Approximate Nearest Neighbors Search in Multidimensional Space

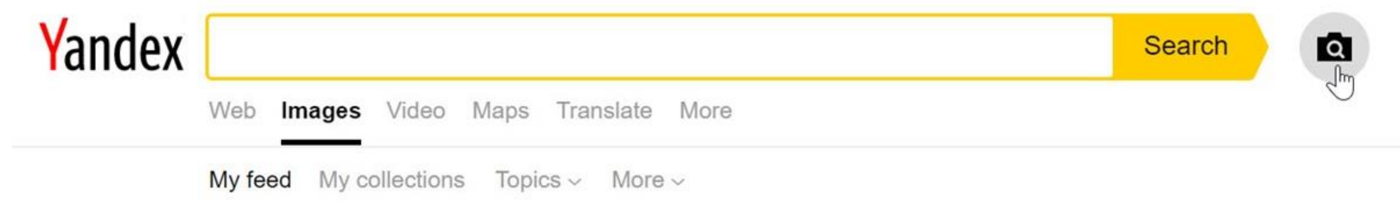
DMITRY RONZHIN

RONZHINDV@MY.MSU.RU

What’s the scope?

A well-known scenario is **exact-search task**, i.e. finding if some database contains records equal to query, or lying inside some query range. For this purpose a lot of algorithms and optimization techniques are developed, one of which is relational database (RDBMS). This can be useful for many applications, but when we start working with media data RDBMS does not always help.

There are several scenarios where, **given some database,** one needs to **find similar entries to the given request**. One of the most popular scenario is **searching for similar images** (implemented in Google, Yandex etc.).



Where’s multidimensional space?

A lot of machine-learning approaches transfer media data to some numeric multidimensional representation, with the premise that machine-learning algorithm that is used somehow preserves specific properties of the input data on vector space (i.e. preserves some metric properties between images that have similar objects).

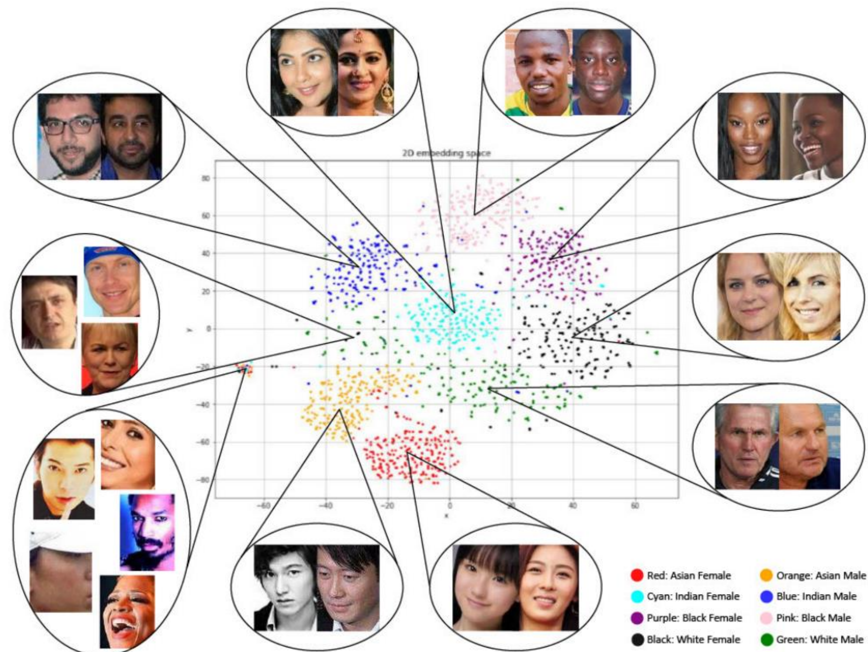


Image from https://www.mdpi.com/2813-0324/3/1/6

Embedding representation can be made in a different way, for example with neural networks or classic descriptors like SIFT, SUFR, FAST, etc. Please refer to this article for some info: https://rom1504.medium.com/image-embeddings-ed1b194d113e

Such embedding representational are usually multi-dimensional (*typically 64D,128D,256D,512D float32 for images*)

Task statement

Data:

Some (relatively) large dataset with high-dimensional vector representations for some data

Input request:

Vector representation in the same space for some input data object.

Output:

Set of similar vectors from the dataset to the requested one. By similar we can understand that they are close to the requested vector by some metric (L0,L1,L2).

Note: If one stores correspondence between vector representation of data-objects from which the dataset was made, then it is easy to retrieve similar files for the end user (i.e. find similar image by requested one).

Possible limitations

1. **Memory constraints**

When dataset is too big, even vector representation of data can have size which will not fit in memory (not only RAM, any memory). With this in mind one should need to have a sort of **dimensionality reduction** for the dataset.

2. **Time limitations (in inference)**

In many cases search through all the dataset will take too much time for end user, especially if the data is multidimensional. **There are several approaches that either minimize the response time with some extra cost on memory but without quality degradation, or vice versa.**

Note: Many of the algorithms that we discuss here have both training and inference stages. Usually training stage is not strictly limited in terms of time, although sometimes training may be too long to end in reasonable time for system. One should keep this in mind when selecting an algorithm for ANN task.

Is there open-source?

Luckily yes, out there you can find several open-source projects which can help us with the ANN task. Here are just some of them:

1. FAISS library by Facebook

Written in C++, well optimized, contains a lot of implemented algorithms, can be used for product purposes in real systems. Available at https://github.com/facebookresearch/faiss

Examples in these slides are built with the use of this library.

2. ANNOY library

C++ based, available at https://github.com/spotify/annoy

3. Neighborhood Graph and Tree (NGT) library

C++ based, available at https://github.com/yahoojapan/NGT/

Contains graph-based methods for ANN search task

4. Rayuela.jl

Julia programming language based, contains several non-orthogonal quantization methods for ANN task. Not optimized for use in product, only for research. Available at https://github.com/una-dinosauria/Rayuela.jl

What if we have not so much dimensions

Well-known data structure which could help solving such a problem is **KD-tree.**

A KD-Tree (short for *k-dimensional tree*) is a *balanced* binary tree which splits points between alternating axes. Every leaf node is a *k*-dimensional point.

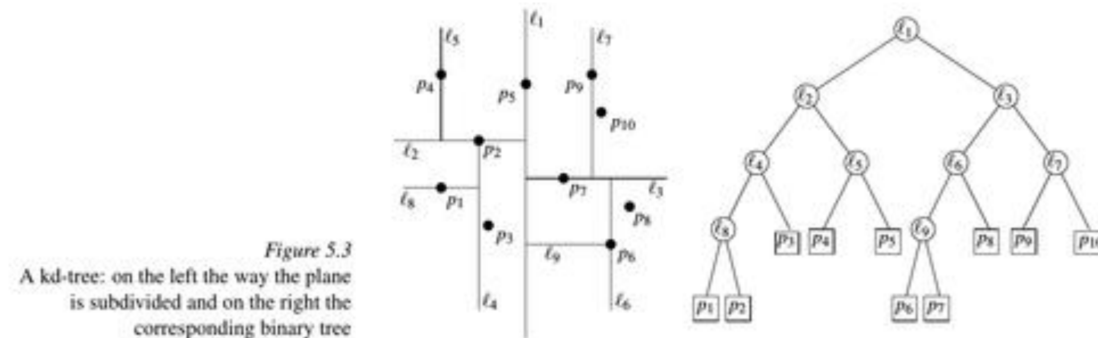


Figure 5.3
A kd-tree: on the left the way the plane is subdivided and on the right the corresponding binary tree

Training cost: $O(nlogn)$

Finding closest neighbor, insert new point, remove a point: $O(logn)$

Pros: Very good algorithm for low-dimensional data. See examples of usage in scikit learn:

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html

Cons: For high-dimensional data a lot of problems arise in balancing. Quality of search also becomes less with dimensionality increase.

For product use in low dimensions you can use FLANN or NANO FLANN

Naïve algorithm

*Scan through all vector representations. Select **K** nearest vectors to the requested one (by some metric, i.e. L0,1,2). Return result to the end user.*

***Note:** Such an approach is also known as flat index search.*

Pros:

- This is a **precise solution for the K-nearest neighbors** task. End user will receive the best answer we can get from dataset with the given vector-embeddings.

Cons:

- **Extremely low speed** on large-scale datasets (need to search through all vectors and perform a lot of arithmetic on each one)
- **Too much memory** can be required to store the whole dataset.

Example (using FAISS)

```
1  import numpy as np
2  d = 64                          # dimension
3  nb = 100000                     # database size
4  nq = 10000                      # nb of queries
5  np.random.seed(1234)            # make reproducible
6  xb = np.random.random((nb, d)).astype('float32')
7  xb[:, 0] += np.arange(nb) / 1000.
8  xq = np.random.random((nq, d)).astype('float32')
9  xq[:, 0] += np.arange(nq) / 1000.
10
11 import faiss                   # make faiss available
12 index = faiss.IndexFlatL2(d)   # build the index
13 print(index.is_trained)
14 index.add(xb)                  # add vectors to the index
15 print(index.ntotal)
16
17 k = 4                          # we want to see 4 nearest neighbors
18 D, I = index.search(xb[:5], k) # sanity check
19 print(I)
20 print(D)
21 D, I = index.search(xq, k)     # actual search
22 print(I[:5])                   # neighbors of the 5 first queries
23 print(I[-5:])                  # neighbors of the 5 last queries
```

The output of the sanity check should look like

```
[[  0 393 363  78]
 [  1 555 277 364]
 [  2 304 101  13]
 [  3 173  18 182]
 [  4 288 370 531]]

[[ 0.          7.17517328  7.2076292   7.25116253]
 [ 0.          6.32356453  6.6845808   6.79994535]
 [ 0.          5.79640865  6.39173603  7.28151226]
 [ 0.          7.27790546  7.52798653  7.66284657]
 [ 0.          6.76380348  7.29512024  7.36881447]]
```

The output of the actual search is similar to

```
[[ 381  207  210  477]
 [ 526  911  142   72]
 [ 838  527 1290  425]
 [ 196  184  164  359]
 [ 526  377  120  425]]

[[ 9900 10500  9309  9831]
 [11055 10895 10812 11321]
 [11353 11103 10164  9787]
 [10571 10664 10632  9638]
 [ 9628  9554 10036  9582]]
```

Example code and output from https://github.com/facebookresearch/faiss/wiki/Getting-started

Decreasing search time

First basic idea is to decrease search with some additional usage of **clustering**:

1. With the use of some clustering algorithm (i.e. **k-means**) select the fixed number **nlist** of centroids from the multidimensional space where our dataset relies, using some training sample from the dataset vectors.
2. "Bind" each vector from the dataset to the centroid (*i.e. using centroid number*).
3. When new request arrives we first search for some amount **nprobes** of closest centroids to the request, and then look through lists that are bound to the selected centroids, to search for **K**-nearest neighbors.

Note: *This approach has a well-known embodiment called* ***Inverted File Index****, which uses slight modifications of the same idea. So you will often refer to it as to* ***IVF Index****.*

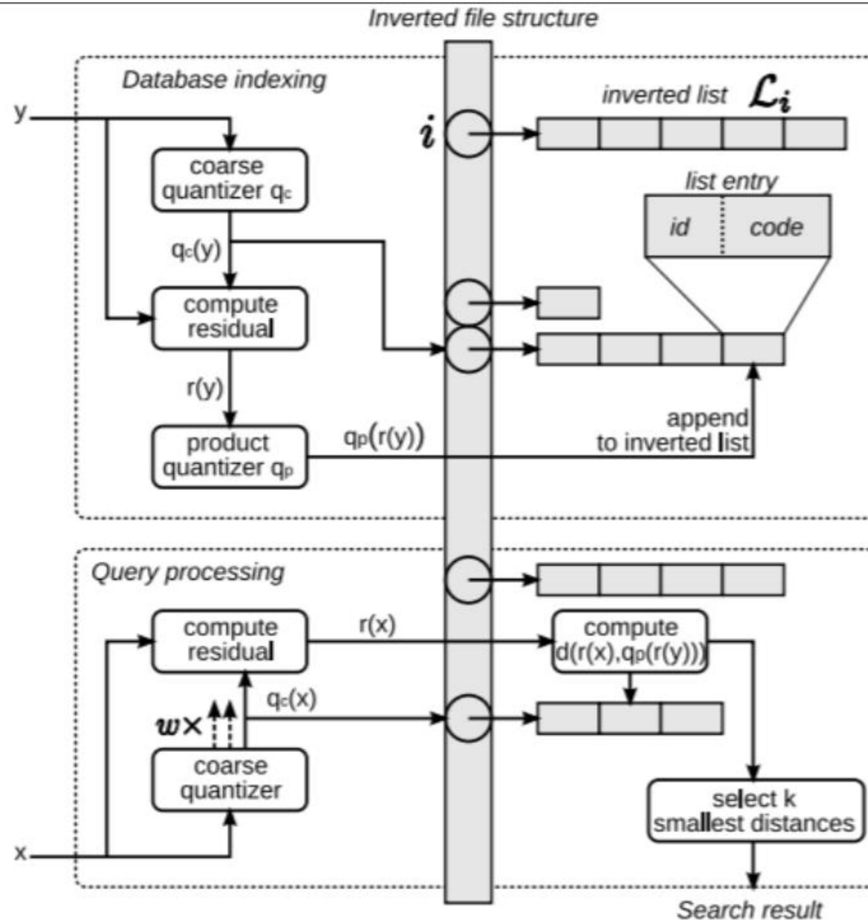
Pros:

- Speed is increased drastically in case dataset is large and parameters **nlist, nprobes** are selected properly

Cons:

- Found k-nearest vectors may not be the exact solution, compared to exhaustive search
- We have some additional memory consumption for IVF indexing (though, rather small)

IVF Index Visualized



The big difference that you can find in here, compared to initial idea, is using **residuals**. Residual is simply a difference between the centroid vector and vector from the dataset.

You can also notice so-called product quantizer in this scheme, right after residual quantization. We will discuss it later, for now you can think that this block is doing nothing at all.

Why bother with residuals?
In many cases residual vectors for real-life datasets have better properties for further clustering and compression, which allows to stack additional algorithms on top of IVF.

Example (using FAISS)

```
nlist = 100
k = 4
quantizer = faiss.IndexFlatL2(d)  # the other index
index = faiss.IndexIVFFlat(quantizer, d, nlist)
assert not index.is_trained
index.train(xb)
assert index.is_trained

index.add(xb)                   # add may be a bit slower as well
D, I = index.search(xq, k)      # actual search
print(I[-5:])                   # neighbors of the 5 last queries
index.nprobe = 10               # default nprobe is 1, try a few more
D, I = index.search(xq, k)
print(I[-5:])                   # neighbors of the 5 last queries
```

Results

For nprobe=1, the result looks like

```
[[ 9900 10500  9831 10808]
 [11055 10812 11321 10260]
 [11353 10164 10719 11013]
 [10571 10203 10793 10952]
 [ 9582 10304  9622  9229]]
```

Increasing nprobe to 10 does exactly this:

```
[[ 9900 10500  9309  9831]
 [11055 10895 10812 11321]
 [11353 11103 10164  9787]
 [10571 10664 10632  9638]
 [ 9628  9554 10036  9582]]
```

Example code and output from https://github.com/facebookresearch/faiss/wiki/Faster-search

Dimensionality Reduction

The main idea behind many (if not all) dimensionality reduction approach is to use some premise about data distribution, which allows to make a coordinate system less than initial (with possible minor data loss) and save both memory and computation costs when working with data.

Some classical dimensionality reduction methods are:

- PCA (Principle Component Analysis) – searches for subspace inside initial vector space to project data on, with the limitation that the new coordinate system for subspace will have maximum dispersion among initial data. Some of the variations of PCA:
  - Randomized PCA
  - Kernel PCA
  - Incremental PCA (*if your dataset is too large*)
- Manifold learning – this approaches try to predict the manifold inside initial vector space, where your data lies on:
  - Locally Linear Embedding (LLE)
  - Isomap

Dimensionality Reduction

For classical dimensionality reduction methods

Pros:

1. Many standard implementations
2. Well studied and work good on large scope of real-world data

Cons:

1. Fast methods like PCA may (*and will*) lead to quality reduction.
2. Slow methods may never finish learning.

Note: Please refer to this tutorial if you want to know more about dimensionality reduction techniques:

https://github.com/ageron/handson-ml2/blob/master/08_dimensionality_reduction.ipynb

Example (using FAISS)

Computing a PCA

Let's reduce 40D vectors to 10D.

```
# random training data
mt = np.random.rand(1000, 40).astype('float32')
mat = faiss.PCAMatrix (40, 10)
mat.train(mt)
assert mat.is_trained
tr = mat.apply(mt)
```

Example code from https://github.com/facebookresearch/faiss/wiki/Faiss-building-blocks:-clustering,-PCA,-quantization

Dimensionality Reduction

Hashing-based approach is known as Locality Sensitive Hash (LSH).

The main idea is to build a family of hash function for the dataset, which will partially represent metric of the initial data-space on the hashed-space, i.e. they will respect close vectors (so they remain close in new space), but may be wrong on distanced vectors.

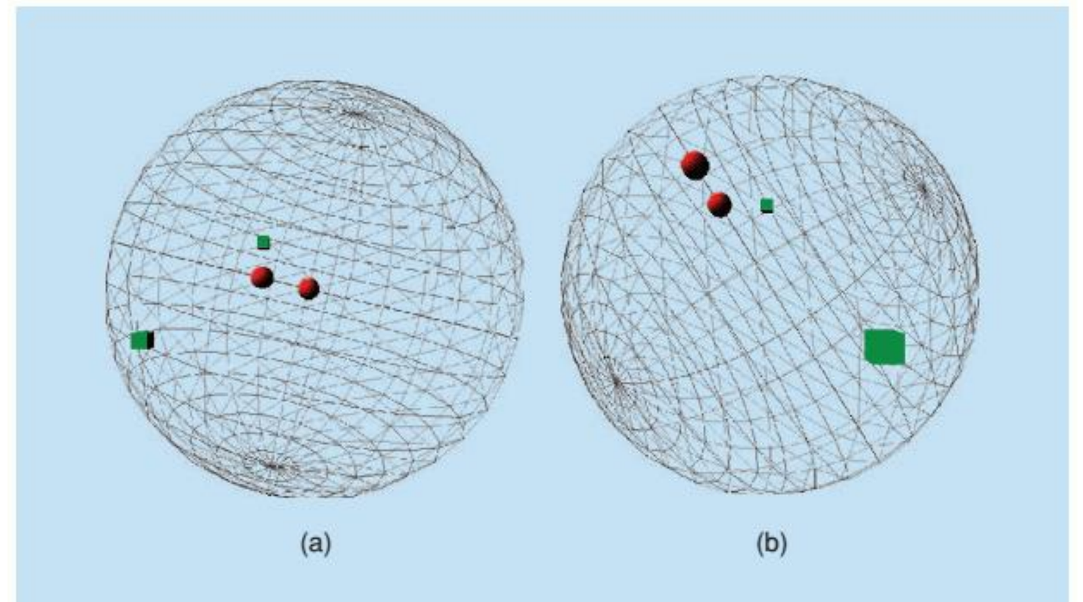
***Note**: LSH is available in FAISS as IndexLSH class*

Pros:

- Good mathematical model
- High speed

Cons:

- Less quality in real life, compared to other algorithms



[FIG1] Two examples showing projections of two close (circles) and two distant (squares) points onto the printed page.

Image from the short tutorial https://web.iitd.ac.in/~sumeet/Slaney2008-LSHTutorial.pdf

Dimensionality Reduction

One of the most used in practice is the approach of **vector quantization**.

Idea is the following: If only there was enough centroids for our datasets and we could find a way to store these centroids in a compact representation, then searching would be faster and memory consumption would decrease. Simple **K-means** algorithm cannot afford too many centroids, but **we can represent each dataset vector as an element from the linear combinations of vectors from some sets of centroids, each of which is selected on some data subspace**.

There are two possibilities:

1. If the sets of centroids are orthogonal, then we simple have a Cartesian product of these sets, which form a new set for centroids on the whole dataset. Such approaches are called **orthogonal**.
2. If we do not require sets to be orthogonal, we have to make more computations and store more data, but results can be more precise. Such approaches are called **non-orthogonal**.

Orthogonal Vector Quantization Illustrated

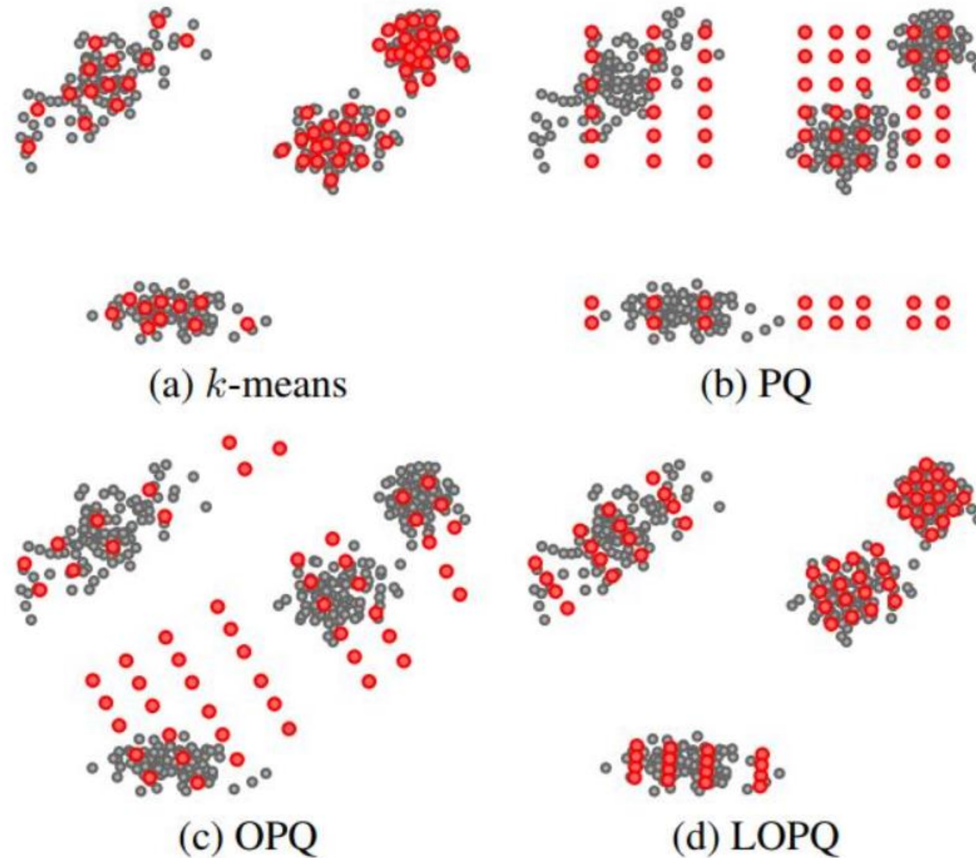


Image from the article http://image.ntua.gr/iva/files/lopq.pdf

Orthogonal Methods - PQ

Product Quantization (PQ) is one of the most used orthogonal method for vector quantization.

Given some vector in multidimensional space we just split it into parts (projections), and for each projection we work with simple algorithms – for example, to train centroids we use simple **K-means** and select the fixed number of centroids for each projection. Since they are orthogonal, each new vector is presented as an element of Cartesian product of the sets of centroids for each projection. If number of centroids in each projection is not too much, this allows to decrease space for vector representation *(i.e. one byte for 256 centroids, instead of several float32 coordinates)*:

$$\mathcal{C} = \mathcal{C}_1 \times \ldots \times \mathcal{C}_m,$$

$$\underbrace{x_1, \ldots, x_{D^*}}_{u_1(x)}, \ldots, \underbrace{x_{D-D^*+1}, \ldots, x_D}_{u_m(x)} \rightarrow q_1\big(u_1(x)\big), \ldots, q_m\big(u_m(x)\big),$$

To compute distances for request we use saved codebooks with the centroid coordinates to quantize the request and make all arithmetic with centroid vectors first.

Product Quantization

Pros:

- Helps to beat the curse of dimensionality, allows to create enough centroids to represent dataset
- Very high speed to process the input request for search (*since we can just calculate distances to codebooks first, and then sum everything up, thanks to orthogonality*)
- High data compression is possible
- Good combination with IVF Index (*use instead of flat index on the inverted-list level*)

Cons:

- The grid of codebooks is rather solid, and it can have a lot of code words which represent too little real vectors (*and some will contain too much, bad balancing of lists*).

Please refer to original article for more details:

https://lear.inrialpes.fr/pubs/2011/JDS11/jegou_searching_with_quantization.pdf

Example (from FAISS)

```python
import numpy as np
d = 64                           # dimension
nb = 100000                      # database size
nq = 10000                       # nb of queries
np.random.seed(1234)             # make reproducible
xb = np.random.random((nb, d)).astype('float32')
xb[:, 0] += np.arange(nb) / 1000.
xq = np.random.random((nq, d)).astype('float32')
xq[:, 0] += np.arange(nq) / 1000.

import faiss
nlist = 100
m = 8                             # number of subquantizers
k = 4
quantizer = faiss.IndexFlatL2(d)  # this remains the same
index = faiss.IndexIVFPQ(quantizer, d, nlist, m, 8)
                                    # 8 specifies that each sub-vector is encoded as 8 bits
index.train(xb)
index.add(xb)
D, I = index.search(xb[:5], k) # sanity check
print(I)
print(D)
index.nprobe = 10              # make comparable with experiment above
D, I = index.search(xq, k)     # search
print(I[-5:])
```

Results

The results look like:

```
[[   0  608  220  228]
 [   1 1063  277  617]
 [   2   46  114  304]
 [   3  791  527  316]
 [   4  159  288  393]]

[[ 1.40704751  6.19361687  6.34912491  6.35771513]
 [ 1.49901485  5.66632462  5.94188499  6.29570007]
 [ 1.63260388  6.04126883  6.18447495  6.26815748]
 [ 1.5356375   6.33165455  6.64519501  6.86594009]
 [ 1.46203303  6.5022912   6.62621975  6.63154221]]
```

PQ modifications

- Optimized Product Quantization (OPQ) – adds rotation matrix, which allows to rotate the grid of centroids and better fit dataset. Rotation matrix is learned once in training, and then has a tiny overhead in inference stage compared to PQ. Please refer to this article for more details:

https://www.cv-foundation.org/openaccess/content_cvpr_2013/papers/Ge_Optimized_Product_Quantization_2013_CVPR_paper.pdf

- Locally Optimized Product Quantization (LOPQ) – adds some set of rotation and shift matrices, which makes the grid fit even more. This approach requires much more time to train than OPQ, and has more overhead in inference, but allows for better search quality. Please refer to this article for more details:

http://image.ntua.gr/iva/files/lopq.pdf

PQ optimizations

- Derived codebooks – adds hierarchy of quantizers that allows for faster search on large scale datasets. Uses bit representation of codebooks to make it more efficient. For more details please refer here: https://arxiv.org/pdf/1905.06900.pdf

- Polysemous codes – uses some Hamming-distance tricks to speed up search and sacrifice a bit of quality, which is useful for large scale datasets. Can be used in FAISS (*index.search_type = faiss.IndexPQ.ST_polysemous*). Please refer here for details: https://arxiv.org/pdf/1609.01882.pdf

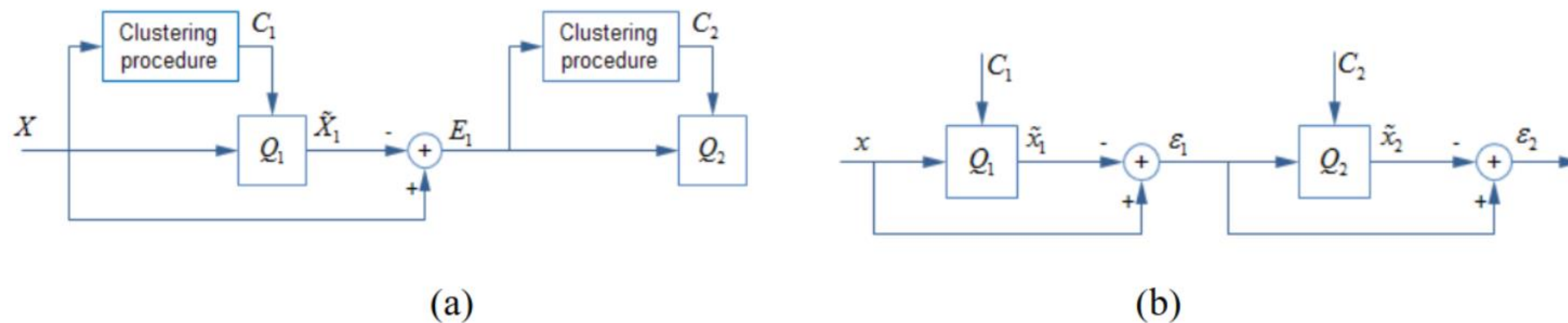
- PQ Fast Scan (v1 and v2) – cache-optimized PQ implementations. Can be used in FAISS now (IndexPQFastScan). Please refer to this article for more details:

http://www.vldb.org/pvldb/vol9/p288-andre.pdf

Non-orthogonal vector quantization

Does not use the premise of orthogonality of codebooks, which leads to heavier computations and more storage overhead, but allows better quality of search. Some of the methods are available in FAISS (please refer to this page with comparison and detailed information: https://github.com/facebookresearch/faiss/wiki/Additive-quantizers)

Example of non-orthogonal method pipeline (Residual Vector Quantization (RVQ)):



Graph-based Methods

Interesting family of approaches for ANN search is a graph-based algorithms.

They are mostly interesting in a use-case when you do not have strict limitations on memory (i.e. dataset is not too large and can possibly fit in RAM), but you have quite strict limitations on speed of search.

They can also server as a first-level (coarse) quantizer in IVF structure (instead of simple K-means) and possibly greatly increase overall search quality.

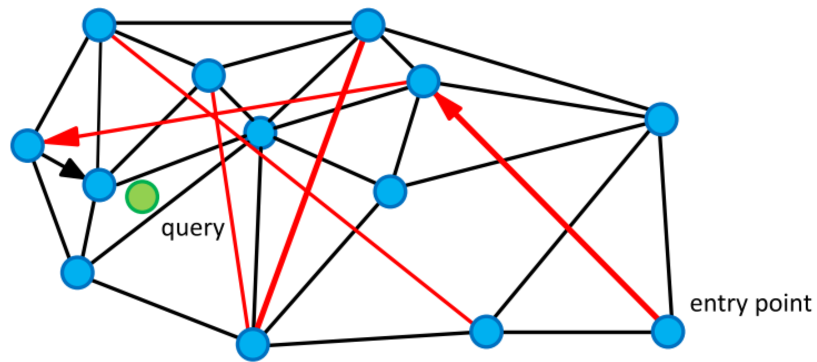
One of the known approaches is **Hierarchical Navigable Small World (HNSW)** which comes from the **NSW approach, based on NSW graphs**. These are the graphs where if the pair of vertices is not connected, they can be accessed via $\log_2 N$ hops in average, where N is number of vertices.

Constructing such an NSW graph helps to perform fast search, although does not allow to compress data, and gives great quality.

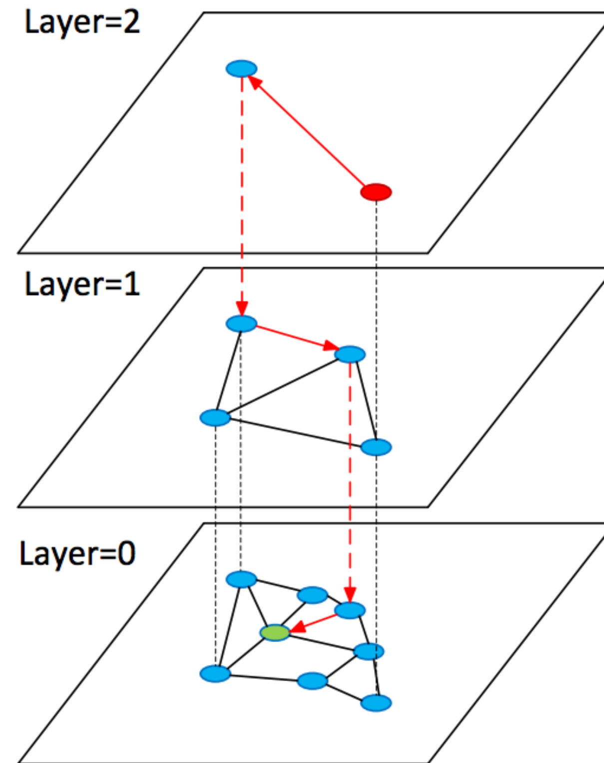
Please refer to this article (in Russian) for detailed algorithms and construction:

https://neerc.ifmo.ru/wiki/index.php?title=Поиск_ближайших_соседей_с_помощью_иерархического_маленького_мира

NSW and HNSW



NSW



HNSW

See FAISS HNSW benchmark:
https://github.com/facebookresearch/faiss/wiki/Indexing-1M-vectors

Images from the article https://arxiv.org/abs/1603.09320

HNSW

Algorithm 1

INSERT(*hnsw*, *q*, *M*, M_{max}, *efConstruction*, m_L)

Input: multilayer graph *hnsw*, new element *q*, number of established connections *M*, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L

Output: update *hnsw* inserting element *q*

```
1  W ← ∅   // list for the currently found nearest elements
2  ep ← get enter point for hnsw
3  L ← level of ep   // top layer for hnsw
4  l ← ⌊-ln(unif(0..1))·mL⌋  // new element's level
5  for lc ← L … l+1
6    W ← SEARCH-LAYER(q, ep, ef=1, lc)
7    ep ← get the nearest element from W to q
8  for lc ← min(L, l) … 0
9    W ← SEARCH-LAYER(q, ep, efConstruction, lc)
10   neighbors ← SELECT-NEIGHBORS(q, W, M, lc) // alg. 3 or alg. 4
11   add bidirectionall connections from neighbors to q at layer lc
12   for each e ∈ neighbors  // shrink connections if needed
13     eConn ← neighbourhood(e) at layer lc
14     if │eConn│ > Mmax // shrink connections of e
                          // if lc = 0 then Mmax = Mmax0
15       eNewConn ← SELECT-NEIGHBORS(e, eConn, Mmax, lc)
                                             // alg. 3 or alg. 4
16       set neighbourhood(e) at layer lc to eNewConn
17   ep ← W
18 if l > L
19   set enter point for hnsw to q
```

Algorithm 2

SEARCH-LAYER(*q*, *ep*, *ef*, l_c)

Input: query element *q*, enter points *ep*, number of nearest to *q* elements to return *ef*, layer number l_c

Output: *ef* closest neighbors to *q*

```
1  v ← ep    // set of visited elements
2  C ← ep    // set of candidates
3  W ← ep    // dynamic list of found nearest neighbors
4  while │C│ > 0
5    c ← extract nearest element from C to q
6    f ← get furthest element from W to q
7    if distance(c, q) > distance(f, q)
8      break  // all elements in W are evaluated
9    for each e ∈ neighbourhood(c) at layer lc  // update C and W
10     if e ∉ v
11       v ← v ⋃ e
12       f ← get furthest element from W to q
13       if distance(e, q) < distance(f, q) or │W│ < ef
14         C ← C ⋃ e
15         W ← W ⋃ e
16         if │W│ > ef
17           remove furthest element from W to q
18 return W
```

Algorithm 3

SELECT-NEIGHBORS-SIMPLE(*q*, *C*, *M*)

Input: base element *q*, candidate elements *C*, number of neighbors to return *M*

Output: *M* nearest elements to *q*

return *M* nearest elements from *C* to *q*

Algorithm 5

K-NN-SEARCH(*hnsw*, *q*, *K*, *ef*)

Input: multilayer graph *hnsw*, query element *q*, number of nearest neighbors to return *K*, size of the dynamic candidate list *ef*

Output: *K* nearest elements to *q*

```
1  W ← ∅  // set for the current nearest elements
2  ep ← get enter point for hnsw
3  L ← level of ep   // top layer for hnsw
4  for lc ← L … 1
5    W ← SEARCH-LAYER(q, ep, ef=1, lc)
6    ep ← get nearest element from W to q
7  W ← SEARCH-LAYER(q, ep, ef, lc =0)
8 return K nearest elements from W to q
```

Pros: High quality of search; **Cons**: No delete operation from index, more memory overhead

What else? A great variety of methods

Table 1 Classification of vector quantization (VQ) methods

Codebook structure	VQ method
Tree	TSVQ (Buzo et al., 1980) HKM (Nister and Stewenius, 2006) RPT (Dasgupta and Freund, 2009) TQ (Babenko and Lempitsky, 2015)
Lattice	LVQ (Gersho, 1979) PVQ (Fischer, 1986)
Classified	CVQ (Ramamurthi and Gersho, 1986) QCVQ (Chen et al., 2014)
Feedback	Feedback VQ (Kieffer, 1982) FSVQ (Foster et al., 1985)
Direct sum	MSVQ/RVQ (Juang and Gray, 1982) ERVQ (Ai et al., 2014) PRVQ (Wei et al., 2014) RVQ-NP (Guo et al., 2016) GRVQ (Liu et al., 2017)

Codebook structure	VQ method
Cartesian product	PQ (Jégou et al., 2010) TC (Brandt, 2010) OPQ (Ge et al., 2013) CKM (Norouzi and Fleet, 2013) DPQ (Heo et al., 2014) LOPQ (Kalantidi and Avrithis, 2014) OCKM (Wang et al., 2014) PTQ (Yuan and Liu, 2015a) KSQ (Ozan et al., 2016a)
Joint	JII (Xia et al., 2013)
Linear combination	AQ (Babenko and Lempitsky, 2014) CQ (Zhang T et al., 2014) SCQ (Zhang et al., 2015) TQ (Babenko and Lempitsky, 2015) LSQ (Martinez et al., 2016) CompQ (Ozan et al., 2016b)

+LSH methods
+KD-trees
+Graph-structure methods

This list is heavily outdated, although list itself shows the variety of approaches in this area

Thank you