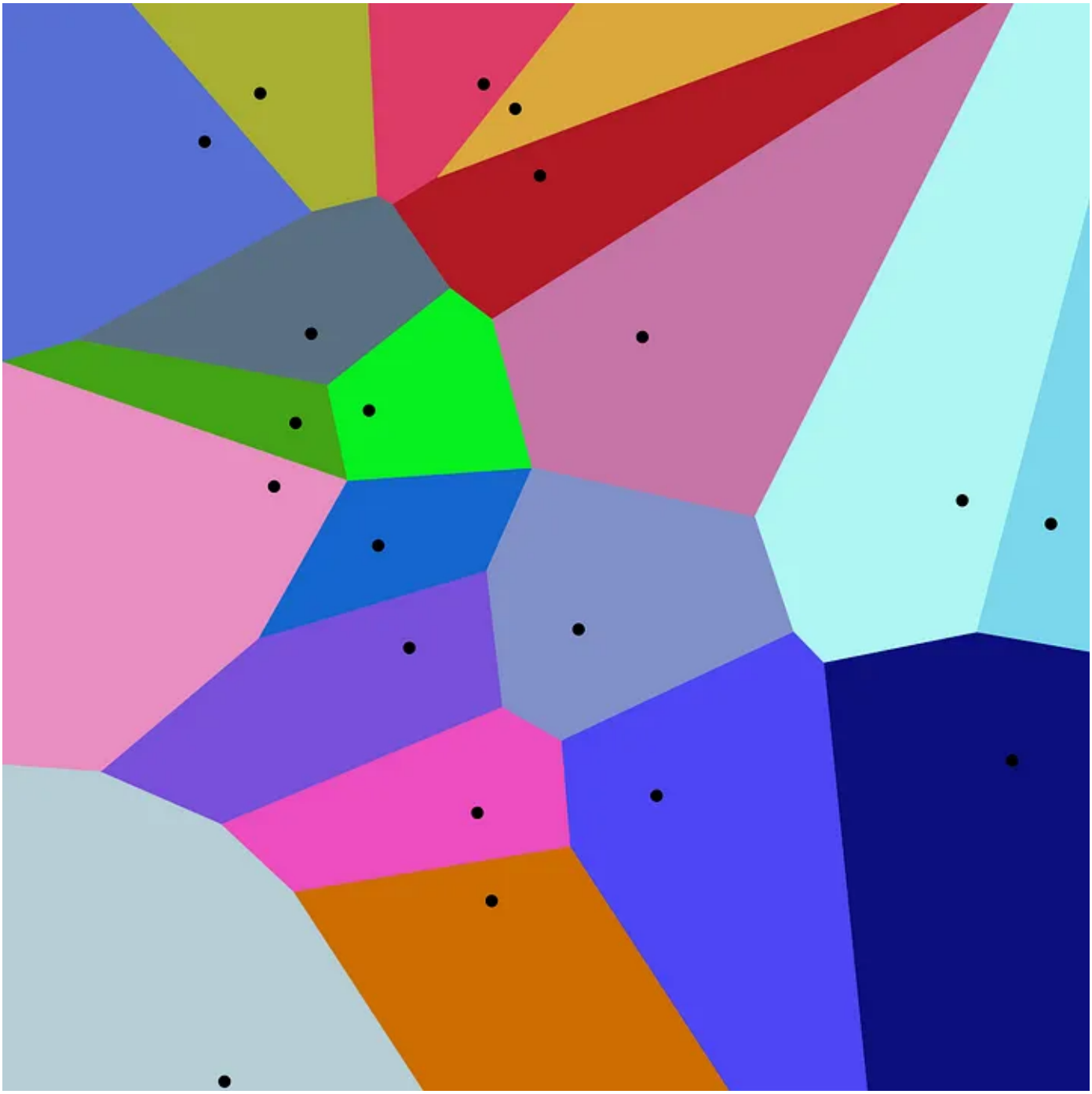
✦ Member-only story

Comprehensive Guide To Approximate Nearest Neighbors Algorithms

Eyal Trabelsi · Follow
Published in Towards Data Science
16 min read · Feb 14, 2020

Listen Share ••• More

Search In Practice- Approximate Nearest Neighbors



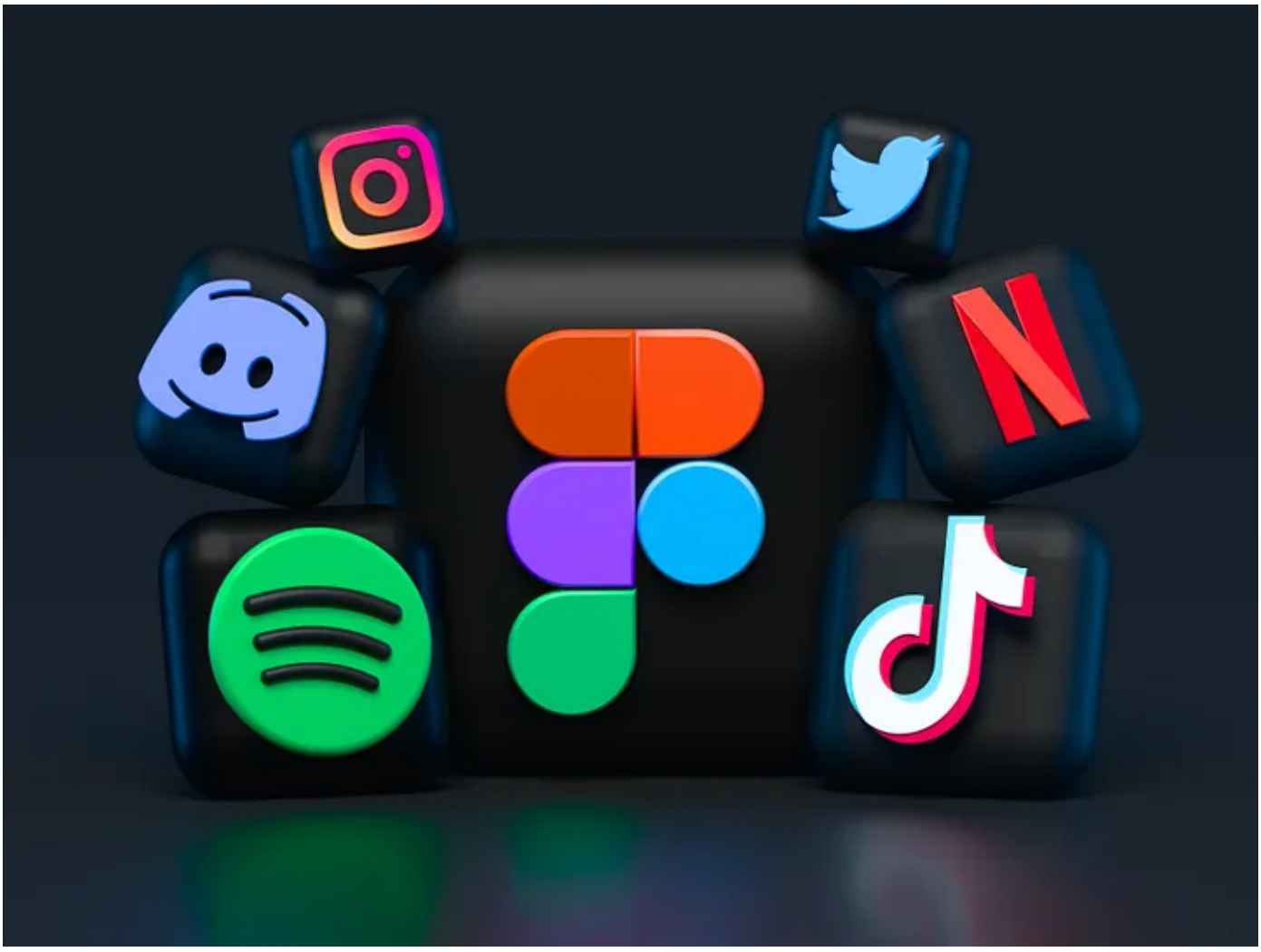
source: https://en.wikipedia.org/wiki/Proximity_analysis#/media/File:Euclidean_Voronoi_diagram.svg

Nearest Neighbors Motivation

Today as users consume more and more information from the internet at a moment's notice, there is an increasing need for efficient ways to do search. **This is why "Nearest Neighbor" has become a hot research topic, in order to increase the chance of users to find the information they are looking for in reasonable time.**

The use cases for "Nearest Neighbor" are endless, and it is in use in many computer-science areas, such as image recognition, machine learning, and computational

linguistics (1, 2 and more).



Amongst the endless use-cases are Netflix's recommendation, Spotify's recommendation, Pinterest's visual search, and many more amazing products. Photo by Alexander Shatov on Unsplash

In order to calculate exact nearest neighbors, the following techniques exist:

- **Exhaustive search**- Comparing each point to *every* other point, which will require Linear query time (the size of the dataset).
- **The Grid Trick**- Subdividing the space into a Grid, which will require exponential space/time (in the dimensionality of the dataset).
  Since we are speaking on high dimension datasets this is impractical.

Exhaustive Search Usage

I am gonna show **how to find similar vectors and will use the movielens dataset to do so** (which contain 100k rows), by using an enriched version of the dataset (which already consists of movie labels and their semantic representation). The entire code for this article can be found as a Jupyter Notebook here.

First, we going to load our dataset which already consists of movie labels and their semantic representation which is calculated here.

```
import pickle
import faiss

def load_data():
    with open('movies.pickle', 'rb') as f:
        data = pickle.load(f)
    return datadata = load_data()
data
```

```
Out[3]: {'name': array(['Toy Story (1995)', 'GoldenEye (1995)', 'Four Rooms (1995)', ...,
                'Sliding Doors (1998)', 'You So Crazy (1994)',
                'Scream of Stone (Schrei aus Stein) (1991)'], dtype=object),
         'vector': array([[-0.01780608, -0.14265831,  0.10308606, ...,  0.09659795,
                 -0.17529577, -0.03061521],
                [-0.03357764,  0.16418771,  0.21801303, ...,  0.16502103,
                 -0.09166156,  0.05047869],
                [-0.2761452 , -0.01991325, -0.04969981, ...,  0.0258275 ,
                 -0.08328608, -0.0152858 ],
                ...,
                [ 0.05142734, -0.01683608, -0.20441587, ...,  0.00045828,
                  0.14679626,  0.2462584 ],
                [ 0.04491899, -0.02819411, -0.09472758, ..., -0.02152078,
                  0.16223577,  0.19897607],
                [ 0.02531924,  0.03099714,  0.06437534, ..., -0.07260127,
                  0.0467432 ,  0.07893164]], dtype=float32)}
```

As we can see data is actually a dictionary, the name column consists of the movies' names, and the vector column consists of the movies vector representation.

I am going to show how to do an exhaustive search using faiss. We first going to create the index class.

```
class ExactIndex():
    def __init__(self, vectors, labels):
        self.dimension = vectors.shape[1]
```

```
        self.vectors = vectors.astype('float32')
        self.labels = labels

    def build(self):
        self.index = faiss.IndexFlatL2(self.dimension,)
        self.index.add(self.vectors)

    def query(self, vectors, k=10):
        distances, indices = self.index.search(vectors, k)
        # I expect only query on one vector thus the slice
        return [self.labels[i] for i in indices[0]]
```

After I define the index class I can build the index with my dataset using the following snippets.

```
index = ExactIndex(data["vector"], data["name"])
index.build()
```

Now it's pretty easy to search, let's say I want to search for the movies that are most similar to "Toy Story" (its located in index number 0) I can write the following code:

```
index.query(data['vector'][0])
```

```
['Toy Story (1995)',
 'Rock, The (1996)',
 'Long Kiss Goodnight, The (1996)',
 'Mars Attacks! (1996)',
 'Twelve Monkeys (1995)',
 'Independence Day (ID4) (1996)',
 'Star Wars (1977)',
 'Broken Arrow (1996)',
 'Star Trek: First Contact (1996)',
 'Mission: Impossible (1996)']
```

And that's it, we have done exact search, we can go nap now :).

Photo by Lidya Nada on Unsplash

But It's Not All Rainbows And Unicorns:

Unfortunately, most modern-day applications have massive datasets with high dimensionality (hundreds or thousands) so linear scan will take a while. If that's not enough, often there are additional constraints such as reasonable memory consumption and/or low latency.

It's important to note that despite all recent advances on the topic, **the only available method for guaranteed retrieval of the exact nearest neighbor is exhaustive search** (due to the curse of dimensionality.)

This makes exact nearest neighbors impractical even and allows "Approximate Nearest Neighbors " (ANN) to come into the game. A similarity search can be orders of magnitude faster if we're willing to trade some accuracy.

Photo by Niklas Kickl on Unsplash

Approximate Nearest Neighbor Introduction

To give a small intuition why approximate nearest neighbors might be good enough I will give two examples:

- **Visual Search:** As a user, if I look for a bee picture I don't mind which ones I get out of these three pictures.
- **Recommendations:** As a user, I don't really mind the order of the nearest neighbors or even if I have only eight of the ten best candidates.

Approximate Nearest Neighbor techniques speed up the search by preprocessing the data into an efficient index and are often tackled using these phases:

- **Vector Transformation** — applied on vectors before they are indexed, amongst them, there is dimensionality reduction and vector rotation.
  In order to this article well structured and somewhat concise, I won't discuss this.
- **Vector Encoding** — applied on vectors in order to construct the actual index for search, amongst these, there are data structure-based techniques like Trees, LSH, and Quantization a technique to encode the vector to a much more compact form.
- **None Exhaustive Search Component** — applied on vectors in order to avoid exhaustive search, amongst these techniques there are Inverted Files and Neighborhood Graphs.

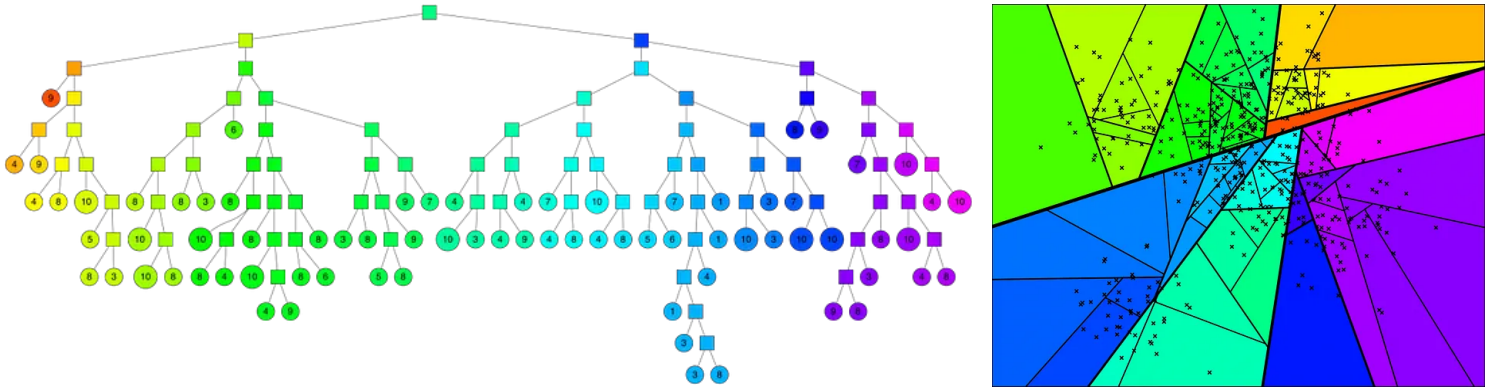
Vector Encoding Using Trees

Introduction And Intuition

Tree-based algorithms are one of the most common strategies when it comes to ANN. They construct forests (collection of trees) as their data structure by splitting the dataset into subsets.

One of the most prominent solutions out there is Annoy, which uses trees (more accurately forests) to enable Spotify' music recommendations. Since there is a comprehensive explanation I will only provide here the intuition behind it, how it should be used, the pros and the cons.

In Annoy, in order to construct the index we create a forest (aka many trees) Each tree is constructed in the following way, we pick two points at random and split the space into two by their hyperplane, we keep splitting into the subspaces recursively until the points associated with a node is small enough.



Source: https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html

In order to search the constructed index, the forest is traversed in order to obtain a set of candidate points from which the closest to the query point is returned.

Annoy Usage

We are going to create an index class like before. We are going to use annoy library. As you can imagine most of the logic is in the build method (index creation), where the accuracy-performance tradeoff is controlled by :

- **number_of_trees** — the number of binary trees we build, a larger value will give more accurate results, but larger indexes.
- **search_k** — the number of binary trees we search for each point, a larger value will give more accurate results, but will take a longer time to return.

```
class AnnoyIndex():
    def __init__(self, vectors, labels):
        self.dimension = vectors.shape[1]
        self.vectors = vectors.astype('float32')
        self.labels = labels

    def build(self, number_of_trees=5):
        self.index = annoy.AnnoyIndex(self.dimension)
        for i, vec in enumerate(self.vectors):
```

```
            self.index.add_item(i, vec.tolist())
        self.index.build(number_of_trees)

    def query(self, vector, k=10):
        indices = self.index.get_nns_by_vector(
              vector.tolist(),
              k,
              search_k=search_in_x_trees)
        return [self.labels[i] for i in indices]
```

After I define the Annoy index class I can build the index with my dataset using the following snippets.

```
index = AnnoyIndex(data["vector"], data["name"])
index.build()
```

Now it's pretty easy to search, let's say I want to search for the movies that are most similar to "Toy Story" (it's located in index number 0).

```
index.query(data['vector'][0])
```

```
['Toy Story (1995)',
 'Rock, The (1996)',
 'Independence Day (ID4) (1996)',
 'Star Wars (1977)',
 'Twelve Monkeys (1995)',
 'Star Trek: First Contact (1996)',
 'Fargo (1996)',
 'Return of the Jedi (1983)',
 'Mission: Impossible (1996)',
 'Mars Attacks! (1996)']
```

And that's it, we have search efficiently using annoy for movies similar to "Toy Story" and we got approximated results.

It's important to note, that I am going to declare Pros and Cons per implementation and not per technique.

Annoy Pros

- Decouple index creation from loading them, so you can pass around indexes as files and map them into memory quickly.
- We can tune the parameters to change the accuracy/speed tradeoff.
- It has the ability to use static files as indexes, this means you can share indexes across processes.

Annoy Cons

- The exact nearest neighbor might be across the boundary to one of the neighboring cells.
- No support for GPU processing.
- No support for batch processing, so in order to increase throughput "further hacking is required".
- Cant incrementally add points to it (annoy2 tries to fix this).

Vector Encoding Using LSH

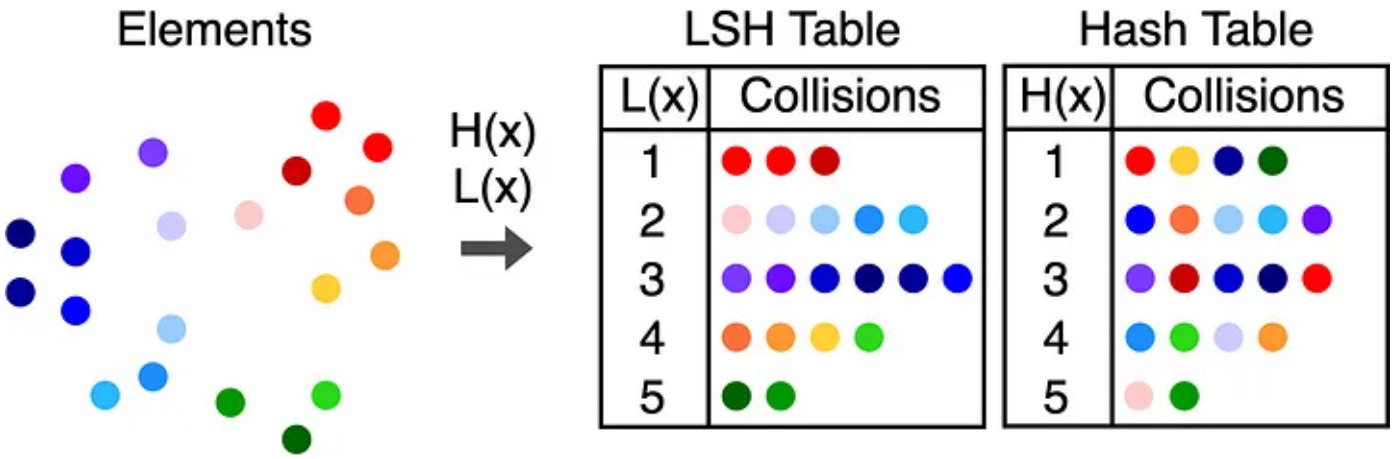
Introduction And Intuition

LSH-based algorithms are one of the most common strategies when it comes to ANN. They construct a hash table as their data structure by mapping points that are nearby into the same bucket.

One of the most prominent implementations out there is Faiss, by facebook. Since there are plenty of LSH explanations out there I will only provide here the intuition

behind it, how it should be used, the pros, and the cons.

In LSH, in order to construct the index, we apply multiple hash functions to map data points into buckets so that data points near each other are located in the same buckets with high probability, while data points far from each other are likely to fall into different buckets.



Source: https://brc7.github.io/2019/09/19/Visual-LSH.html

In order to search the constructed index, the query point is hashed in order to obtain the closest buckets (a set of candidate points) from which the closest to the query points are returned.

It's important to note there are some advancements I have not to check yet like the fly algorithm and LSH on GPU.

LSH Usage

I am going to show how to use faiss, to do "Approximate Nearest Neighbors Using LSH".
We are going to create the index class, as you can see most of the logic is in the build method (index creation), where you can control:

- **num_bits** — A larger value will give more accurate results, but larger indexes.

```
class LSHIndex():
    def __init__(self, vectors, labels):
        self.dimension = vectors.shape[1]
        self.vectors = vectors.astype('float32')
```

```
            self.labels = labels

        def build(self, num_bits=8):
            self.index = faiss.IndexLSH(self.dimension, num_bits)
            self.index.add(self.vectors)

        def query(self, vectors, k=10):
            distances, indices = self.index.search(vectors, k)
            # I expect only query on one vector thus the slice
            return [self.labels[i] for i in indices[0]]
```

After I define the LSH index class I can build the index with my dataset using the following snippets.

```
index = LSHIndex(data["vector"], data["name"])
index.build()
```

Now it's pretty easy to search, let's say I want to search for the movies that are most similar to "Toy Story" (it's located in index number 0).

```
index.query(data['vector'][0])
```

```
['Toy Story (1995)',
 'Return of the Jedi (1983)',
 'Star Wars (1977)',
 'Star Trek: First Contact (1996)',
 'Rock, The (1996)',
 'James and the Giant Peach (1996)',
 'Dead Man Walking (1995)',
 'Fifth Element, The (1997)',
 'Mars Attacks! (1996)',
 'Twister (1996)']
```

And that's it, we have search efficiently using annoy for movies similar to "Toy Story" and we got approximated results.

Like before I am going to declare Pros and Cons per implementation and not per technique.

LSH Pros

- Data characteristics such as data distribution are not needed to generate these random hash functions.
- The accuracy of the approximate search can be tuned without rebuilding the data structure.
- Good theoretical guarantees of sub-linear query time.

LSH Cons

- In practice, the algorithm MIGHT runs slower than a linear scan.
- No support for GPU processing.
- Require a lot of RAM.

Vector Encoding Using Quantization

Quantizations Motivation

Although we managed to improve query performance by constructing an index, we didn't take into account additional constraints.

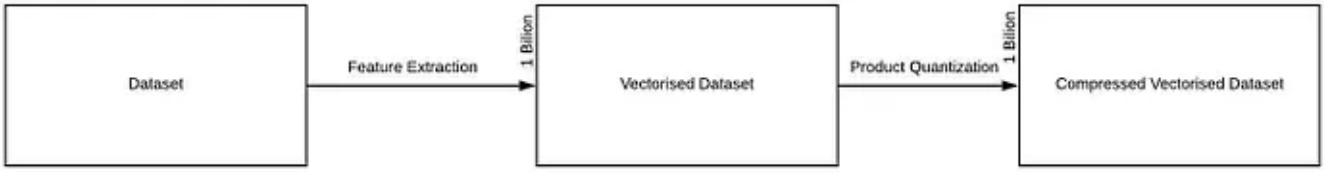
Like many engineers, we assumed that linear storage as a "no issue" (due to systems like S3). **However in practice, linear storage can become very costly very fast**, and the fact that some algorithm requires to load them to RAM and that many systems don't have separation of storage/compute definitely don't improve the situation.

Instance	RAM	# Images	Cost
m5.24xlarge	384	50,331,648	$3,363
m5.12xlarge	192	25,165,824	$1,681
m5.4xlarge	64	8,388,608	$560
m5.2xlarge	32	4,194,304	$280
m5.xlarge	16	2,097,152	$140
m5.large	8	1,048,576	$70

This image shows the price that needed to support X number of images Source: https://www.youtube.com/watch?v=AQau4-VF64w

This is why Quantization-based algorithms are one of the most common strategies when it comes to ANN.

Quantization is a technique to **reduce dataset size** (from linear) by **defining a function** (quantizer) that encodes our data into a compact approximated representation.

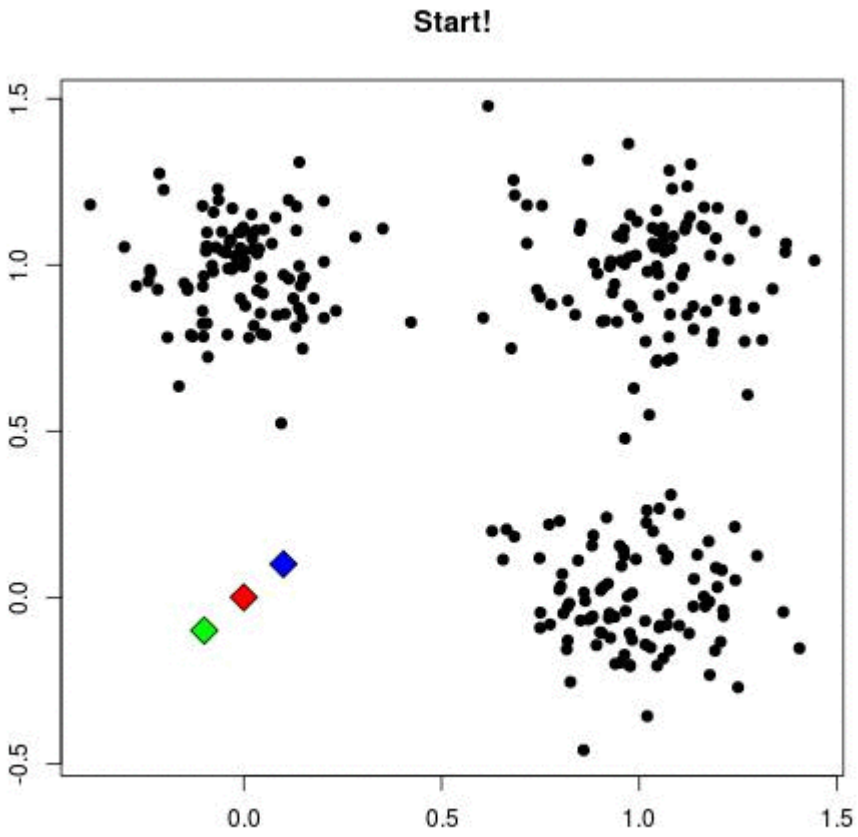


Given a dataset construct numeric vectors that represent our dataset, then compress the vectors to an approximated representation Source: https://medium.com/code-heroku/building-a-movie-recommendation-engine-in-python-using-scikit-learn-c7489d7cb145

Quantization Intuition

The intuition of this method is as follows, **we can reduce the size of the dataset by replacing every vector with a leaner approximated representation** of the vectors (using quantizer) in the encoding phase.

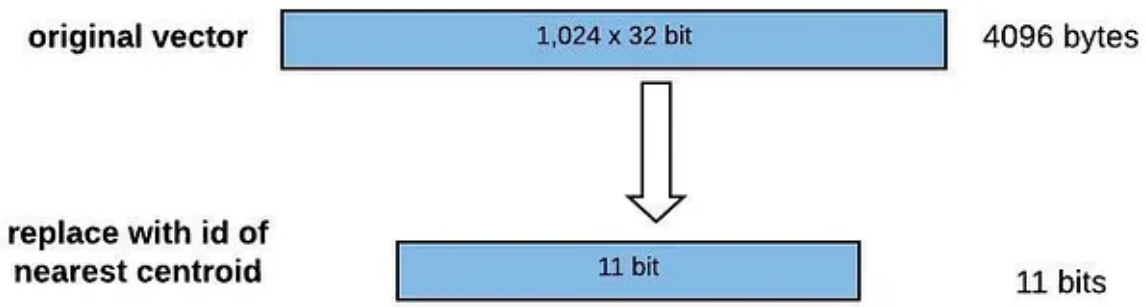
One way to achieve a leaner approximated representation is to give similar vectors to the same representation. This can be done by clustering similar vectors and represent each of those in the same manner (the centroid representation), the most popular way to do so is using k-means.



An animation demonstrating the inner workings of k-means — Courtesy: Mubaris NK

Since k-means divide the vectors in space into k clusters, each vector can be represented as one of these k centroids (the most similar one).

This will allow us to represent each vector in a much more efficient way log(k) bit per vector since each vector can be represented in the label of the centroid.



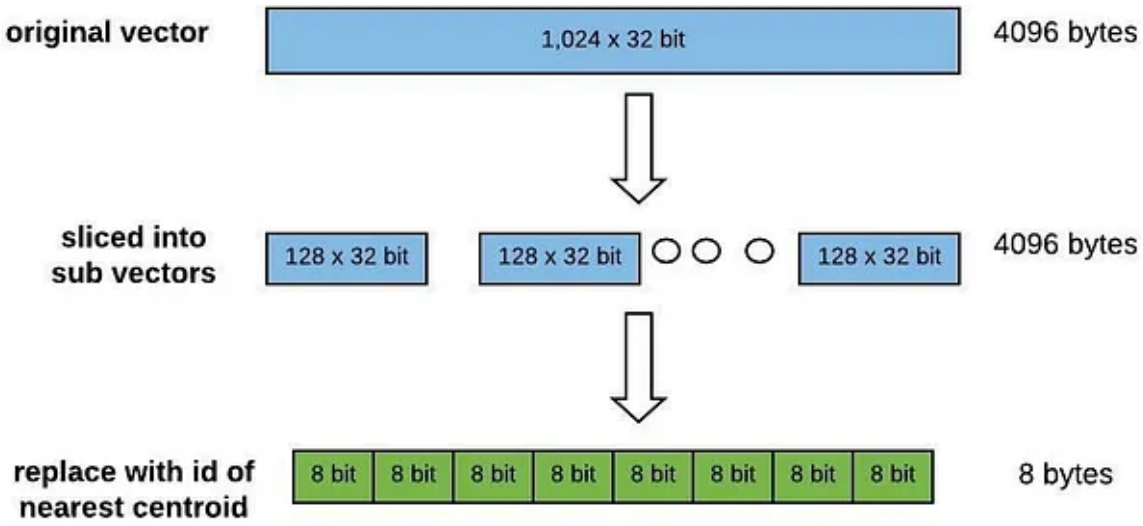
In our example, each vector is represented by one of the centroids. since we have 2042 centroids we can represent each vector with 11 bits, as opposed to 4096 (1024*32).

But, this amazing compaction comes with a great cost, **we lost accuracy as we now cant separate the original vector from the centroid.**

Product Quantization Intuition

We saw that using a quantizer like k-means comes with a price, in order to increase the accuracy of our vectors **we need to increase drastically the number of centroids, which makes the Quantization phase infeasible in practice.**

This is what gave birth to Product Quantization, we can increase drastically the number of centroids by dividing each vector into many vectors and run our quantizer on all of these and thus improves accuracy.



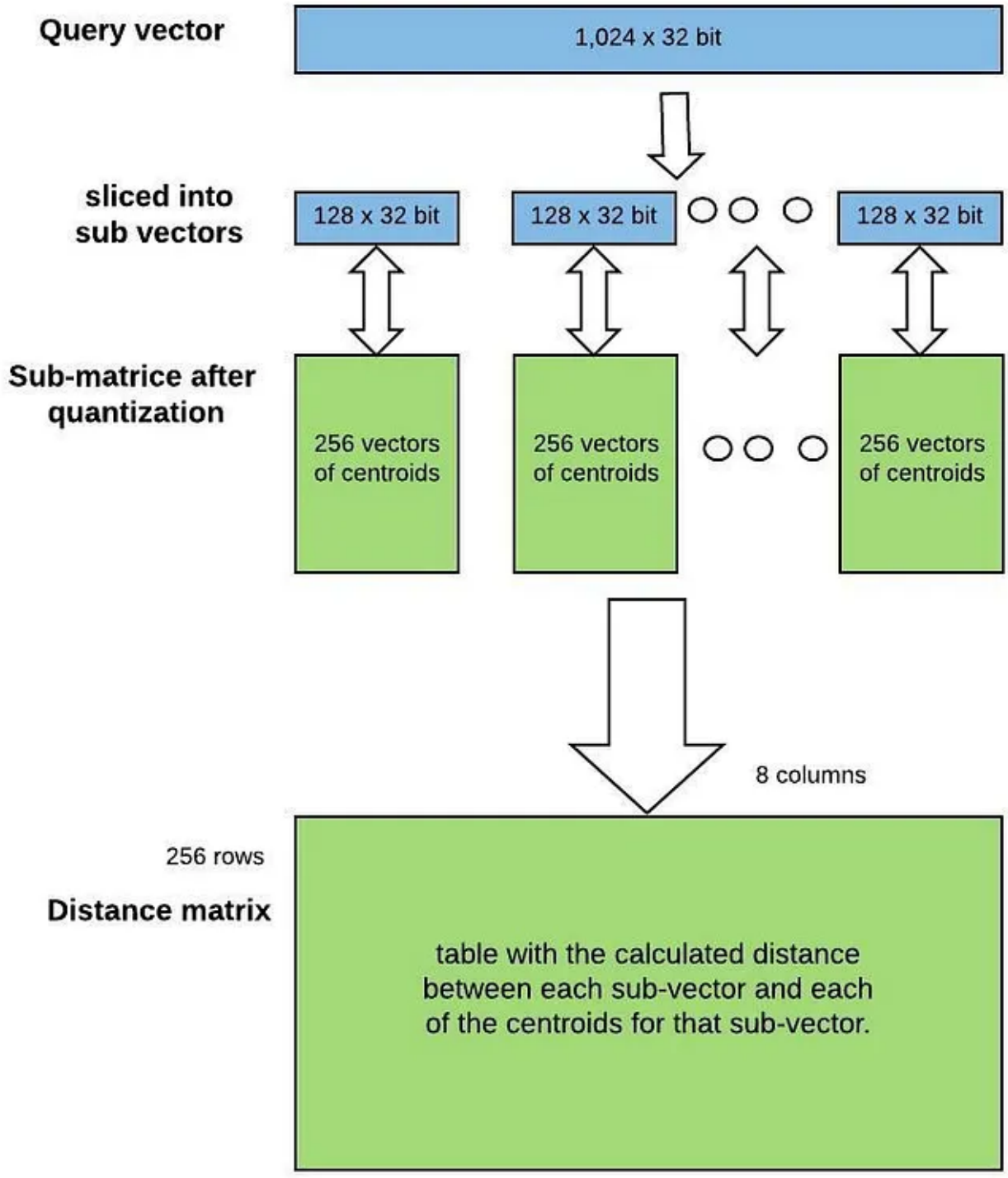
In our example, each vector is represented by 8 sub-vectors which can be represented by one of the centroids. since we have 256 centroids we can represent each matrix in 1 byte, making vector representation 8byte only as oppose to 4096 (1024*32).

Although it increases the size of the vector a bit compared to the regular quantizer, it's still O(log(k)) and allows us to increase the accuracy drastically and still work in practice.

Unfortunately in terms of search, even though we can calculate the distances in more efficiently using table look-ups and some addition. **We are still going to do an exhaustive search.**

The search is done using the following algorithm:

- Construct a table with the calculated distance between each sub-vector and each of the centroids for that sub-vector.
- Calculating approximate distance values for each of the vectors in the dataset, we just use those centroids id's to look up the partial distances in the table and sum those up!



In our example, this means that this means building a table of subvector distances with 256 rows (one for each centroid) and 8 columns (one for each subvector). Remember that each database vector is now just a sequence of 8 centroid ids.

- The exact nearest neighbor might be across the boundary to one of the neighboring cells.

Inverted File Index Intuition

The intuition of the algorithm is, that we can **avoid the exhaustive search** if we **partition our dataset** in such a way that on search, **we only query relevant partitions** (also called Voronoi cells). The reason this tends to work well in practice is since many datasets are actually multi-modal.

However, **dividing the dataset up this way reduces accuracy yet again**, because if a query vector falls on the outskirts of the closest cluster, then its nearest neighbors are likely sitting in multiple nearby clusters.

The solution to this issue is simply to **search for multiple partitions** (this is also called a probe), searching multiple nearby partitions obviously takes more time but it gives us better accuracy.

So as we saw by now **it's all about tradeoffs,** the number of partitions and the number of the partitions to be searched can be tuned to find the time/accuracy tradeoff sweet spot.

It's important to note that the Inverted File Index is a technique that can be used with other encoding strategies apart from quantization.

Encoding Residuals Intuition

The intuition of the algorithm is, we want **the same number of centroids to give has a more accurate representation**. This can be achieved if the vectors are less distinct than they were.

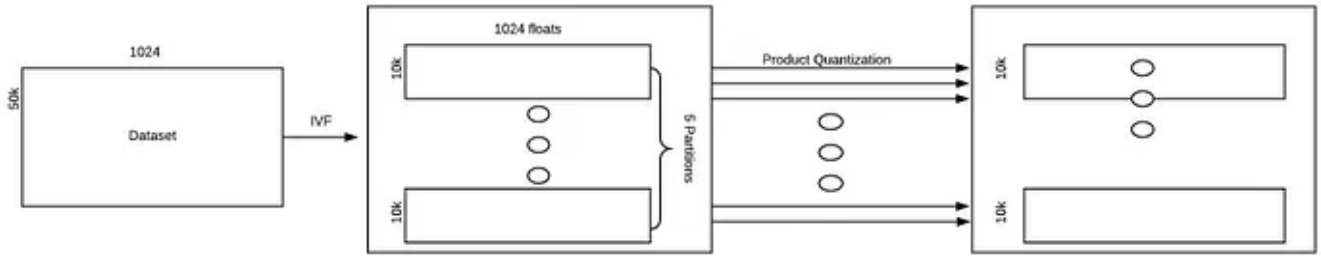
In order, to do so, for each database vector, instead of using the PQ to encode the original database vector we instead encode the vector's offset from its partition centroid.

However, **replace vectors with their offsets will increase search time yet again,** as we will need to calculate a separate distance table for each partition we probe since the query vector is different for each partition.

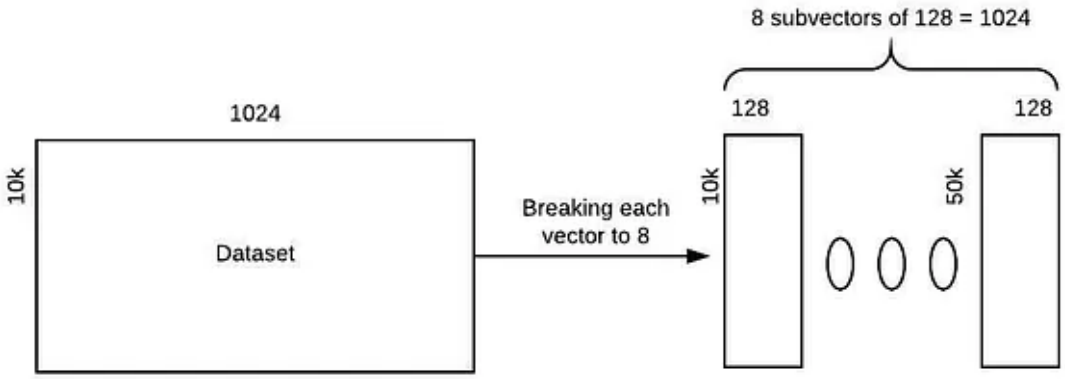
Apparently, the trade-off is worth it, though, because the IVFPQ works pretty well in practice.

Product Quantization With Inverted File Index

The algorithm goes as follows, we partition our dataset ahead of time with k-means clustering to produce a large number of dataset partitions (Inverted Index). Then, for each of the partitions, we run regular product quantization.

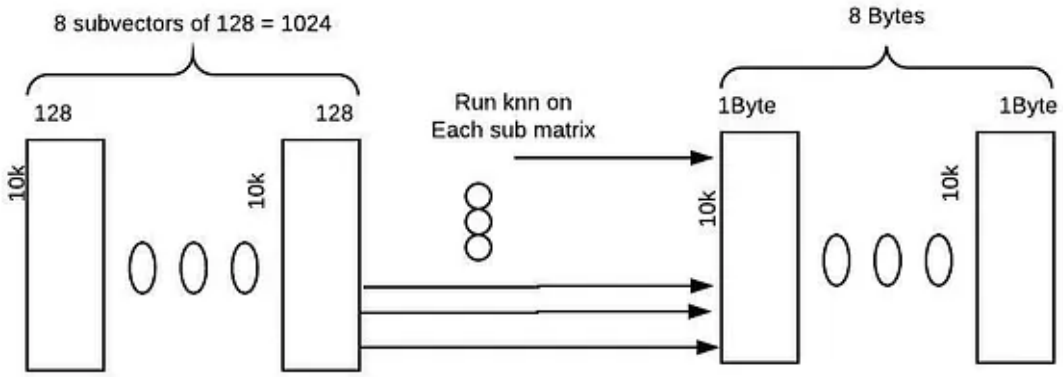


Then, for each of the partitions, we are going to break each its' vector into D/M sub-vectors, this will transform our N x D matrix to D/M matrices of N x M.



In our example, we are going to break each 1024 vector into 8 vectors of 128, thus we look at our dataset as 8 matrices of size 10k x 128.

Then we are going to run the k-means algorithm on each sub-matrix, such that each sub-vector (row) will be connected to one of the k centroids.



In our example, we are going to run k-means on our 8 matrices where k=256. This means, that each of our rows is connected to one of these 256 on each matrix (each row in our original matrix is connected to 8 centroids).

We are going to replace each sub-vector with the id of the closest matching centroid. This is what we have waited for since we have repeated elements after the previous part, we can now represent each of them with a very small label and keep the actual value only once.

If you want to get even more theory you can watch this amazing video.

Product Quantization With Inverted Index Usage

We are going to create the index class, as you can see most of the logic is in the build method (index creation), where you can control:

- **subvector_size** — the target size of the sub-vectors (product quantization phase).
- **number_of_partitions** — the numbers of partitions to divide the dataset by (Inverted File Index phase).
- **search_in_x_partitions** — the numbers of partitions to search on (Inverted File Index phase).

```
class IVPQIndex():
    def __init__(self, vectors, labels):
        self.dimension = vectors.shape[1]
        self.vectors = vectors.astype('float32')
        self.labels = labels    def build(self,
              number_of_partition=8,
              search_in_x_partitions=2,
              subvector_size=8):
        quantizer = faiss.IndexFlatL2(self.dimension)
        self.index = faiss.IndexIVFPQ(quantizer,
                                      self.dimension,
                                      number_of_partition,
                                      search_in_x_partitions,
                                      subvector_size)
        self.index.train(self.vectors)
        self.index.add(self.vectors)

    def query(self, vectors, k=10):
        distances, indices = self.index.search(vectors, k)
        # I expect only query on one vector thus the slice
        return [self.labels[i] for i in indices[0]]
```

After I define the IVPQIndex class I can build the index with my dataset using the following snippets.

```
index = IVPQIndex(data["vector"], data["name"])
index.build()
```

Now it's pretty easy to search, let's say I want to search for the movies that are most similar to "Toy Story" (its located in the 0 indexes).

```
index.query(data['vector'][0:1])
```

```
['Toy Story (1995)',
 'Rock, The (1996)',
 'Long Kiss Goodnight, The (1996)',
 'Sabrina (1995)',
 'Willy Wonka and the Chocolate Factory (1971)',
 'Broken Arrow (1996)',
 'Eraser (1996)',
 'Mars Attacks! (1996)',
 'Independence Day (ID4) (1996)',
 'Mission: Impossible (1996)']
```

And that's it, we have search efficiently using IVPQ for movies similar to "Toy Story" and we got approximated results.

Product Quantization With Inverted File Pros

- The only method with sub-linear space, great compression ratio (log(k) bits per vector.
- We can tune the parameters to change the accuracy/speed tradeoff.
- We can tune the parameters to change the space/accuracy tradeoff.
- Support batch queries.

Product Quantization With Inverted File Cons

- The exact nearest neighbor might be across the boundary to one of the neighboring cells.
- Cant incrementally add points to it.
- The exact nearest neighbor might be across the boundary to one of the neighboring cells.

Hierarchical Navigable Small World Graphs

The intuition of this method is as follows, in order to reduce the search time on a graph we would want our graph to have an average path.

This is strongly connected to the famous *"six handshake rule"* statement.

> "There is at most 6 degrees of separation between you and anyone else on Earth." — Frigyes Karinthy

Many real-world graphs on average are highly clustered and tend to have nodes that are close to each other which are formally called small-world graph:

- highly transitive (community structure) it's often hierarchical.
- small average distance ~log(N).

In order to search, we start at some entry point and iteratively traverse the graph. At each step of the traversal, the algorithm examines the distances from a query to the neighbors of a current base node and then selects as the next base node the adjacent node that minimizes the distance, while constantly keeping track of the best-discovered neighbors. The search is terminated when some stopping condition is met.

Hierarchical Navigable Small World Graphs Usage

I am going to show how to use nmslib, to do "Approximate Nearest Neighbors Using HNSW".

We are going to create the index class, as you can see most of the logic is in the build method (index creation).

```
class NMSLIBIndex():
    def __init__(self, vectors, labels):
        self.dimention = vectors.shape[1]
        self.vectors = vectors.astype('float32')
        self.labels = labels

def build(self):
        self.index = nmslib.init(method='hnsw', space='cosinesimil')
        self.index.addDataPointBatch(self.vectors)
```

```
        self.index.createIndex({'post': 2})

    def query(self, vector, k=10):
        indices = self.index.knnQuery(vector, k=k)
        return [self.labels[i] for i in indices[0]]
```

After I define the NMSLIB index class I can build the index with my dataset using the following snippets.

```
index = NMSLIBIndex(data["vector"], data["name"])
index.build()
```

Now it's pretty easy to search, let's say I want to search for the movies that are most similar to "Toy Story" (it's located in index number 0).

```
index.query(data['vector'][0])
```

```
Out[7]: ['Nightmare Before Christmas, The (1993)',
         'Beauty and the Beast (1991)',
         'Fantasia (1940)',
         'Aladdin (1992)',
         'Snow White and the Seven Dwarfs (1937)',
         'Jurassic Park (1993)',
         'Lion King, The (1994)',
         'Pink Floyd - The Wall (1982)',
         'Akira (1988)',
         'Benny & Joon (1993)']
```

And that's it, we have search efficiently using annoy for movies similar to "Toy Story" and we got approximated results.

Like before I am going to declare Pros and Cons per implementation and not per technique.

Hierarchical Navigable Small World Graphs Pros

- We can tune the parameters to change the accuracy/speed tradeoff.
- Support batch queries.
- The NSW algorithm has polylogarithmic time complexity and can outperform rival algorithms on many real-world datasets.

Hierarchical Navigable Small World Graphs Cons

- The exact nearest neighbor might be across the boundary to one of the neighboring cells.
- Cant incrementally add points to it.
- Require quite a lot of RAM.

Picking The Right Approximate Nearest Neighbours Algorithm

What Should Effect Our Decision

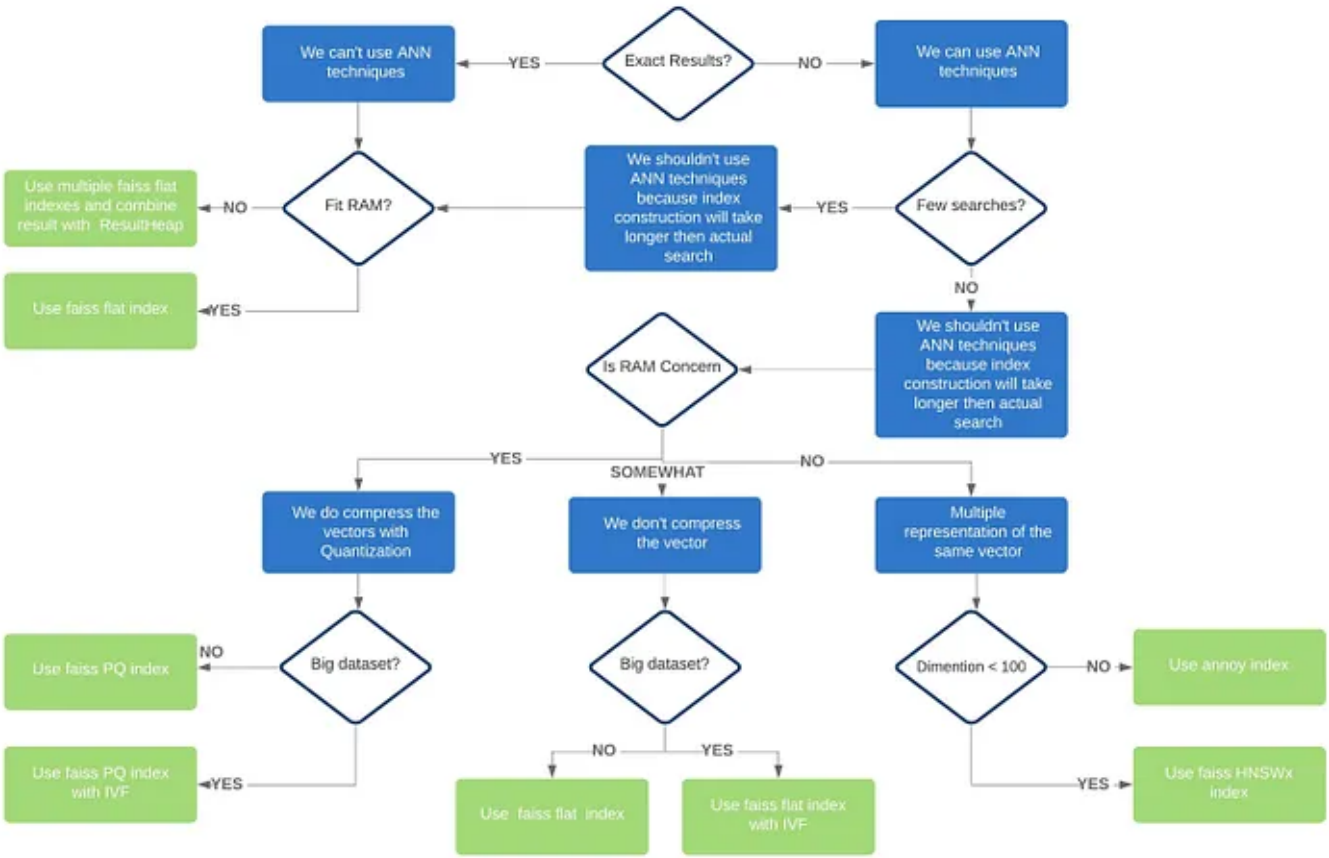
Evaluating which algorithms should be used and when is deeply depends on the use case and can be affected by these metrics:

- **Speed**- Index creation and Index construction.
- **Hardware and Resources**- Disk size, RAM size, and whether we have GPU.
- **Accuracy Requirements.**
- **Access Patterns —** Number of queries, batch or not, and whether we should update the index.

It’s important to note that there is no perfect algorithm it’s all about tradeoffs and that's what makes the subject so interesting.

How Do We Pick

I created a somewhat naive flow chart in order to allow one to choose which technique and implementation should one choose for his use-case and the metrics that we defined above.



Each implementation has its own parameters which affect either the accuracy/speed tradeoff or the space/accuracy tradeoff.

Last Words

We started this article by showing the value Nearest Neighbours algorithms provide,

Open in app ↗

Search Medium

practice, these techniques allow us to play with the storage/accuracy tradeoff and the speed/accuracy tradeoff as well.

There are many things I didn't cover like the usage of GPU by some of the algorithms, due to the extent of the topic.

I hope I was able to share my enthusiasm for this fascinating topic and that you find it useful, and as always I am open to any kind of constructive feedback.

Algorithms Performance Data Science Programming Data

Follow

Written by Eyal Trabelsi

707 Followers · Writer for Towards Data Science

Data architect at bigabid with a passion for performance, scale, python, machine learning and making software easier to use.

More from Eyal Trabelsi and Towards Data Science

Eyal Trabelsi in Towards Data Science

Style Pandas Dataframe Like a Master

What is styling and why care?

5 min read · Sep 6, 2019

1.3K 6

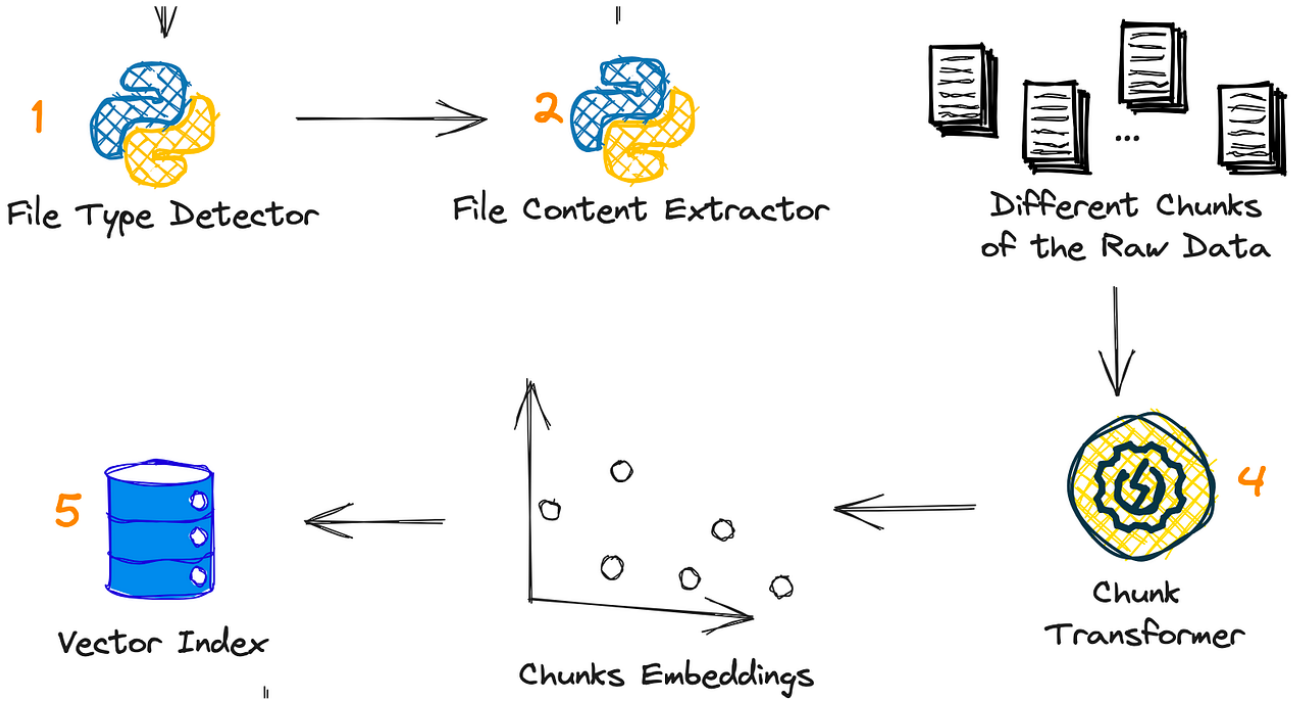
Maxime Labonne in Towards Data Science

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

A practical introduction to LLM fine-tuning

· 12 min read · Jul 25

1.8K 33



Zoumana Keita in Towards Data Science

How to Chat With Any File from PDFs to Images Using Large Language Models—With Code

Complete guide to building an AI assistant that can answer questions about any file

· 9 min read · Aug 5

909 11

Eyal Trabelsi in Towards AI

A Systematic Approach to Choosing the Best Technology/Vendor: MLOps version

The Ultimate Shopping Spree in the ML Shop

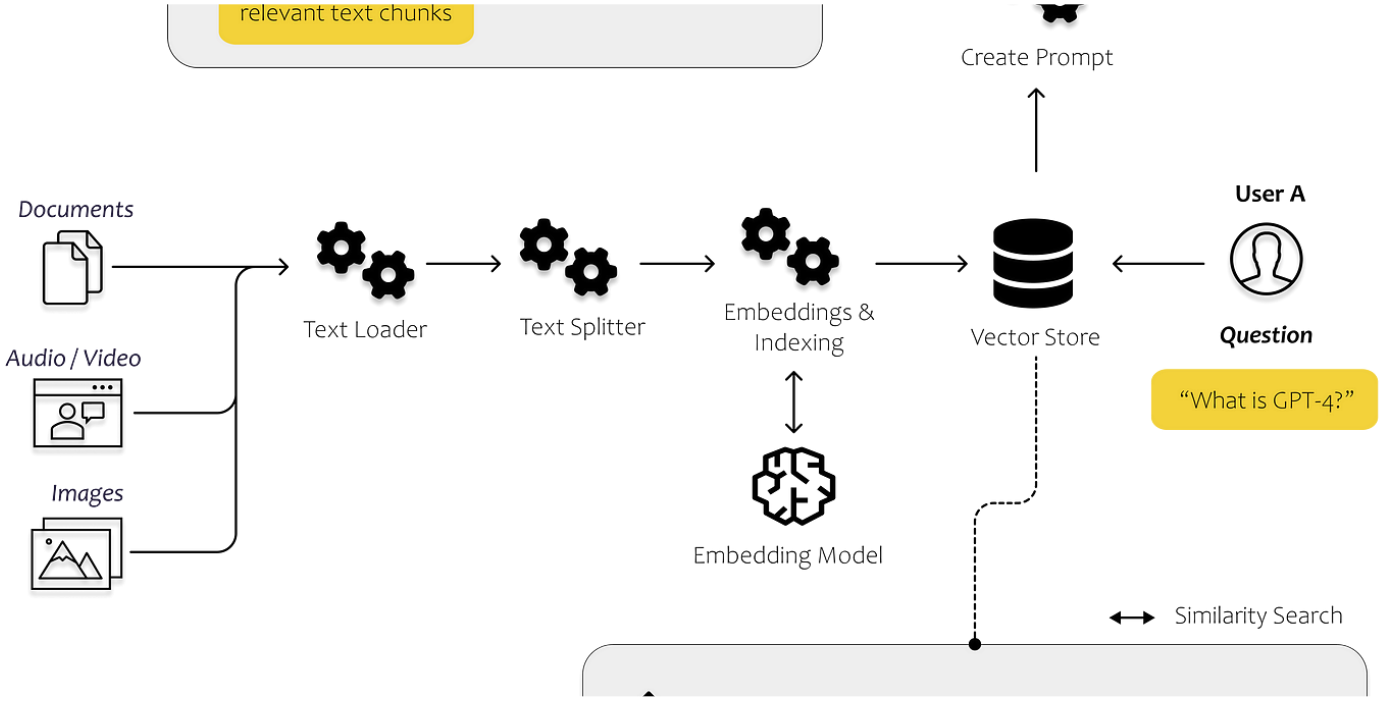
5 min read · Jul 31

77 1

See all from Eyal Trabelsi

See all from Towards Data Science

Recommended from Medium



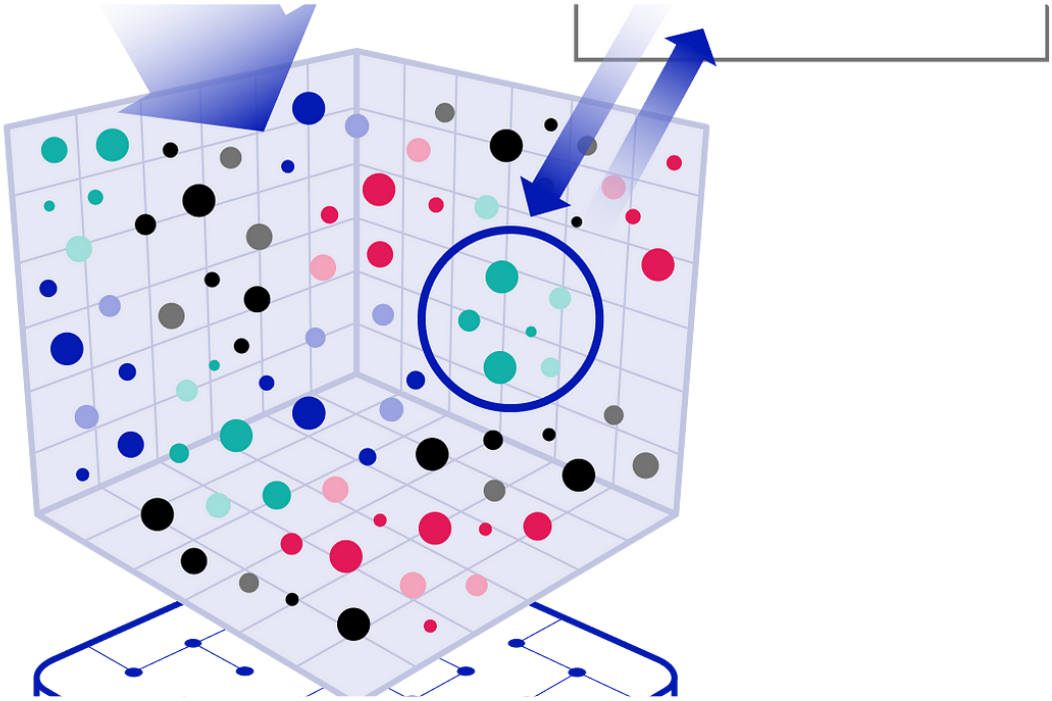
Dominik Polzer in Towards Data Science

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

4.6K 42



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...

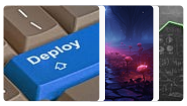
3 min read · Jun 13

57

Lists

General Coding Knowledge

20 stories · 249 saves



Predictive Modeling w/ Python

20 stories · 306 saves

New_Reading_List

174 stories · 80 saves

It's never too late or early to start something

15 stories · 91 saves

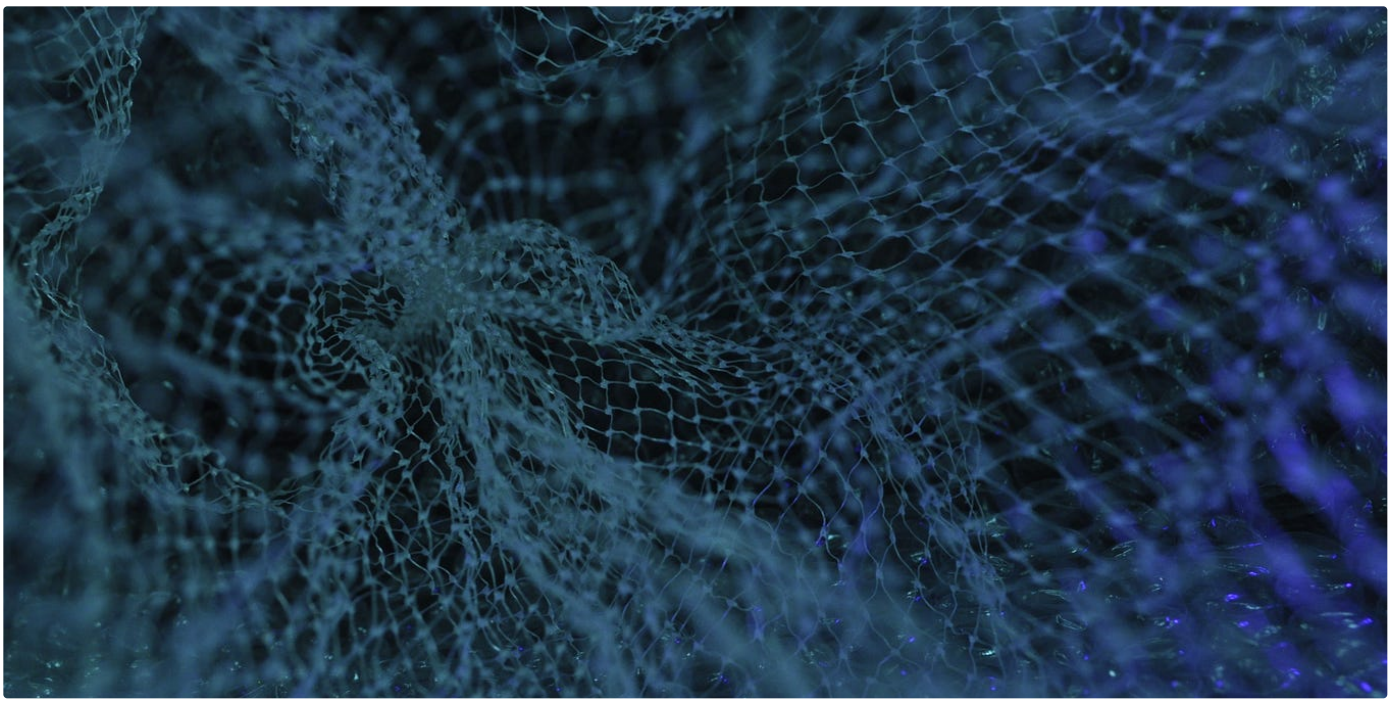
Pratyush Khare in MLearning.ai

How to perform High-Performance Search using FAISS

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

6 min read · Mar 4

128 1



dominiconorton

Optimizing Similarity Search with OpenAI's Word Embeddings for Pinecone Database

In today's data-driven world, many businesses and organizations rely on machine learning to process and analyze large amounts of data...

4 min read · Feb 28

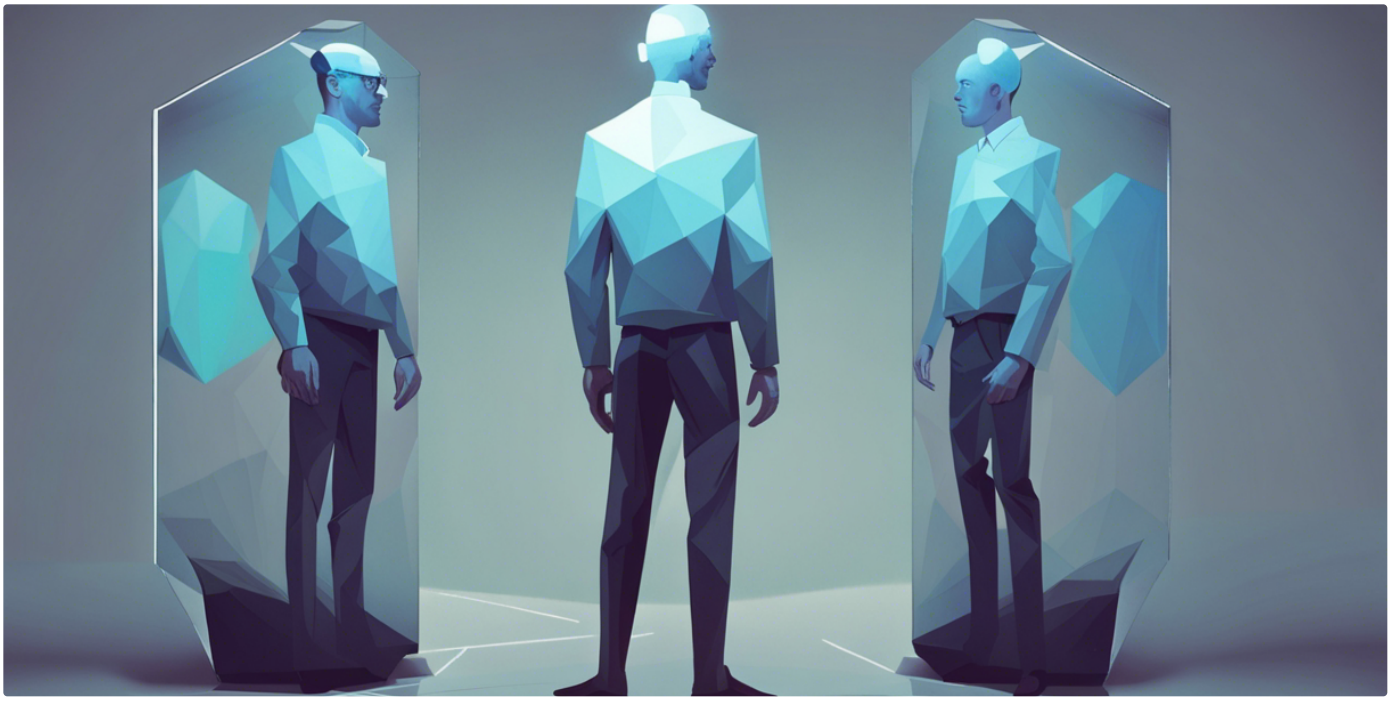
Sathishhariram

OpenAI Embedding & Semantic Search using Vector data

Exploring OpenAI Embedding API and Semantic search

7 min read · Jul 13

22 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

1.92K 14

See more recommendations