✨ Member-only story

Product Quantization for Similarity Search

How to compress and fit a humongous set of vectors in memory for similarity search with asymmetric distance computation (ADC)

Peggy Chang · Follow
Published in Towards Data Science
8 min read · May 9, 2022

Listen Share ••• More

Photo by Markus Winkler on Unsplash

Similarity search and nearest neighbor search are very popular and widely used in many fields. They are used in recommendation systems, in online stores and marketplaces that enable product image searches, or in systems specialized in document, media, or object matching and retrieval.

Similarity search is often done over a large collection of object embeddings, usually in the form of high-dimensional vectors.

Fitting a humongous set of high-dimensional vectors in memory to perform similarity search is a challenge, and product quantization can help to overcome this with some tradeoffs.

What is Product Quantization

Product quantization (PQ) is a technique used for vector compression. It is very effective in compressing high dimensional vectors for nearest neighbor search. According to the authors of **Product Quantization for Nearest Neighbor Search** [1],

> "The idea is to decompose the space into a Cartesian product of low dimensional subspaces and to quantize each subspace separately.
>
> A vector is represented by a short code composed of its subspace quantization indices. The Euclidean distance between two vectors can be efficiently estimated from their codes. An asymmetric version increases precision, as it computes the approximate distance between a vector and a code".

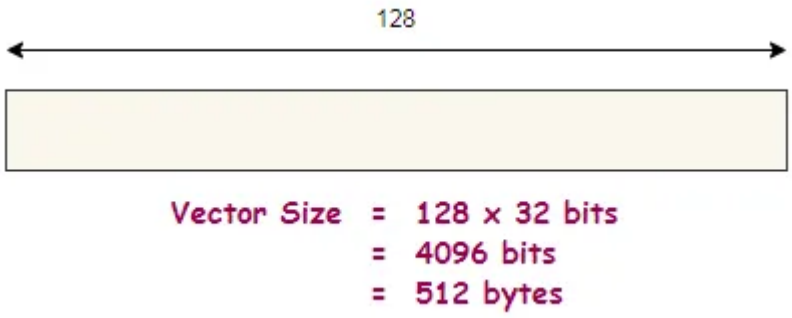
Note that product quantization is not dimensionality reduction.

In product quantization, which is a form of vector quantization, the number of vectors will remain the same after quantization. However, the values in the compressed vector are now

> *transformed into short codes, and therefore they are symbolic and are no longer numeric. With this representation, the size of each vector is reduced dramatically.*
>
> *Product quantization is also one of the many index types implemented in Faiss (Facebook AI Similarity Search), a library that is highly optimized for efficient similarity search.*

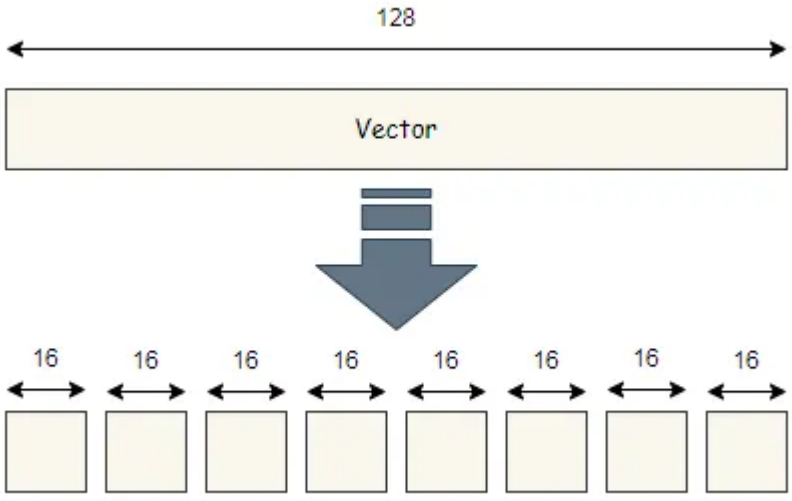
How Product Quantization Works

Let's say we have a collection of vectors in the database, and the dimension (or length) of each vector is 128. This means the size of one vector is 128 x 32 bits = 4096 bits (which is equivalent to 512 bytes).



All images are by the author unless otherwise specified

First, we divide and split the vector into segments. The diagram below shows that the vector is divided and split into 8 segments, where the length of each segment is 16.



Dividing and splitting a vector into segments

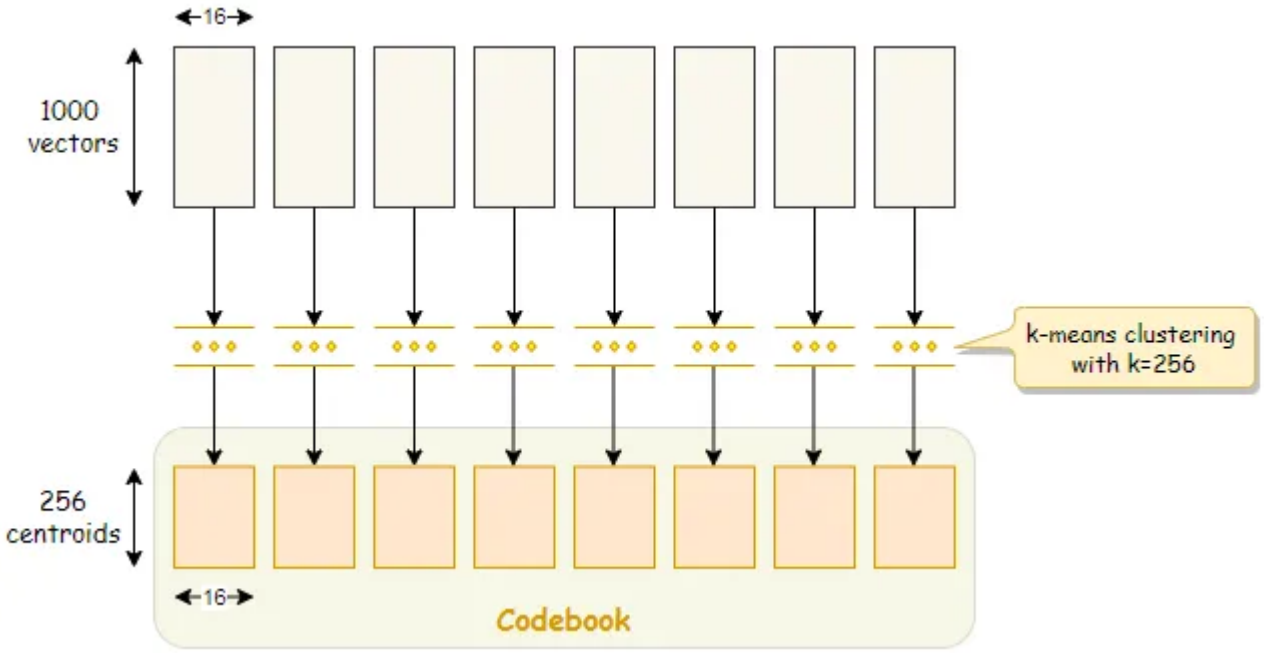
We do this for every vector, and hence what we're doing is essentially separating the vectors into 8 different sub-spaces. If we have 1000 vectors in the database, then each sub-space would contain 1000 segments of length 16.

Training

Next, we proceed to train our vectors by running k-means clustering on each of the sub-space. Based on the value of `k` that we choose, the k-means clustering would generate `k` centroids (i.e. cluster centers) within that sub-space, and these centroids have the same length as the segment.

The centroids are also known as reproduction values. The set of centroids is called the codebook, and we would talk more about it later.

As an example, if we choose `k=256`, we would end up having a total of 256 x 8 = 2048 centroids.



Running k-means clustering on each sub-space

> *The centroids are also known as reproduction values because they can be used to approximate the reconstruction of the vectors by concatenating the respective centroids from each segment. However, the reconstructed vectors would not be exactly the same as the original vectors, as product quantization is a lossy compression.*

> *For training, a different set or a subset of vectors could also be used, as long as they have the same distribution as the database vectors.*

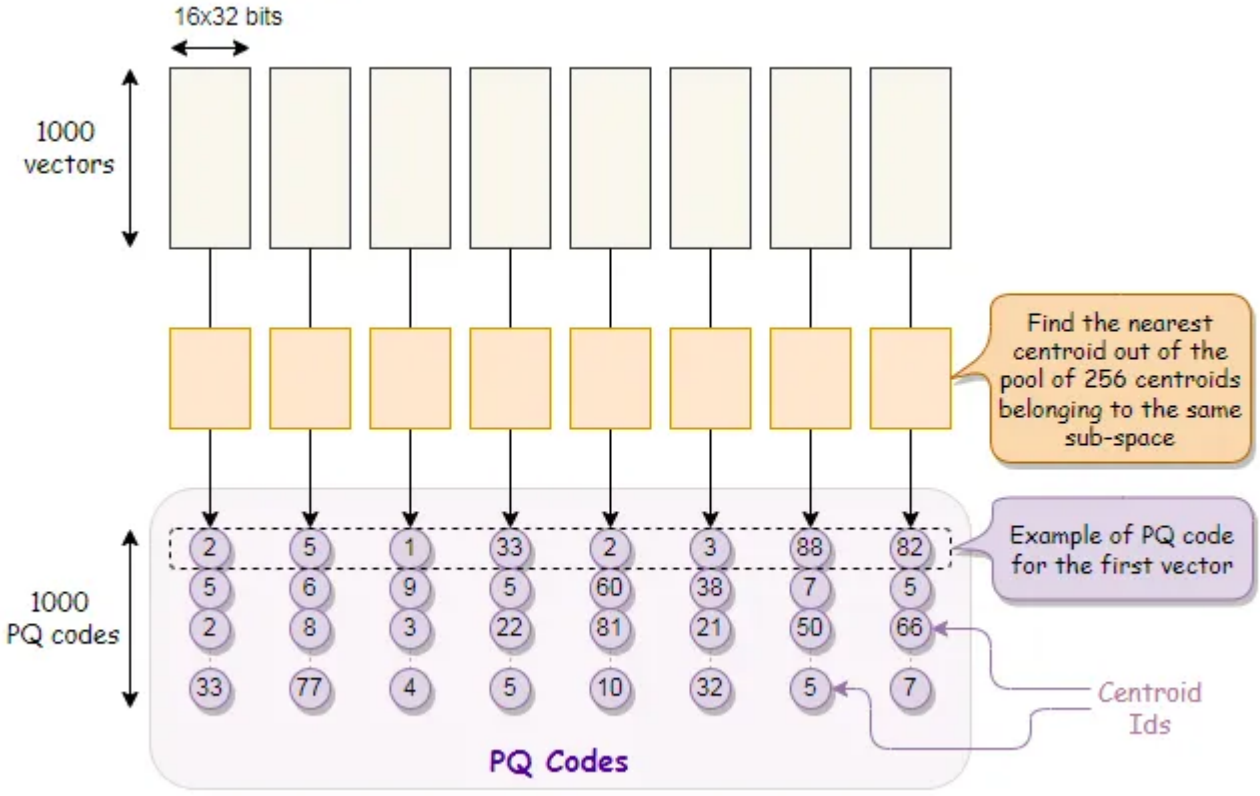
Encoding

After training is completed, for each segment of vector in the database, we find the nearest centroid from the respective sub-space. In other words, for each segment of the vector, we only need to find the nearest centroid out of the pool of 256 centroids belonging to the same sub-space.

For each segment, after obtaining the nearest centroid, we substitute it with that centroid's Id. The centroid Ids are nothing but indices of the centroids (a number from 0 to 255) within the sub-space.

And there we have them, they are the compressed representation of the vectors. These are the short codes that we talked about earlier, and let's refer to them as PQ codes.

In this example, one PQ code consists of 8 centroid Ids across the segments. We have 1000 vectors in the database, so that would transform into 1000 PQ codes.

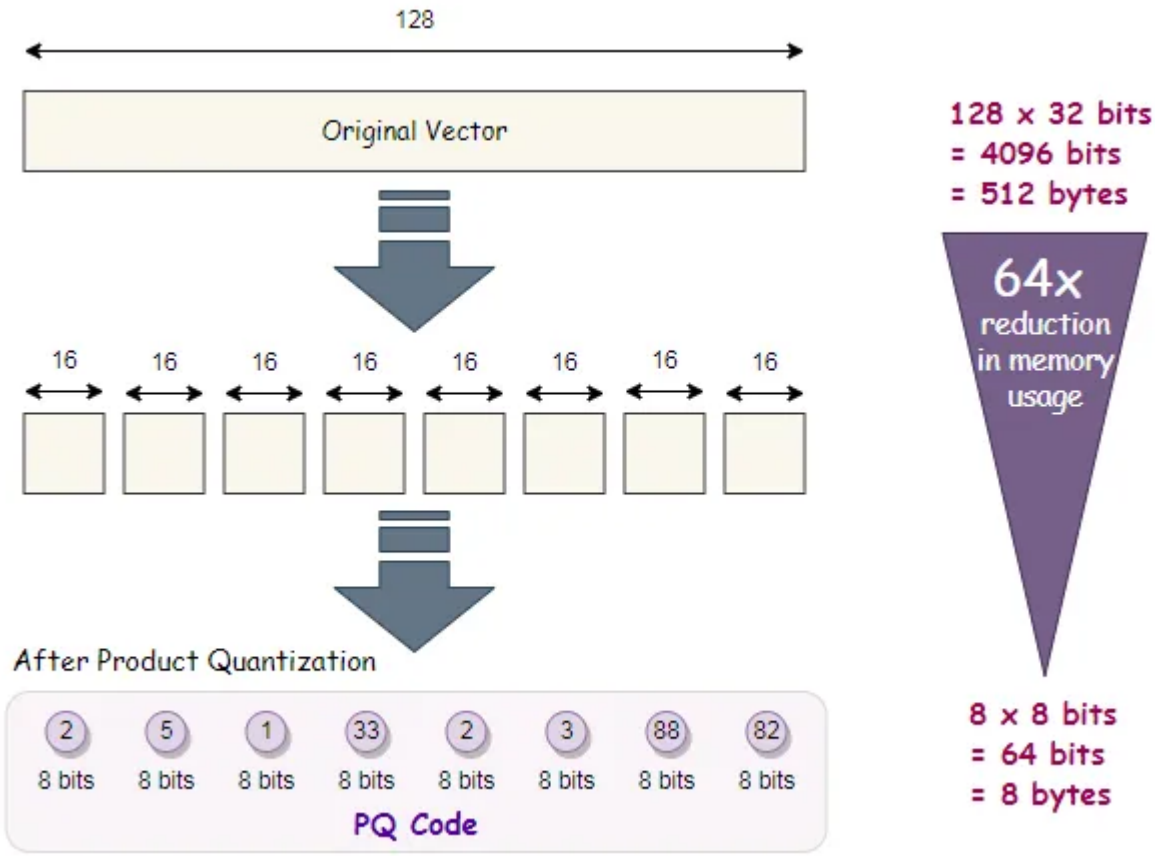


PQ Codes, the compressed representation of the vectors

Basically what we've done is encode our original vectors with centroid Ids, where each segment is encoded with 8 bits.

There are 8 segments per vector, thus every encoded vector takes up only 8 x 8 bits = 8 bytes of space with this representation. Compared to storing the original vector of 512 bytes, that is a huge saving of space.

As demonstrated in this example, we got a whopping 64 times reduction in memory usage (from 512 bytes to 8 bytes per vector), and that's a significant amount when we're dealing with hundreds of thousands of records, if not millions!



Memory usage reduction after product quantization

Photo by krakenimages on Unsplash

> *The value of* `k` *is usually a power of two.*
>
> *For* `M` *segments, the memory requirement for one PQ code is* `M*(log base 2 of k)` *bits.*

Searching with Quantization

So we are effectively using 8 centroid Ids to represent a vector. But how exactly does this representation work for similarity search?

The answer lies in the codebook, which contains centroids, the reproduction values. The process is explained below.

Given a query vector `q`, our goal is to find vectors that are very similar to `q` from the collection of vectors that we have in the database.

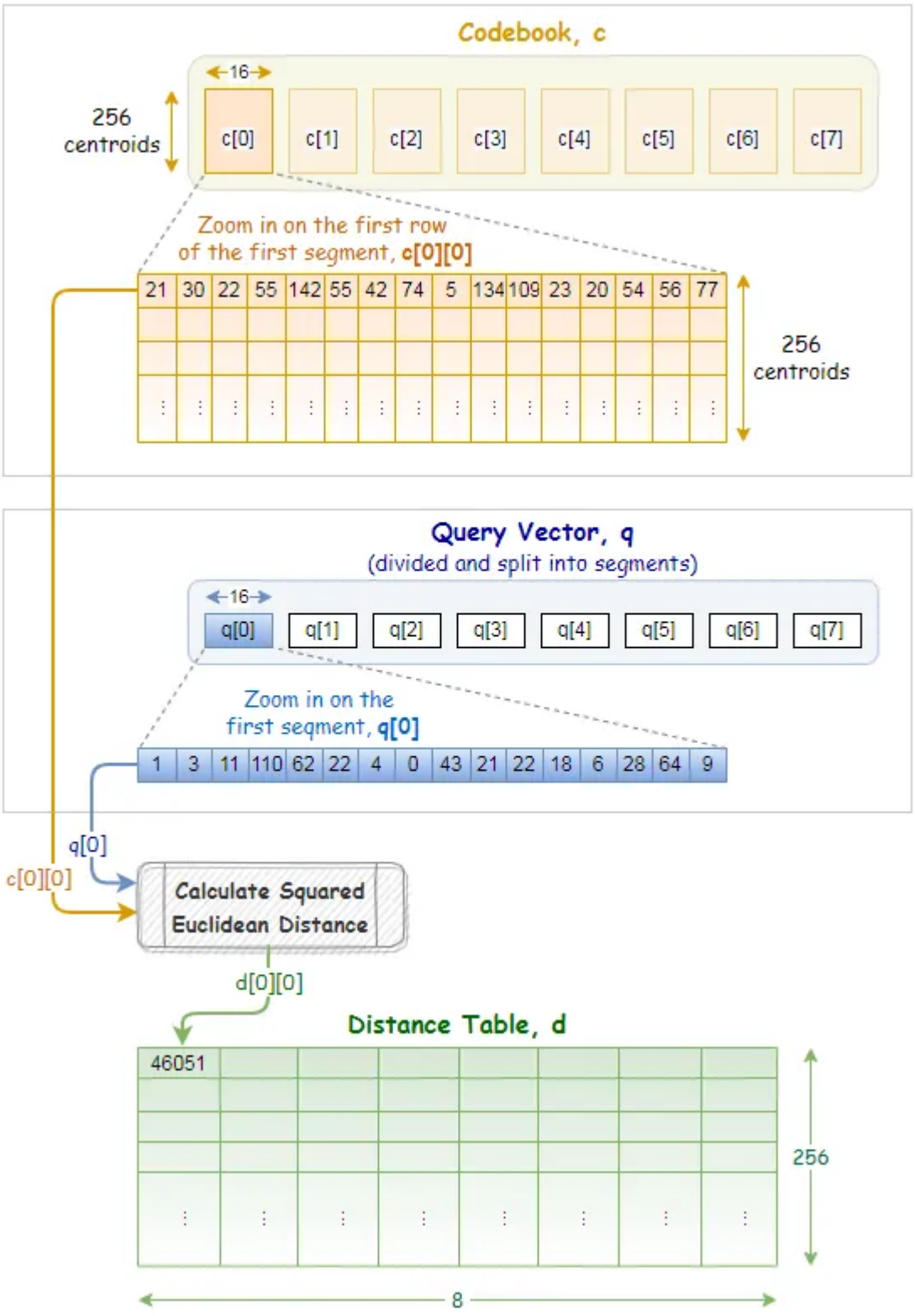
A typical process would be to calculate and compare the distances between the query vector and all the vectors in the database and return the top N records with the shortest distance.

However, we're going to do something different. We're not going to use the original vectors at all for the distance calculation. Instead, we will do asymmetric distance computation (ADC) and estimate the distances using the vector-to-centroid distances.

We begin by dividing and splitting the query vector into the same number of segments.

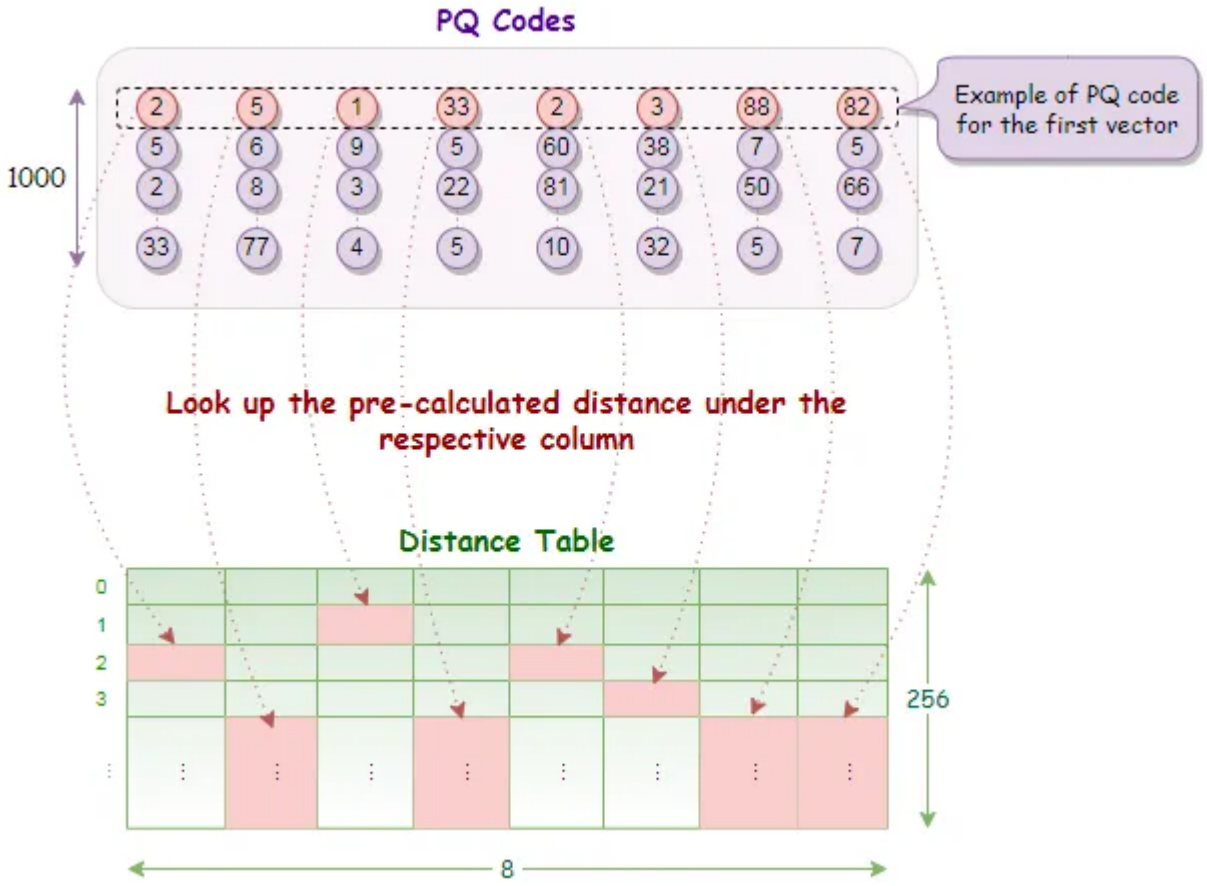
For each query vector segment, we pre-calculate the partial squared Euclidean distance with all the centroids of the same segment from the codebook.

These partial squared Euclidean distances are recorded in a distance table `d`. In our example, as shown below, the distance table consists of 256 rows and 8 columns.



Pre-compute the Distance Table

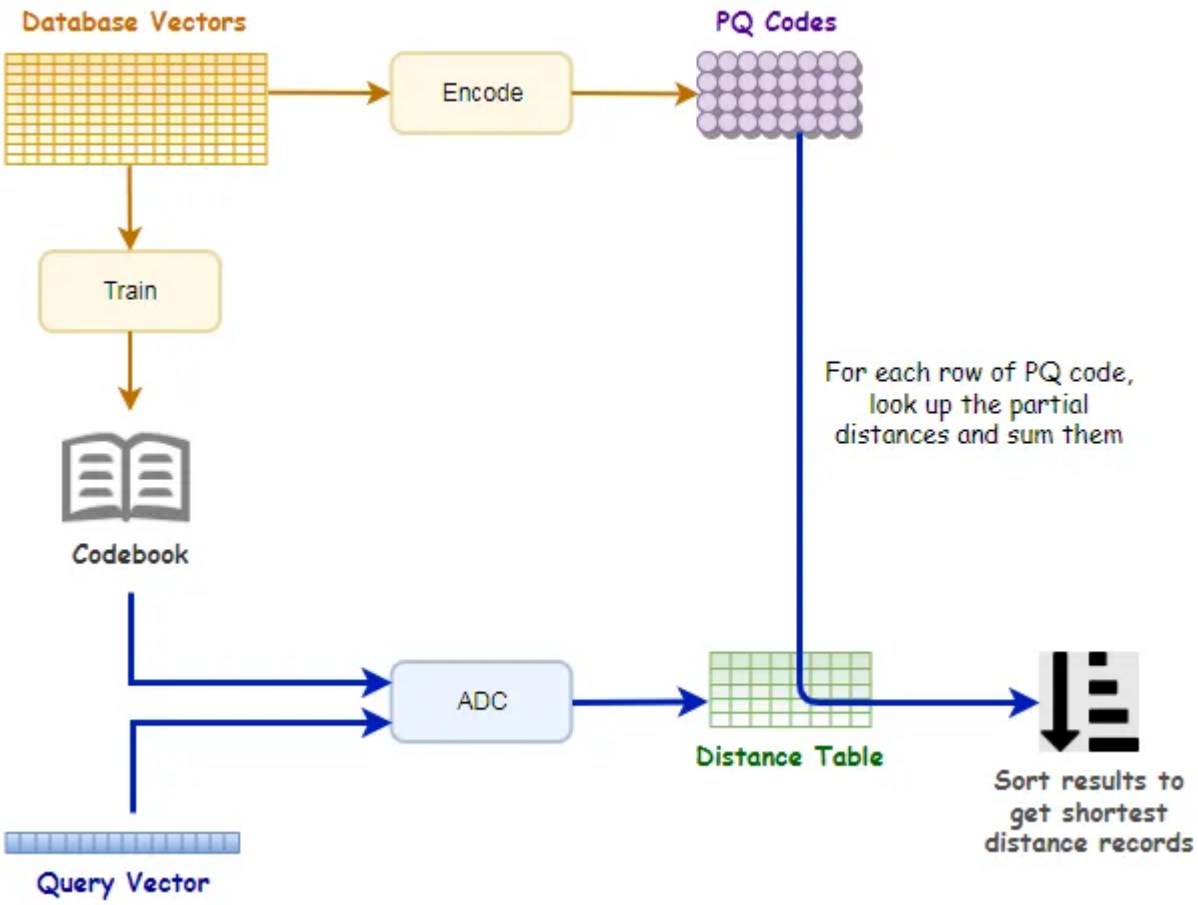
Now that we have the distance table, we can obtain the distance for each row of PQ code easily just by looking up the partial distances and doing a summation.



Look up partial distances from the Distance Table

After obtaining the distances for all rows of PQ codes, we sort them in ascending order and from the top results (i.e. records with the shortest distance), find and return the corresponding vectors from the database.

The following diagram summarizes the product quantization process for similarity search.



Product quantization process for similarity search

It is worth noting that with product quantization, the search is still a brute force search, as the distance lookup and summation are exhaustive and need to be done for all rows of PQ codes.

Also, since we're comparing vector-to-centroid distances, the distances are not exact vector-to-vector distances. They are just estimated distances, and thus the results may be less precise and may not always be the true nearest neighbors.

The search quality can be improved by tuning the number of centroids or the number of segments. More centroids or segments result in higher accuracy and precision, but they would also slow down the search operation as well as the time required for training and encoding. Other than that, more centroids may potentially lead to an increase in the number of bits required to represent a code, and hence lesser memory savings.

Here's a glimpse of simple Python implementation of product quantization, inspired and adapted from nanopq.

```python
import numpy as np
from scipy.cluster.vq import kmeans2, vq
from scipy.spatial.distance import cdist

def PQ_train(vectors, M, k):
    s = int(vectors.shape[1] / M)                       # Dimension (or length) of a segment.
    codebook = np.empty((M, k, s), np.float32)

    for m in range(M):
        sub_vectors = vectors[:, m*s:(m+1)*s]           # Sub-vectors for segment m.
        codebook[m], label = kmeans2(sub_vectors, k)    # Run k-means clustering for each segment.

    return codebook
#-------------------------------------------------------------------------------------------

def PQ_encode(vectors, codebook):
    M, k, s = codebook.shape
    PQ_code = np.empty((vectors.shape[0], M), np.uint8)

    for m in range(M):
        sub_vectors = vectors[:, m*s:(m+1)*s]            # Sub-vectors for segment m.
        centroid_ids, _ = vq(sub_vectors, codebook[m])  # vq returns the nearest centroid Ids.
        PQ_code[:, m] = centroid_ids                     # Assign centroid Ids to PQ_code.

    return PQ_code
#-------------------------------------------------------------------------------------------

def PQ_search(query_vector, codebook, PQ_code):
    M, k, s = codebook.shape
    #=========================================================================
    # Build the distance table.
    #=========================================================================

    distance_table = np.empty((M, k), np.float32)    # Shape is (M, k)

    for m in range(M):
        query_segment = query_vector[m*s:(m+1)*s]    # Query vector for segment m.
        distance_table[m] = cdist([query_segment], codebook[m], "sqeuclidean")[0]

    #=========================================================================
    # Look up the partial distances from the distance table.
    #=========================================================================

    N, M = PQ_code.shape
    distance_table = distance_table.T                # Transpose the distance table to shape (k, M)
```

```
    distance_table = distance_table.T                    # Transpose the distance table to shape (K, M)
    distances = np.zeros((N, )).astype(np.float32)

    for n in range(N):                                   # For each PQ Code, lookup the partial distanc
        for m in range(M):
            distances[n] += distance_table[PQ_code[n][m]][m] # Sum the partial distances from all

    return distance_table, distances

#-----------------------------------------------------------------------------------------------
# Test case

M = 8                    # Number of segments
k = 256                  # Number of centroids per segment
vector_dim = 128         # Dimension (length) of a vector
total_vectors = 1000000  # Number of database vectors

# Generate random vectors
np.random.seed(2022)
vectors = np.random.random((total_vectors, vector_dim)).astype(np.float32)   # Database vectors
q = np.random.random((vector_dim, )).astype(np.float32)                      # Query vector

# Train, encode and search with Product Quantization
codebook = PQ_train(vectors, M, k)
PQ_code = PQ_encode(vectors, codebook)
distance_table, distances = PQ_search(q, codebook, PQ_code)
# All the distances are returned, you may sort them to get the shortest distance.
```

PQ.py hosted with ❤ by GitHub view raw

Summary

Product quantization divides and splits vectors into segments, and quantizes each segment of the vectors separately.

Each vector in the database is converted to a short code (PQ code), a representation that is extremely memory-efficient for the approximate nearest neighbor search.

Similarity search with product quantization is highly scalable, but we trade some precision for memory space.

> At the expense of a less precise search, product quantization enables a large-scale search that would otherwise not be possible.

As product quantization alone is not the most effective method for very large-scale search, we will see how we can implement a faster search method that is non-

Reference

[1] H. Jégou, M. Douze, C. Schmid, Product Quantization for Nearest Neighbor Search (2010)

[2] C. McCormick, Product Quantizers for k-NN Tutorial Part 1 (2017)

[3] J. Briggs, Product Quantization: Compressing high-dimensional vectors by 97%

[4] Nano Product Quantization (nanopq)

Before You Go...

Thank you for reading this post, and I hope you've enjoyed learning about product quantization for similarity search.

If you like my post, don't forget to hit ***Follow*** *and* ***Subscribe*** *to get notified via email when I publish.*

Optionally, you may also sign up for a Medium membership to get full access to every story on Medium.

Visit this GitHub repo for all codes and notebooks that I shared in my posts.

Interested to read my other data science articles? Check out the following:

Transformers, can you rate the complexity of reading passages?

Fine-tuning RoBERTa with PyTorch to predict reading ease of text excerpts

towardsdatascience.com

Advanced Techniques for Fine-tuning Transformers

Learn these advanced techniques and see how they can help improve results

towardsdatascience.com

Building a Product Recommendation Engine with AWS SageMaker

Learn how to build and train a personalized recommender engine with Amazon SageMaker Factorization Machines

pub.towardsai.net

AWS Certified Machine Learning — Specialty

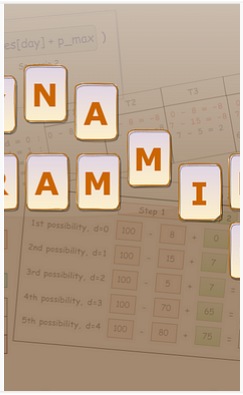
Tips and suggestions on how to prepare and pass the exam

towardsdatascience.com

Peggy Chang

Series on Mastering Dynamic Programming

View list 2 stories



Similarity Search | Product Quantization | Data Science | Artificial Intelligence | Editors Pick

Follow

Written by Peggy Chang

509 Followers · Writer for Towards Data Science

Machine Learning | Data Science | “Getting 1% better every day” | https://www.linkedin.com/in/peggy1502/

More from Peggy Chang and Towards Data Science

Peggy Chang in Towards Data Science

IVFPQ + HNSW for Billion-scale Similarity Search

The best indexing approach for billion-sized vector datasets

· 17 min read · Aug 29, 2022

Bex T. in Towards Data Science

130 ML Tricks And Resources Curated Carefully From 3 Years (Plus Free eBook)

Each one is worth your time

· 48 min read · Aug 1

2.8K 10

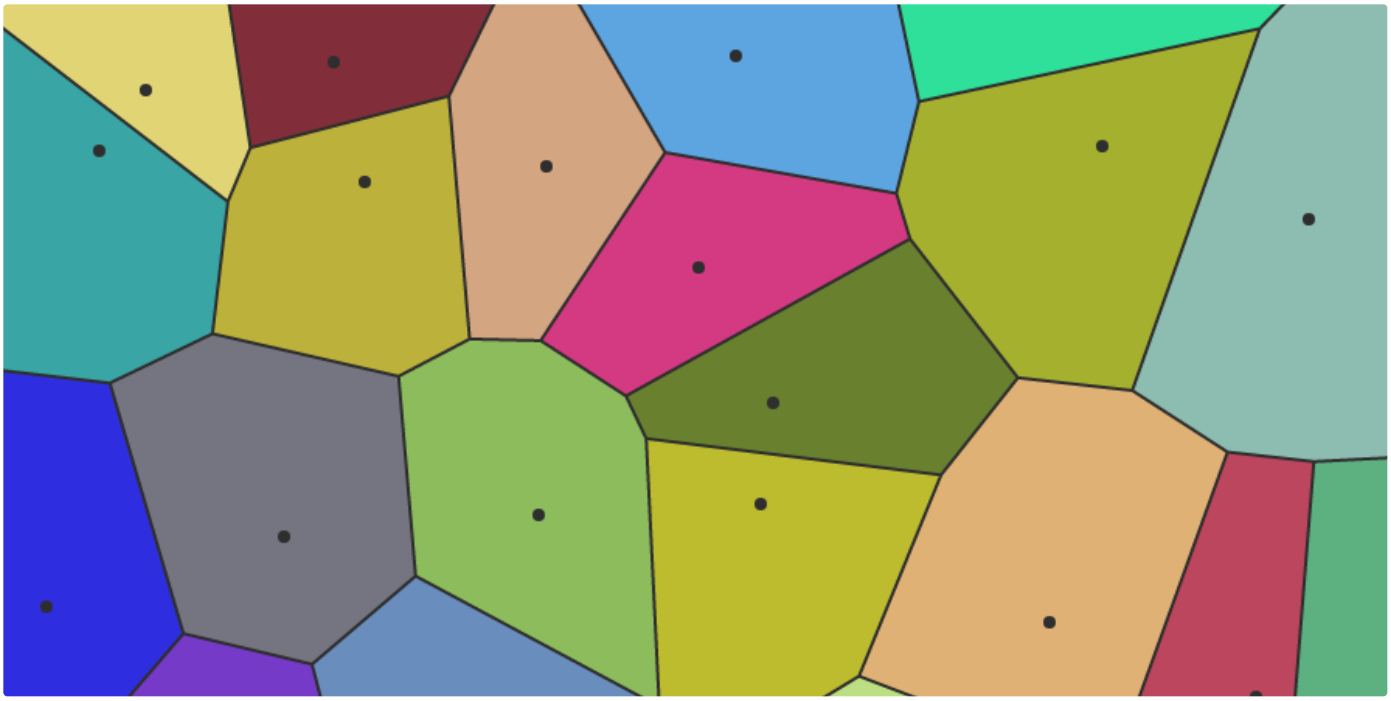
Maxime Labonne in Towards Data Science

Fine-Tune Your Own Llama 2 Model in a Colab Notebook

A practical introduction to LLM fine-tuning

· 12 min read · Jul 25

1.7K 31

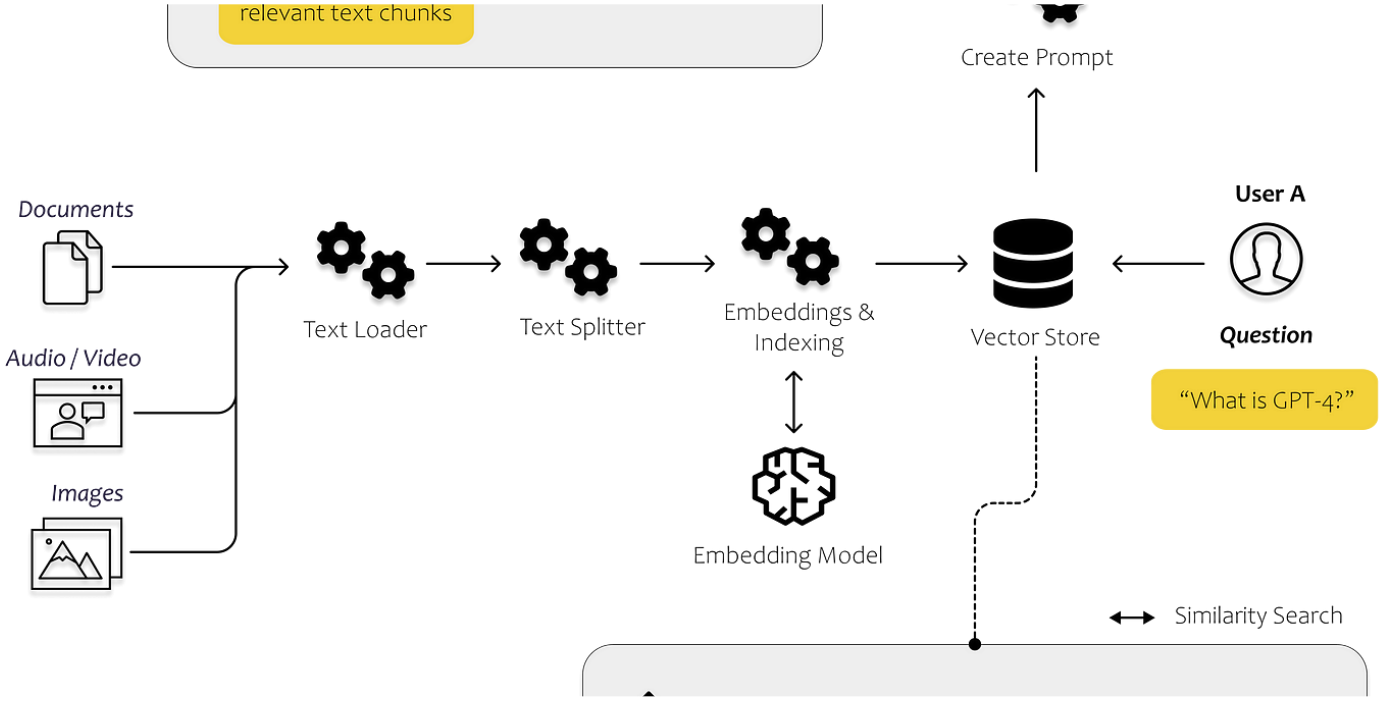


Peggy Chang in Towards Data Science

Similarity Search with IVFPQ

Find out how the inverted file index (IVF) is implemented alongside product quantization (PQ) for a fast and efficient approximate nearest...

· 9 min read · May 25, 2022



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 21

4.4K 40

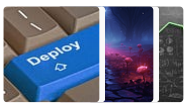
Pratyush Khare in MLearning.ai

How to perform High-Performance Search using FAISS

A Beginner's Guide to FAISS, use-cases, Mathematical foundations & implementation

6 min read · Mar 4

Lists



Predictive Modeling w/ Python

20 stories · 292 saves

ChatGPT prompts

24 stories · 270 saves

ChatGPT

21 stories · 114 saves

AI Regulation

6 stories · 82 saves

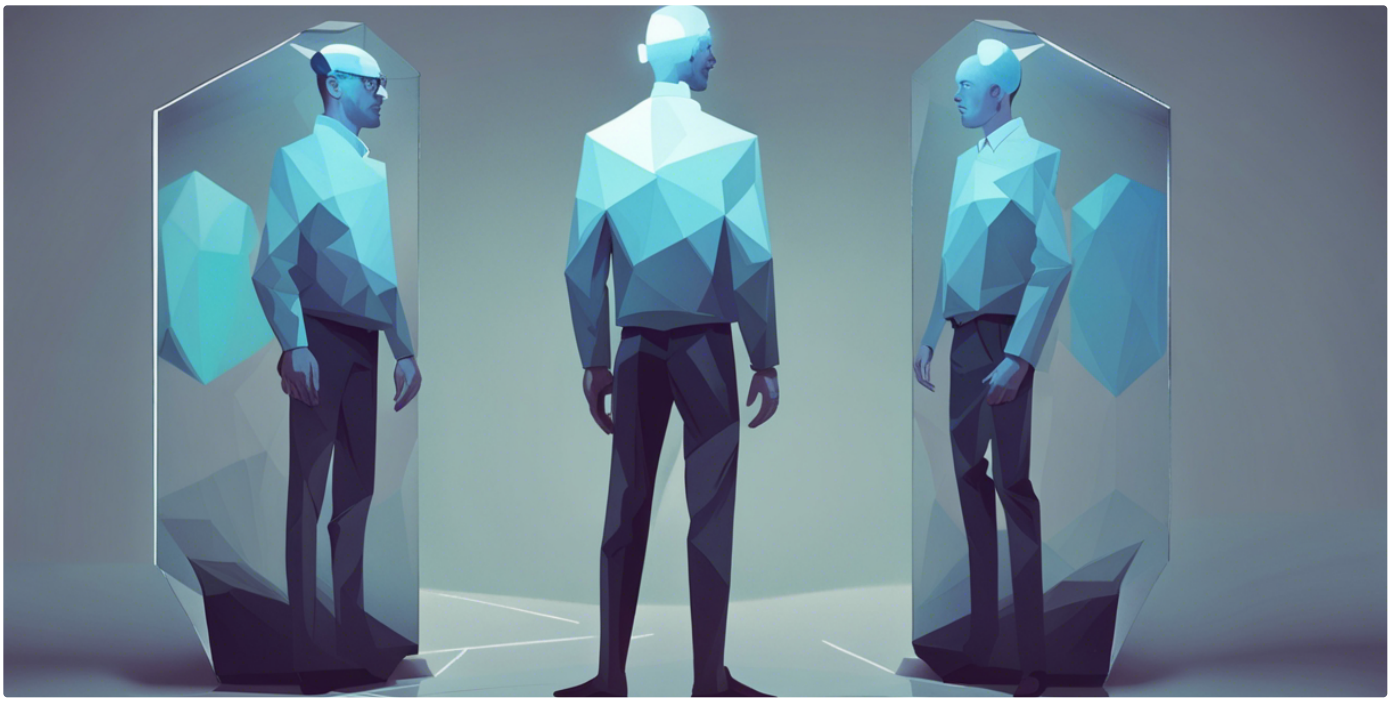
Amod's Notes

Understanding Retrieval-Augmented Generation: A Simple Guide

Have you ever asked an AI language model like ChatGPT about the latest developments on a certain topic, only to receive this response: 'I...

7 min read · Jul 2

41 1



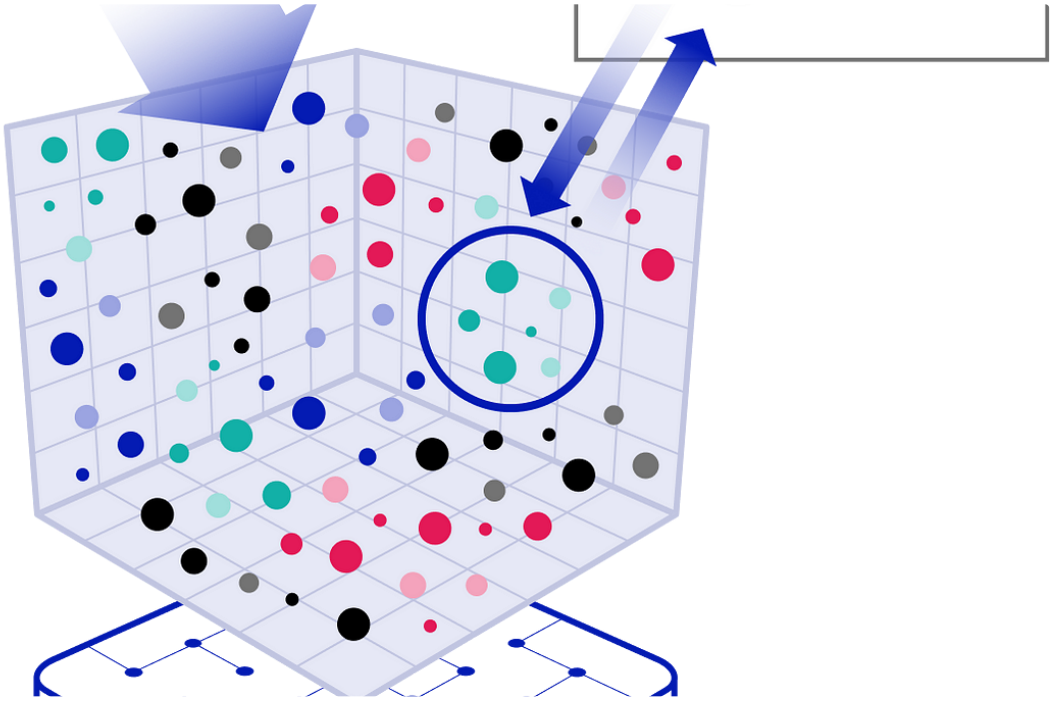
Sergei Savvov in Better Programming

Create a Clone of Yourself With a Fine-tuned LLM

Unleash your digital twin

11 min read · Jul 27

1.8K 14



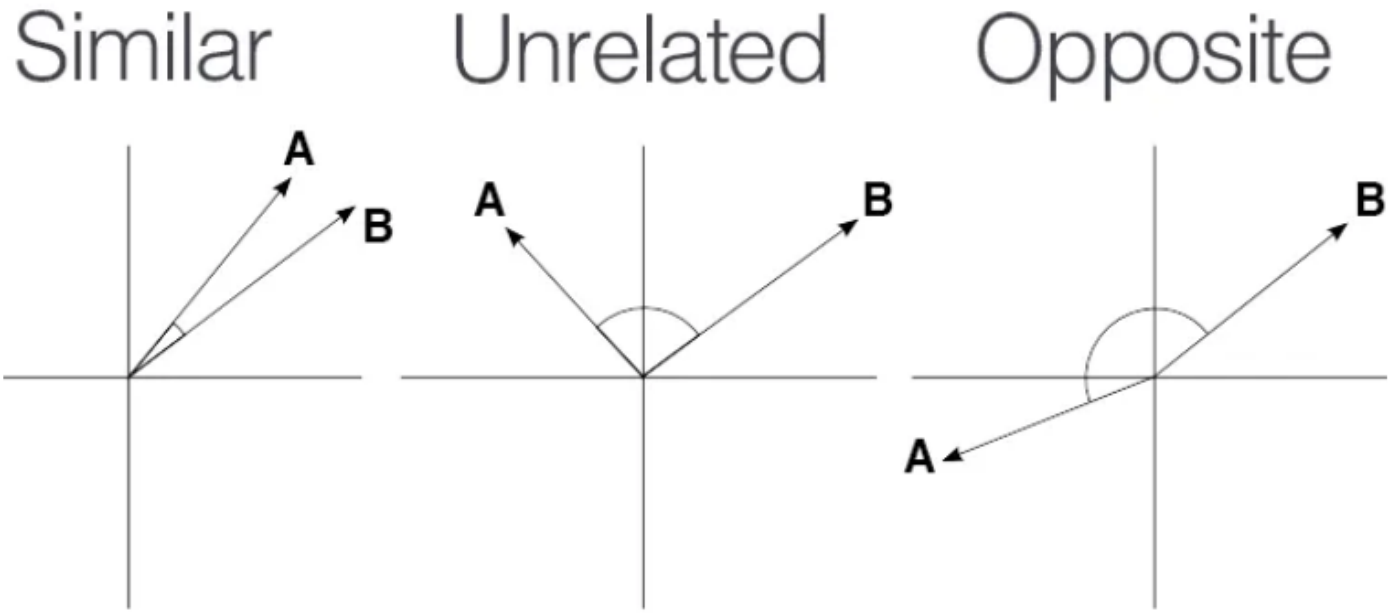
Pankaj Pandey

Faiss: Efficient Similarity Search and Clustering of Dense Vectors

Faiss is a powerful library designed for efficient similarity search and clustering of dense vectors. It offers various algorithms for...

3 min read · Jun 13

57




Milana Shkhanukova

Cosine distance and cosine similarity

-Okay, Milana, there is a mistake: cosine similarity cannot be negative. - Oh, it can be.

3 min read · Mar 4


19

See more recommendations