can see in the formula above, for more efficient memory usage the value of M * nbits should be divisible by* 8.

```
d = 256 # data dimension
M = 4 # number of subspaces (splits for each vector)
nbits = 8 # number of bits required to encode a single cluster ID in each subspace

assert d % M == 0
index = faiss.IndexPQ(d, M, nbits)

index.train(data)
index.add(data)

k = 3 # how many nearest neighbours to search for
D, I = index.search(query, k) # retuning closest distances and neighbours
```

Faiss implementation of IndexPQ

Conclusion

We have looked through a very popular algorithm in information retrieval systems that efficiently compresses large volumes of data. Its principal downside is a slow inference speed. Despite this fact, the algorithm is widely used in modern Big data applications, especially in combination with other similarity search techniques.

In the first part of the article series, we described the workflow of the inverted file index. In fact, we can merge these two algorithms into a more efficient one which will possess the advantages of both! This is what exactly we are going to do in the next part of this series.

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

In the first two parts of this series we have discussed two fundamental algorithms in information retrieval: inverted...

medium.com

Resources

- Product quantization for nearest neighbour search
- Faiss documentation
- Faiss repository
- Summary of Faiss indexes

All images unless otherwise noted are by the author.

Similarity Search | Quantization | Faiss | Machine Learning | Hands On Tutorials

More from the list: "NLP"

Curated by Himanshu Birla

 Jon Gi... in Towards Data ...

 Jon Gi... in Towards Data ...

 Jon Gi... in

The Word2ve

Characteristics of Word Embeddings
✦ · 11 min read · Sep 4, 2021

The Word2vec Hyperparameters
✦ · 6 min read · Sep 3, 2021

✦ · 15 min rea

View list

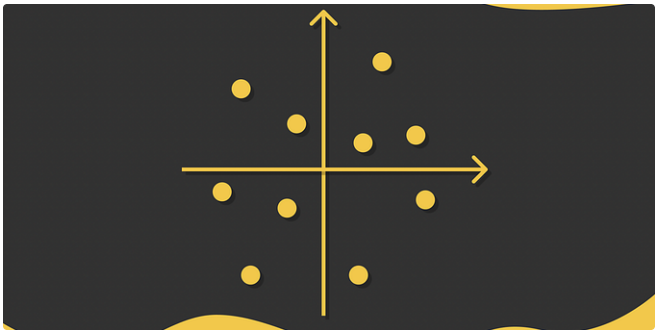
Written by Vyacheslav Efimov

Following

470 Followers · Writer for Towards Data Science

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

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

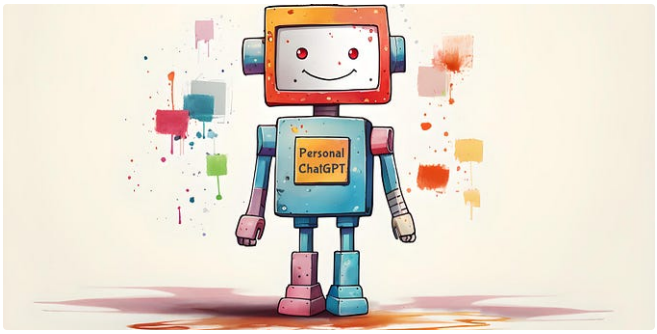
Antonis Makropoulos in Towards Data Science

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

128 549 11



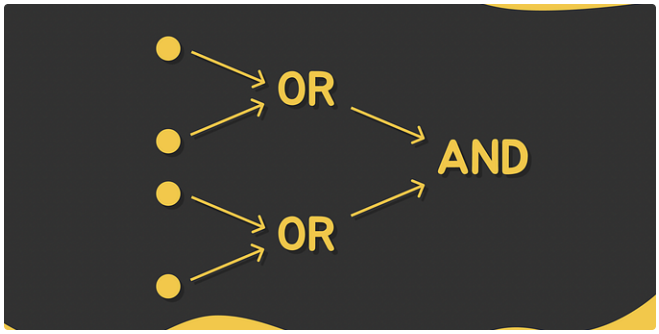
Robert A. Gonsalves in Towards Data Science

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

595 7



Vyacheslav Efimov in Towards Data Science

Similarity Search, Part 7: LSH Compositions

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

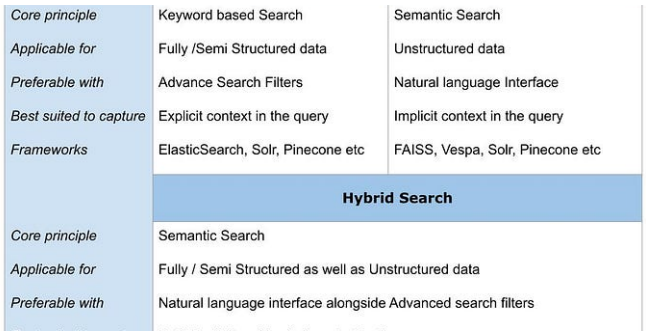
11 min read · Jul 24

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

Alyx

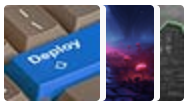
Semantic Search with FAISS

HuggingFace get_neareast_example and Cosine Similarity Search

9 min read · Jul 15

71 1

Lists



Predictive Modeling w/ Python

20 stories · 452 saves

Practical Guides to Machine Learning

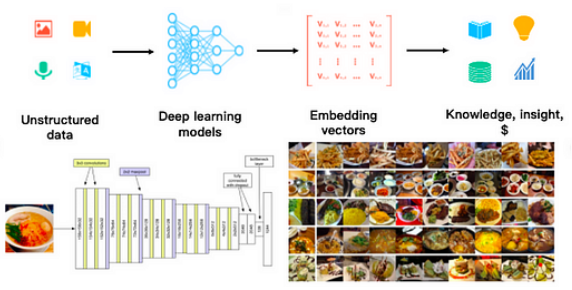
10 stories · 519 saves

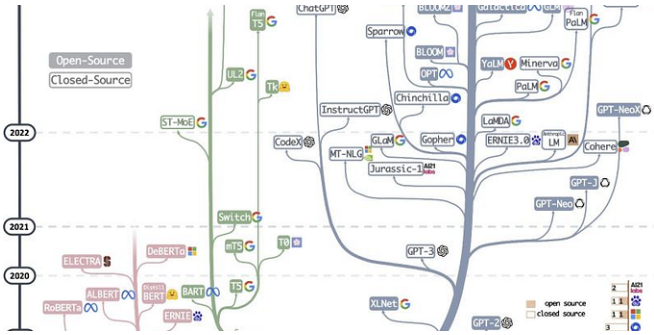
Natural Language Processing

669 stories · 283 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves





Jayita Bhattacharyya in GoPenAI

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

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

9 min read · Aug 16

228 1

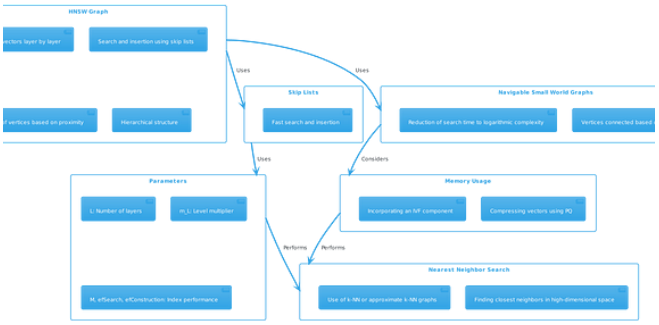
Haifeng Li

A Tutorial on LLM

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

15 min read · Sep 14

372



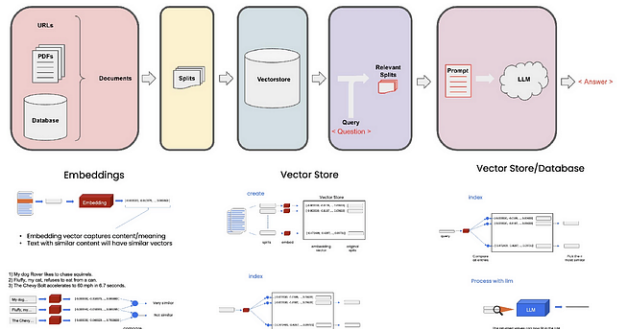
Bidhan R

Harnessing the Power of Hierarchical Navigable Small Wor...

Ever wondered how artificial intelligence systems retrieve precise, context-specific...

3 min read · Jun 17

9



TeeTracker

Chat with your PDF (Streamlit Demo)

Conversation with specific files

4 min read · Sep 15

56

See more recommendations