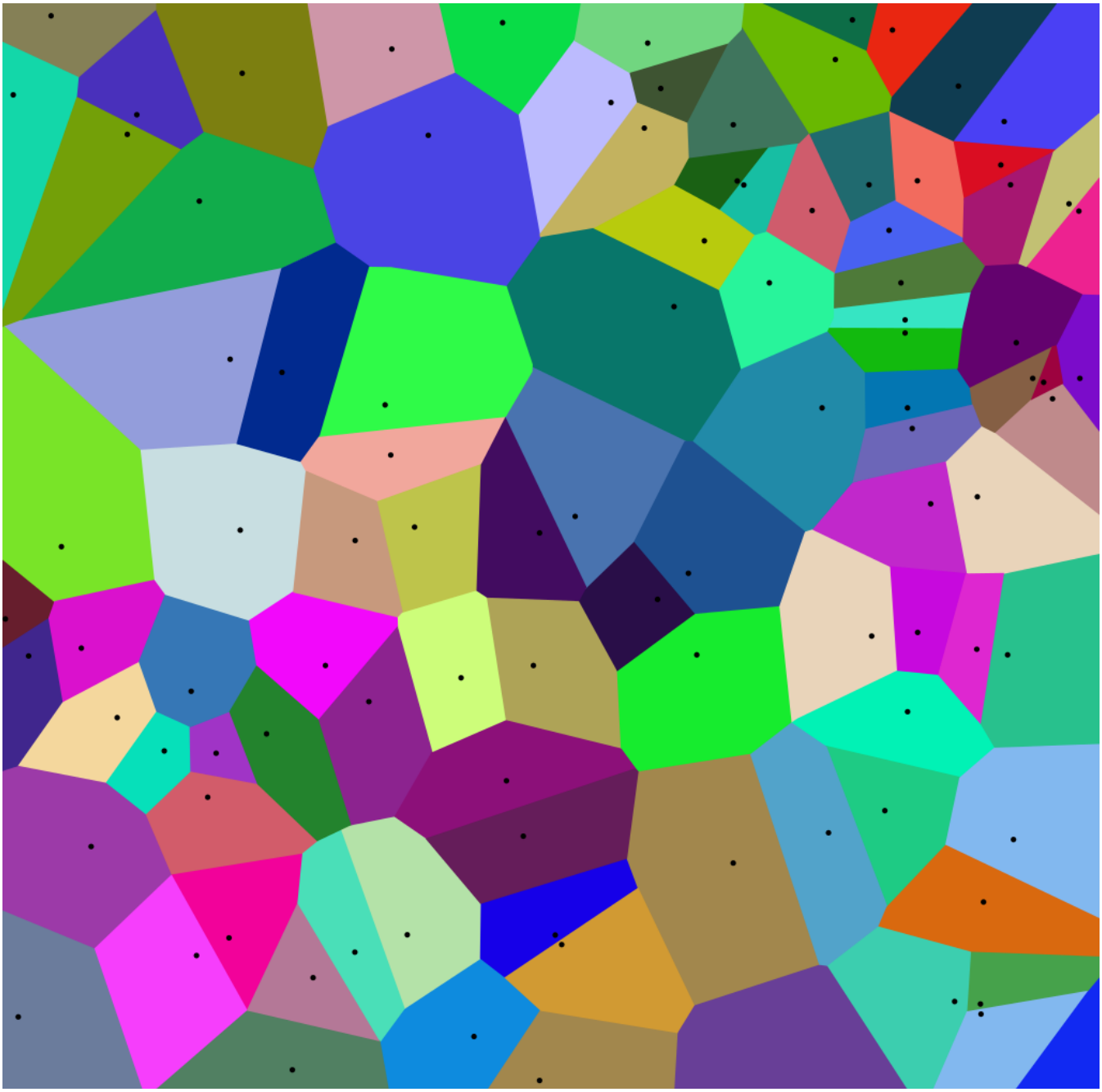
Comprehensive Guide To Approximate Nearest Neighbors Algorithms

Eyal Trabelsi

Feb 14 · 18 min read ★

Search In Practice- Approximate Nearest Neighbors

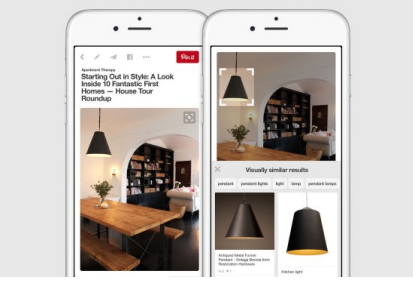


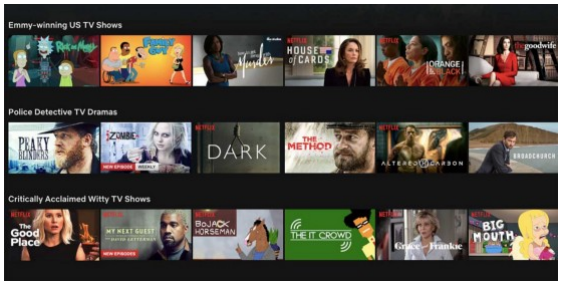
Source: https://baike.baidu.com/item/%E6%B3%B0%E6%A3%AE%E5%A4%9A%E8%BE%B9%E5%BD%A2/3428661?fromtitle=voronoi&fromid=9089406

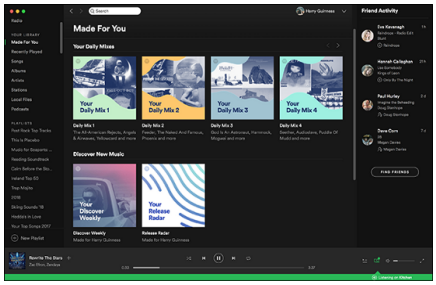
Nearest Neighbors Motivation

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.

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

- **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 to 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.

Source: https://imgflip.com/i/3nizkg

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.

Source: https://medium.com/code-heroku/building-a-movie-recommendation-engine-in-python-using-scikit-learn-c7489d7cb145

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
```

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 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 the 0 index) I can write the following code:

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


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

Source:https://www.pinterest.com/pin/52284045659543481/

. . .

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.

Source: https://unicornyard.com/unicorns-and-rainbows/

. . .

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.

Source: https://www.natgeokids.com/za/discover/animals/insects/honey-bees/

- **Visual Search:** As a user I don't really mind the order of the nearest neighbours or even if I have only eight of of the ten best candidates.

For many users these are pretty much the same. If thats not your use-case approximate solutions might not be for you :)

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

- **Vector Transformation** — applied on vector before they are indexed, amongst them there is dimensionality reduction and vector rotation.
  In order to this article well structured and somewhat concise I wont 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 .

Source: https://www.pinterest.com/pin/496310821435361577/?lp=true

. . .

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 forrest (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 in 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 are 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]
```

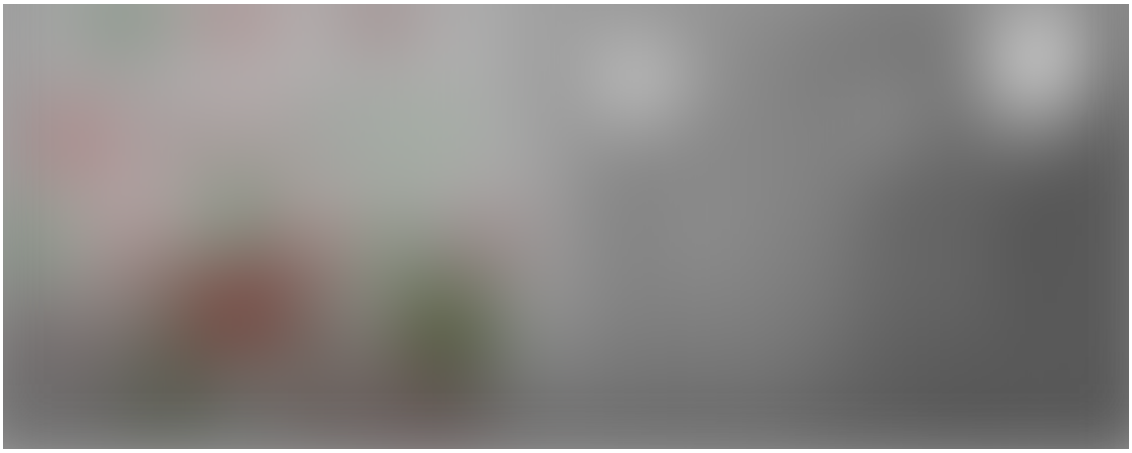
After I define the Annoy index class I can build the index with my dataset using the following snippets.

```
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" (its located in the 0 index).

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


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



Source:https://xkcd.com/835/

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 index 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 hash table as their data structure by mapping points which 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 such 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 be in 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 point are returned.

Its important to note that are some advance that i have not 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" (its located in the 0 index).

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


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

Source: https://xkcd.com/421/

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.
- Accuracy of the approximate search can be tuned without rebuilding 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.

Source: https://cheezburger.com/6338985728

. . .

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.

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 encode 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 so is using k-means.

An animation demonstrating the inner workings of k-mean, for further information one can go https://towardsdatascience.com/cluster-analysis-create-visualize-and-interpret-customer-segments-474e55d00ebb

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 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 centroid 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 neighbour might be across the boundary to one of the neighbouring 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.

We can see that the data here is multi-modal, we can split the data into three partitions without a critical hit to accuracy. Source: https://www.geeksforgeeks.org/ml-classification-vs-clustering/

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

The solution to this issue is simply to **search for multiple partitions** (this also called probe), searching multiple nearby partitions obviously take 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 Inverted File Index is a technique that can be use 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.

Source:http://mccormickml.com/2017/10/22/product-quantizer-tutorial-part-2/

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 follow, 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 to 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])
```

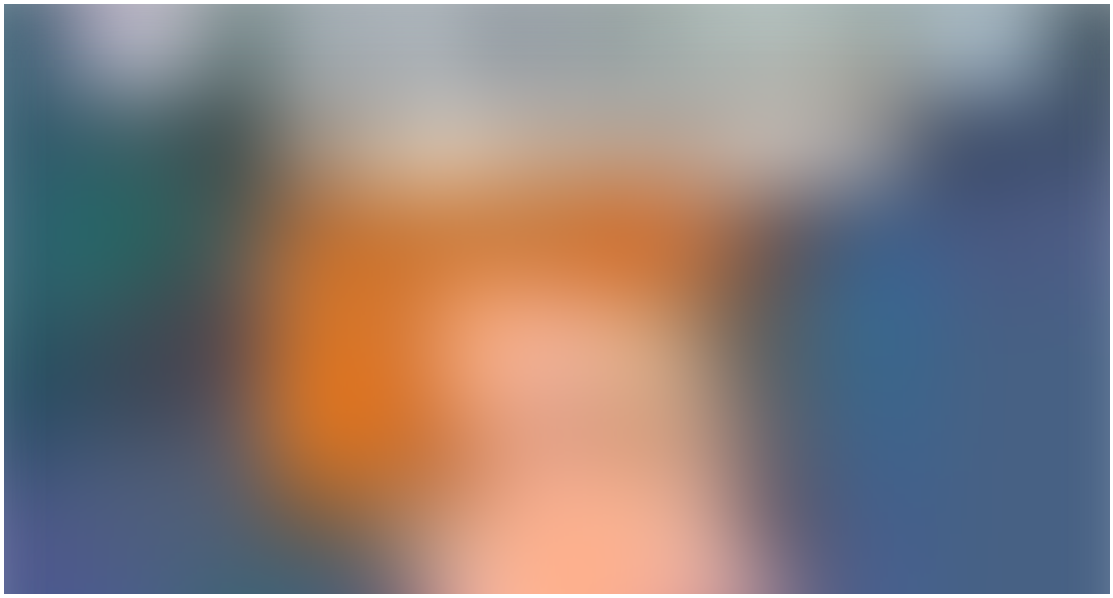
And that's it, we have search efficiently using IVPQ for movies similar to "Toy Story" and we got approximated results.

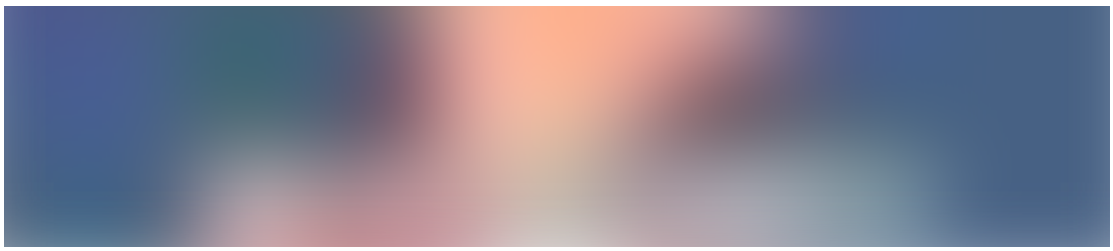
Product Quantization With Inverted File Pros

- 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.





Source: https://imgflip.com/tag/compression?sort=top-2018

. . .

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 is formally called small-world graph:

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

Source: https://medium.com/stellargraph/knowing-your-neighbours-machine-learning-on-graphs-9b7c3d0d5896

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" (its located in the 0 index).

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

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

Source: https://xkcd.com/761/

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 polylogerithmic 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.

Source:https://www.redbubble.com/people/yedayuri/works/31546381-meme-graph?p=poster

. . .

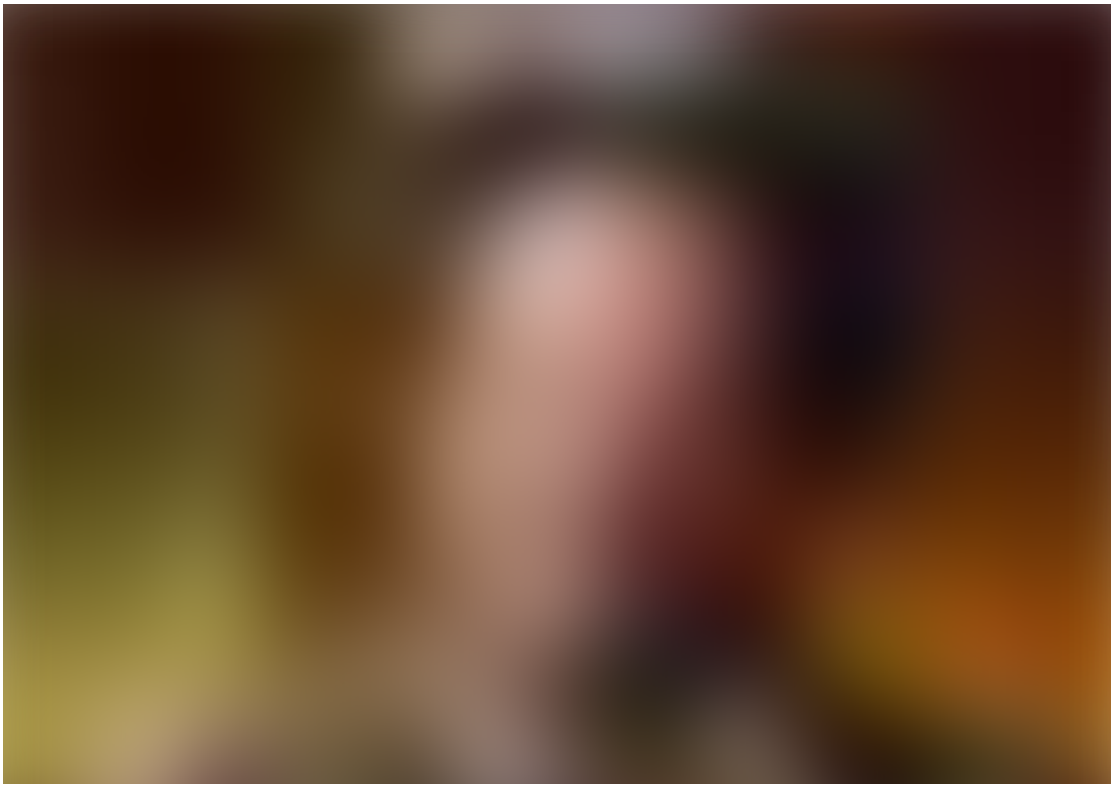
Picking The Right Approximate Nearest Neighbours Algorithm

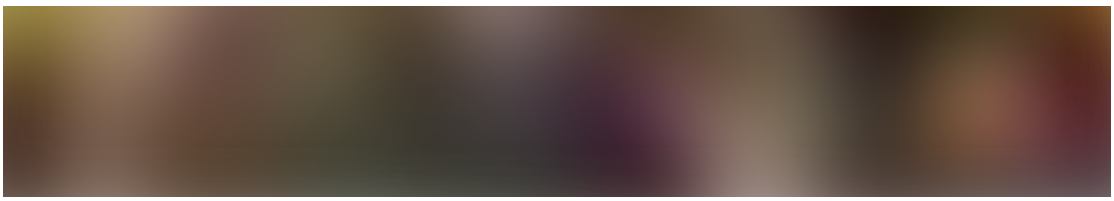
What Should Effect Our Decision

Evaluating which algorithms should be used and when is deeply depends on the use case and can be effected 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 updated the index.

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



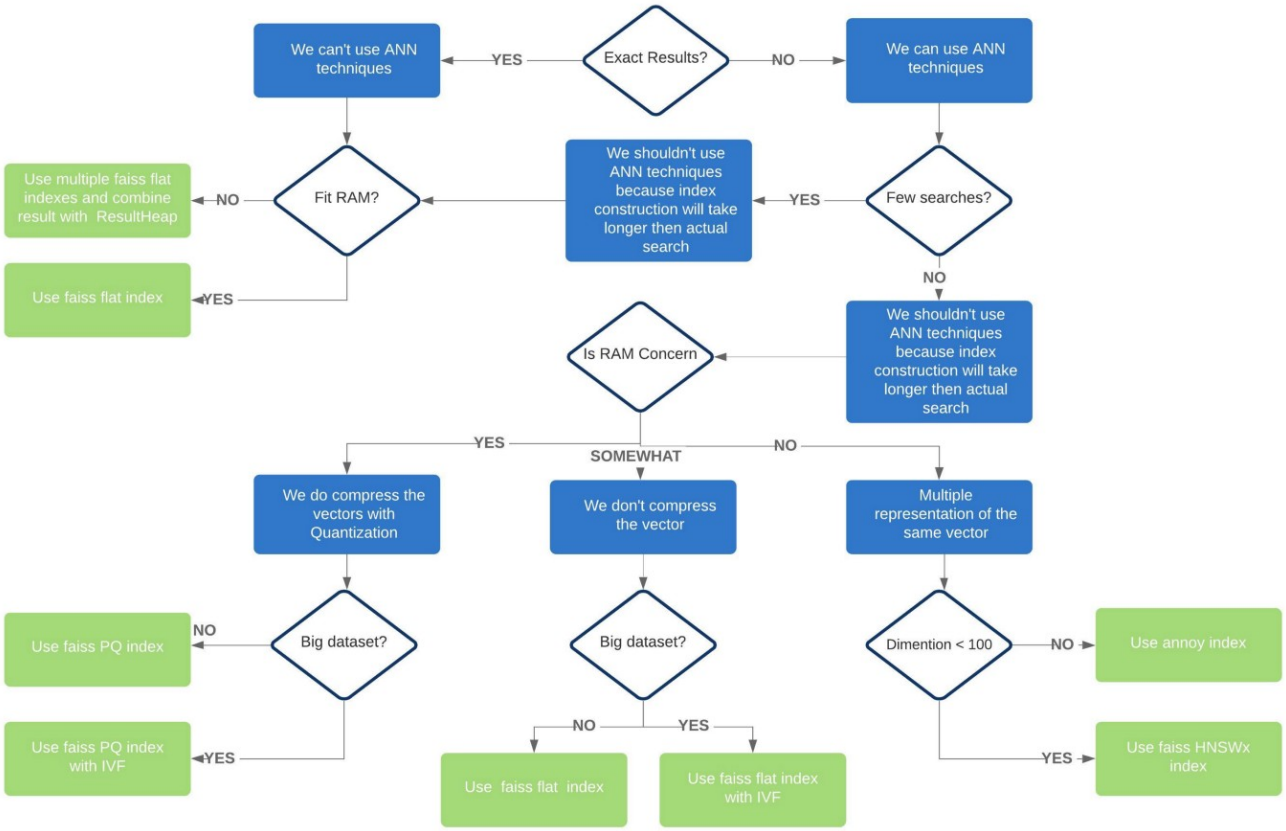


Source: https://memecrunch.com/meme/17805/metrics

. . .

How Do We Pick

I created a some what 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 it own parameters which effect either the accuracy/speed tradeoff or the space/accuracy tradeoff.

. . .

Last Words

We started this article by showing the value Nearest Neighbours algorithms provide, then I listed the problems of using these algorithms in modern apps that led to the “birth of Approximate Nearest Neighbour techniques”.

Then i explained the intuition behind the leading techniques and how to use them in practice, these techniques allows 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