* FAISS (Facebook AI Similarity Search)

→ It is a library for efficient similarity search and clustering of dense vectors

→ What is similarity search?

- Given a set of vectors x_i in dimension d, FAISS builds a data structure in RAM from it. After the structure is constructed, when given a new vector x in dimension d it performs efficiently the operation

$$j = \text{argmin}_i \|x - x_i\|$$

where $\|\cdot\|$ is the Euclidean distance (L^2)

- In FAISS terms, the data structure is an index, an object that has an add method to add x_i vectors.
  NOTE: x_i are assumed to be fixed

→ Popular Indexes:

1. Flat
2. LSH
3. HNSW
4. IVF

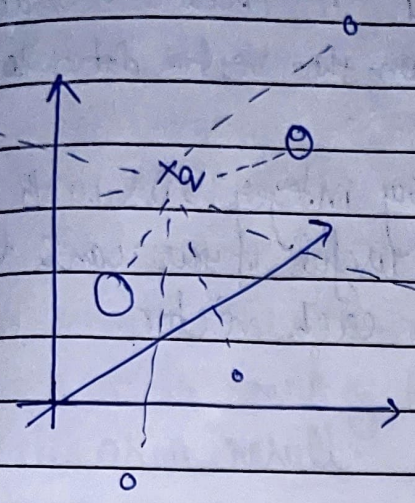
x_b = for the database, that contains all the vectors that must be indexed

x_q = query vector(s) for which we need to find the nearest neighbors

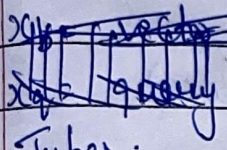
K = find K-nearest neighbors

(KNN)

① Flat - Simplest version that just performs brute-force L2 distance search



When to use Flat index?
① Search quality is a high priority
② Search time doesn't matter OR when using a small index (<10k)



Types:
- IndexFlatL2 – using Euclidean distance
- IndexFlatIP – using Inner Product distance

Code

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('model_name')

import faiss
sentence_embeddings = model.encode(data)
d = sentence_embeddings.shape[1]
index = faiss.IndexFlatL2(d)
index.is_trained   # to check if index requires training. Since it is
                   # flat index and is already trained, this is_trained
                   # returns True
index.add(sentence_embeddings)
index.ntotal   # no. of embeddings

xq = model.encode('query sentence')
```

K = 4 # Similar vectors need for input xq

D, I = index.search(xq, K)

print(I)

>> [[4586 10252 12465 190]] # these are indexes from the vector database

Note: you can add/associate arbitary integer IDs with the vectors in the index. This can be useful if you want to store some metadata or identifiers for each vector.

How?

d

index = faiss.IndexFlat2(d)

index = faiss.IndexIDMap(index)

index.add_with_ids(embeddings, df.id.values)

↓ document embeddings

↓ dataframe whose ids we want to associate

② IVF ⇒ The Inverted File Index consists of search scope reduction through clustering.

→ IVF works on Voronoi cell concept.

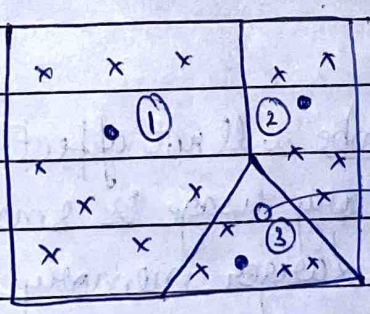
→ Voronoi cells are regions of space that are defined by the distance to a given set of points, such that each point in a cell is closer to its corresponding point than to any other point

→ One way to understand the difference b/w K-Means and voronoi cells is to think of Kmeans clustering as a process and voronoi cells as a result.

→ Now, every datapoint will be contained within a cell, and be assigned to that respective centroid

→ One Problem ⇒ if our query vector (x_q) lands near the edge of a cell, there's a good chance that its closest other datapoint is contained within a neighboring cell. We call this the edge problem

Ex.



→ query with n-probe = 1, i.e. scope is restricted to ③ cell only

Soln: We increase n-probe to >1, ex. to 5 now, the query will search for results in 5 nearest cells

> **Python Code**

d = 786

```
n_list = 128   # no. of cells/clusters to partition data into
quantizer = faiss.IndexFlatIP(d)  # dimension of vectors = d
index = faiss.IndexIVFFlat(quantizer, d, n_list)
index.train(sentence_embeddings)  # we must train the index to cluster into cells
index.add(sentence_embeddings)
index.nprobe = 8  # set how many of nearest cells to search
D, I = index.search(xq, k)
```

- **n_list** means that we must compare our vector to more centroid vectors - but after selecting the nearest centroid's cells to search, there will be fewer vectors within each cell. So, increase n_list to prioritize **search speed**

- **n_probe** - we find the opposite. Increase n_prob increases the search scope - thus prioritizing search quality

- In terms of memory, nprobe will not affect this. The effect of nlist on memory usage is small too. Higher nlist means a marginally larger memory requirement

- So, we must decide b/w greater search-quality with nprobe and faster search speed with nlist

What is the role quantization?

① Indexing Process

→ During indexing (adding vectors to IndexIVF), each input vector is assigned a quantisation index by the quantiser (which is an instance of another FAISS Index, such as IndexFlatIP, IndexFlatL2 or IndexIVFPQ, IndexLSH)

→ The quantisation index determines which inverted list (cell) the vector belongs to

→ The group Id (quantization Index) maps to a specific inverted list (cell), where the Id (vector Index) of the vector is stored

→ The inverted list (cell) object is required only after training

② Search Process

→ During search, when a query vector is quantised using the same quantiser, it obtains a quantisation Index (group Id)

→ The search process involves looking up the inverted list (cell) corresponding to this quantisation index

→ The vectors within inverted list (cell) are then considered for further distance computation

③ Multi-Probe :

→ By default, only the inverted list corresponding to the quantisation index is selected

→ In multi-probe:

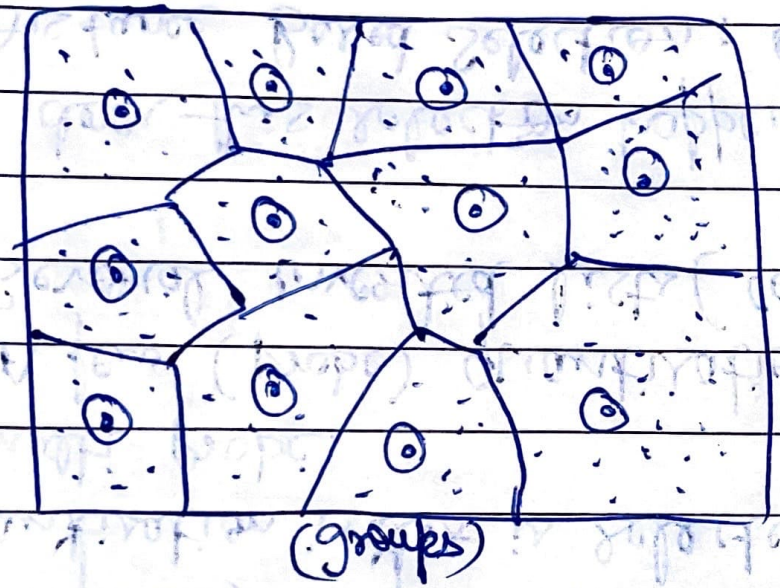
3.1 A few (probe) quantisation indices are selected

3.2 Several inverted lists (cells) are visited during search

→ How does this selection happen?

↳ Distance Based Selection:

(Indexing process)



(Typically an integer value) ← Quantized Index / Group Id

S.No		Id
1		—
2		—
3		—
4		—
		—
		↑
12		1

Vector Id of Centroid

- Vector Id
  - Id of centroid mapped to quantized Index

Each Id is mapped to the original vector Ids within its cluster

(Search process)

x_q ⇒ Input query → (Converted to) Quantized Index → Gets the group Id (multiple group Id in case of n_probe > 1)

Search in those clusters to get answer