Indexing, retrieval, semantic search

TDA 2025 week 2

Information Retrieval

- Quick recap and partial reminder from the Intro to HLT course
- Information Retrieval (for the purpose of this lecture/course):

Given a query and a set of documents, which documents are most relevant to the query?

- Think web search

BoW-based retrieval - Naive

- BoW = Bag of Words (no order information)
- Naive definition:

$$score(D,Q) = \sum_i f(q_i, D)$$

- f(term,document) -> frequency of term in document (term frequency or TF)

BoW-based retrieval - TF-IDF

- Naive approach -> common words like “and” overpower the score
- Common words need to be discounted

$$IDF(q_i) = \log \frac{N}{n(q_i)}$$

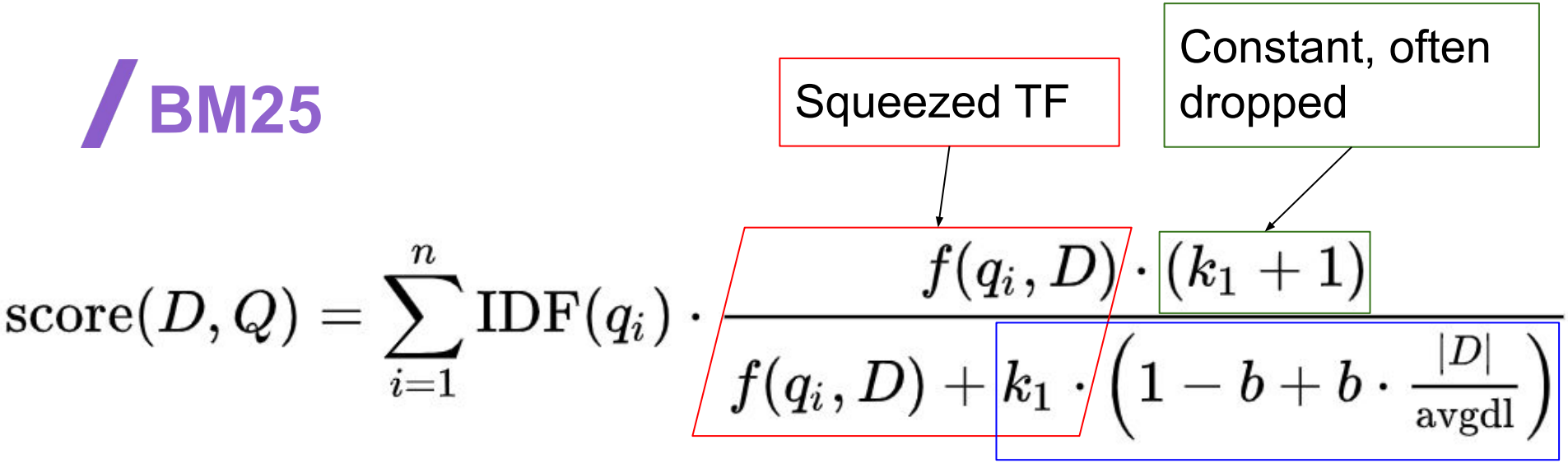
- IDF = inverse document frequency, N is total number of documents, n(q_i) is number of documents containing q_i, log “squeezes” the values

$$score(D, Q) = \sum_i f(q_i, D) \cdot IDF(q_i)$$

BoW-based retrieval - BM25

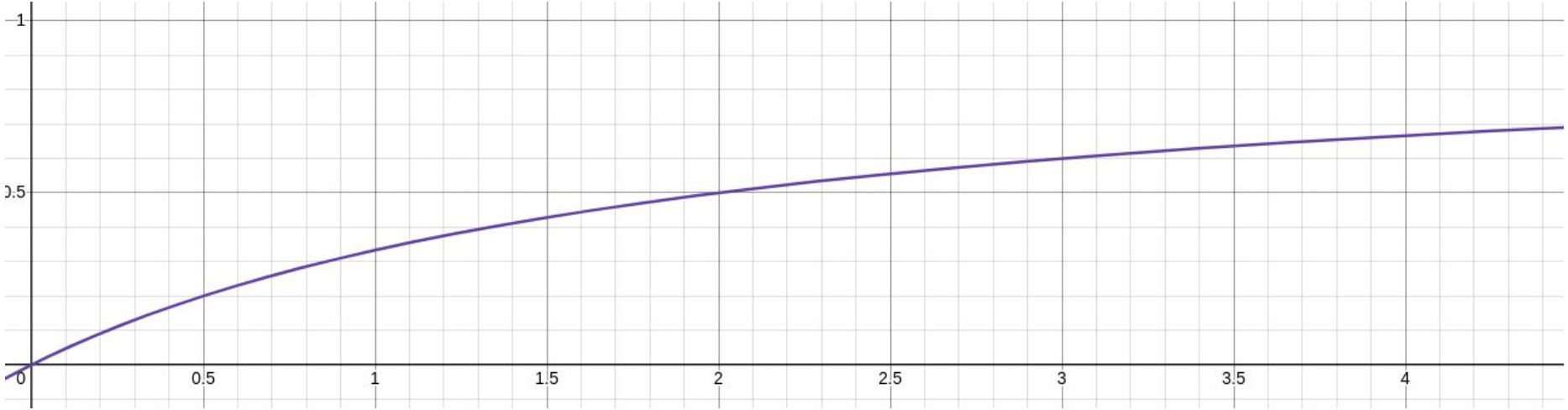
- There are a number of variations to the basic TF-IDF formula accounting for several effects:
  - TF needs to be “squeezed” as well -> if “cat” is in a document 20x it does not mean the document matches “cat” 20x more strongly than if it was there once
  - So TF should be squeezed
  - Also: Longer documents have naturally higher TF simply because they are longer
  - Discount longer documents more strongly, to avoid bias towards long documents
- There are a myriad of scoring functions in BoW-based retrieval
- You are most likely to meet BM25
  - (BM = Best Matching)
  - Minimally as a baseline

BM25



- k_1, b are parameters that can be tuned to your needs
- avgdl: average document length

x/(x+2)



Retrieval as vector comparison

- View both document and query as (very, very) sparse vectors (number of dimensions == vocabulary size)
- Then we can calculate the score as a simple dot product

$$score(D,Q)=\sum_i q_i d_i = D \cdot Q = \langle D, Q \rangle$$

- Here d_i could e.g. be the BM25-style TF-IDF score
- Note that if we set q_i=1 for all terms in the query, this is exactly equivalent to the BM25 formula
- You may want to set q_i to some value expressing the weight of the query term

Retrieval as vector comparison (cont.)

- Seeing the score as dot product of the query vector and the document vector =>
- If we think of our document collection as a matrix of size (num_documents, vocab_size) then we can see retrieval as multiplying this matrix by the query vector
- This will be a useful way of thinking of this when we get to dense embeddings
- (But is also useful for quick'n'dirty implementation of retrieval using some sparse matrix implementation like the one in SciPy)

BoW -based retrieval

- Definitely relevant and still valid especially for keyword-oriented retrieval applications
- BM25 a strong baseline for document search
- BUT:
  - Does not take into account word order
  - Does not take into account meaning (e.g. synonyms)
  - Various query expansion techniques can be applied to account for these
- Recently: shift to *embeddings* and dense representations
- We covered the dense representation models at quite some depth in the DL-in-HLT course, so here a simple recap now

Quick recap from DL in HLT course: Dense embeddings

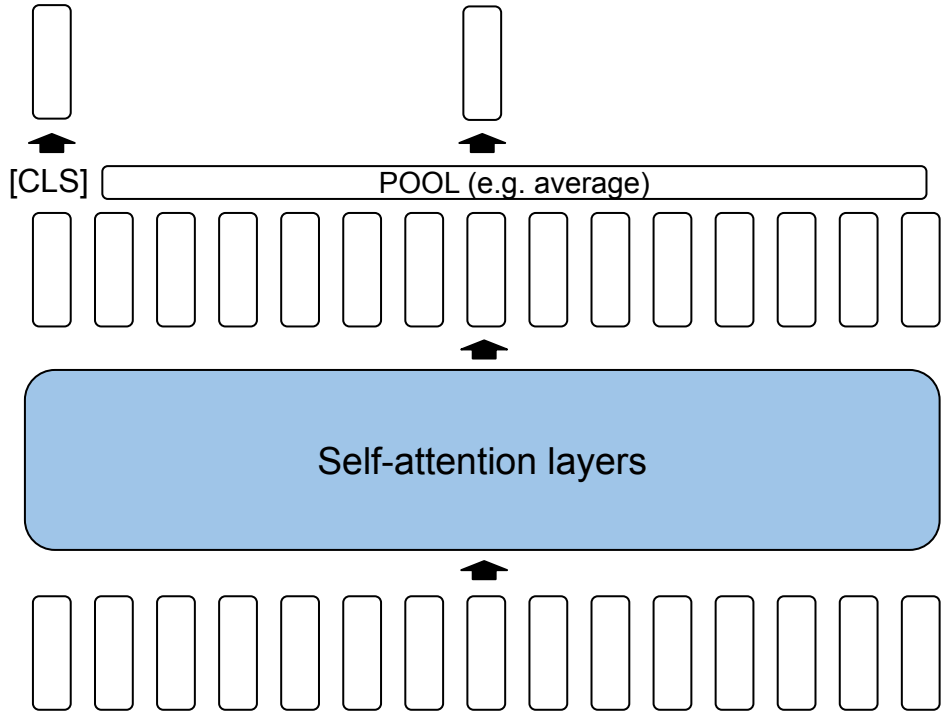
Embeddings of units of text

- Reminder: Embedding = dense vector representation
- Embedding words
  - Comparatively easy - there is just so many words in the end
- Embedding larger units of text
  - There are many words
  - But there are many more sentences! :)
  - Some methods
    - Average of word2vec word embeddings
    - LSTM final states
    - Skip-Thought (generate surrounding text from sentence embedding)
    - Average of Transformer word output embeddings
    - [CLS] token embedding where applicable
    - Dedicated models

Practical use of text embeddings

- Comparison of the embeddings -> measure of similarity between texts
- Often carried out at a mass scale -> cosine similarity has a huge computational advantage since it boils down to a dot product of normalized vectors
- Mass scale might mean comparing 500M text segments in a pairwise manner
  - Cannot afford a complex computation to establish the similarity of each pair
  - Can afford a (relatively) complex computation to compute the embedding for each individual text segment

Embedding with BERT



Sentence Transformers

- BERT is not trained to produce particularly good text embeddings
  - It does a decent job, but that has never been its training objective
- Sentence transformers are an example of a method that can be used to enforce better text embeddings
- Also a good example of:
  - Fine-tuning on a proxy task so as to manipulate the model itself
  - Cross-lingual transfer learning
- By no means only such method, rather a representative of a class of methods

Sentence Transformers

- Fine-tune the model on pairwise sentence classification tasks
  - Remember that in BERT [CLS] is trained on the next sentence prediction (NSP) task
  - But are there other such tasks? Is NSP really a good way to get text embeddings?
  - Yes - there are other tasks! The model is fine-tuned on the Stanford Natural Language Inference (SNLI) dataset and many other text pair datasets

Sentence transformers

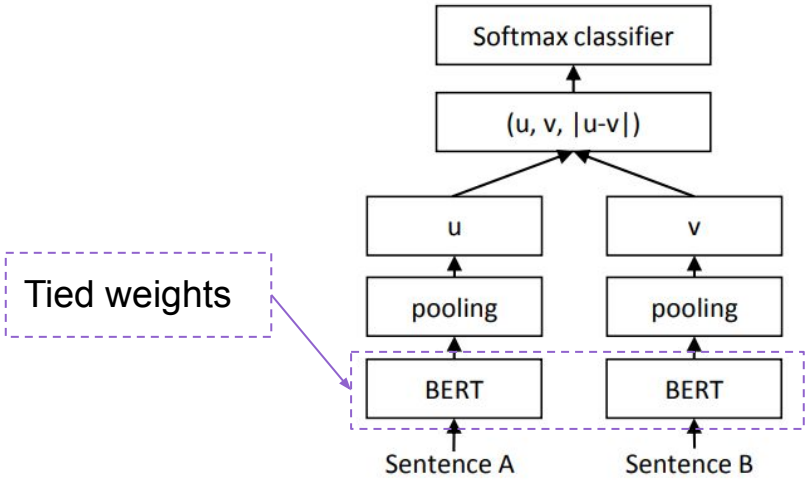


Figure 1: SBERT architecture with classification objective function, e.g., for fine-tuning on SNLI dataset. The two BERT networks have tied weights (siamese network structure).

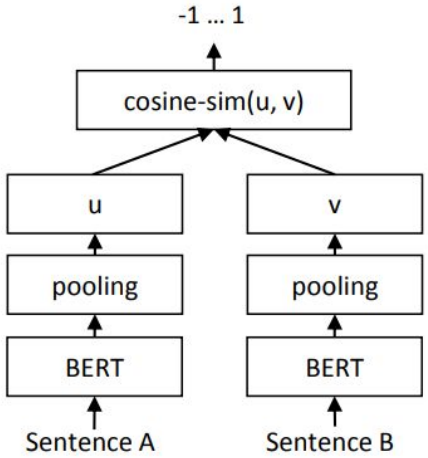


Figure 2: SBERT architecture at inference, for example, to compute similarity scores. This architecture is also used with the regression objective function.

https://arxiv.org/pdf/1908.10084.pdf

Sentence Transformers: cross-lingual transfer

- Multilingual sentence transformers are a natural extension of the monolingual model
- But how to get them without the likes of SNLI for other languages?
- What we have:
  - English sentence transformer, fully trained on SNLI or other similar data
  - Lots of translation pair data across languages in the OPUS dataset
  - Multi- and monolingual pre-trained transformer language models for many languages (but these are not of the sentence transformer kind)
  - Multilingual model: model pre-trained on data in many languages without an explicit pairing across languages

Crosslingual sentence transformers

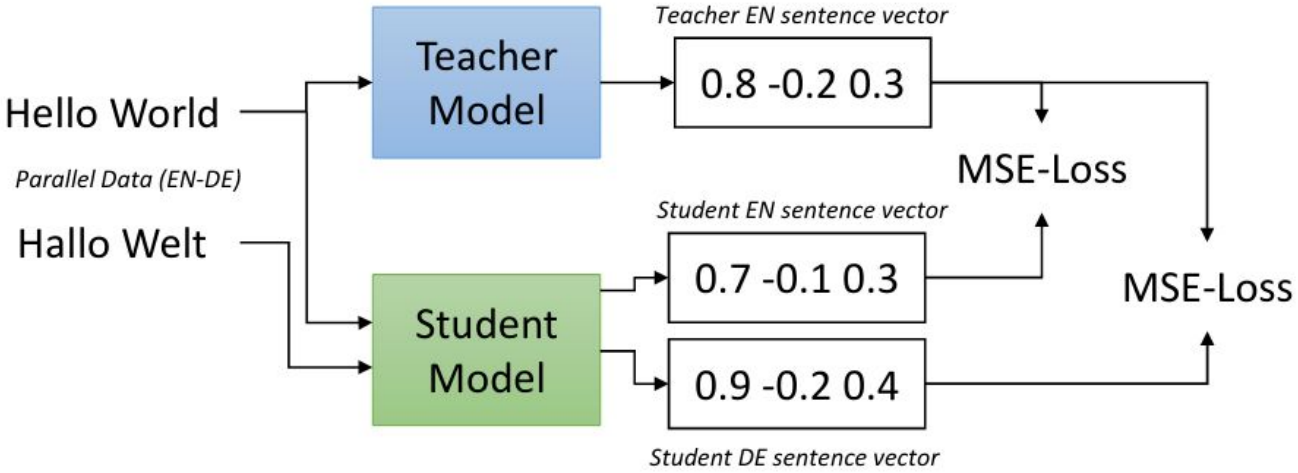


Figure 1: Given parallel data (e.g. English and German), train the student model such that the produced vectors for the English and German sentences are close to the teacher English sentence vector.

https://arxiv.org/pdf/2004.09813.pdf

Embedding models - summary

- Traditionally the best embedding models were based on encoders, recent approaches found ways in which LLMs can be utilized
- All models apply some form of contrastive training since the training data is always in the form of pairs (as there is no source of ground truth embedding vectors)
- Training based on a large number of datasets, some large mined, some small manually created; not all necessarily “embedding” or “retrieval” -like tasks in their nature, though
- Evaluation typically based on MTEB and the MTEB leaderboard has the best models to use

Nearest Neighbor search

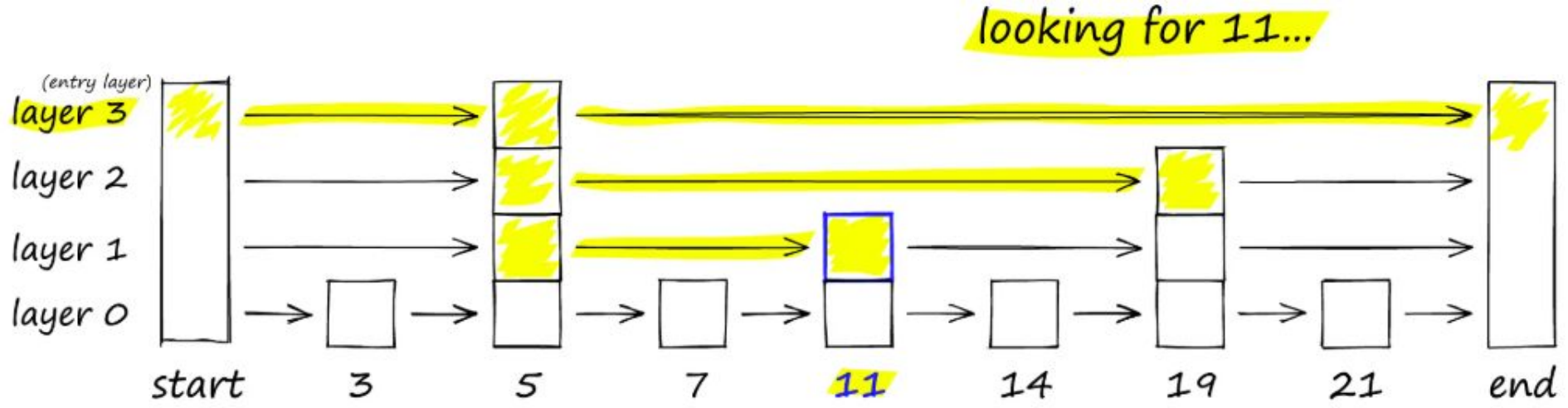
- Retrieval tends to happen in very large collections
- The search space can be a collection of 100s of millions of text snippets, each snippet represented by an embedding
- The query is also represented by an embedding
- Retrieval/search boils down to:
  - Given a similarity/distance measure (typically cosine, or Euclidean)
  - Find K nearest neighbors of the query vector among the collection
- Naive bruteforce solution works just fine for modestly-sized collections
- Not doable for 100s of millions vectors large collections

K-NN search

- **Exact retrieval** returns the exactly correct list of K nearest neighbors, but leads to a large number of comparisons
- **Approximate retrieval** not guaranteed to return the exactly correct list of K nearest neighbors, but allows for faster search algorithms
- Among the most popular: HNSW (Hierarchical Navigable Small Worlds) and Inverted File (IVF)
- In the next few slides, I will borrow some material and illustrations on HNSW from [this tutorial] by Pinecone and hereby acknowledge them as the source of the material and thank for it :)

Vector search requirements and evaluation

- Requirement 1: low memory footprint
- Requirement 2: fast lookup time for K-NN retrieval
- Requirement 3: recall: overlap between the K-NN list retrieved by the algorithm and the correct gold standard K-NN list in the data; this can serve as the evaluation criteria
- These are in some sense mutually exclusive, i.e. improving one tends to degrade the performance on others

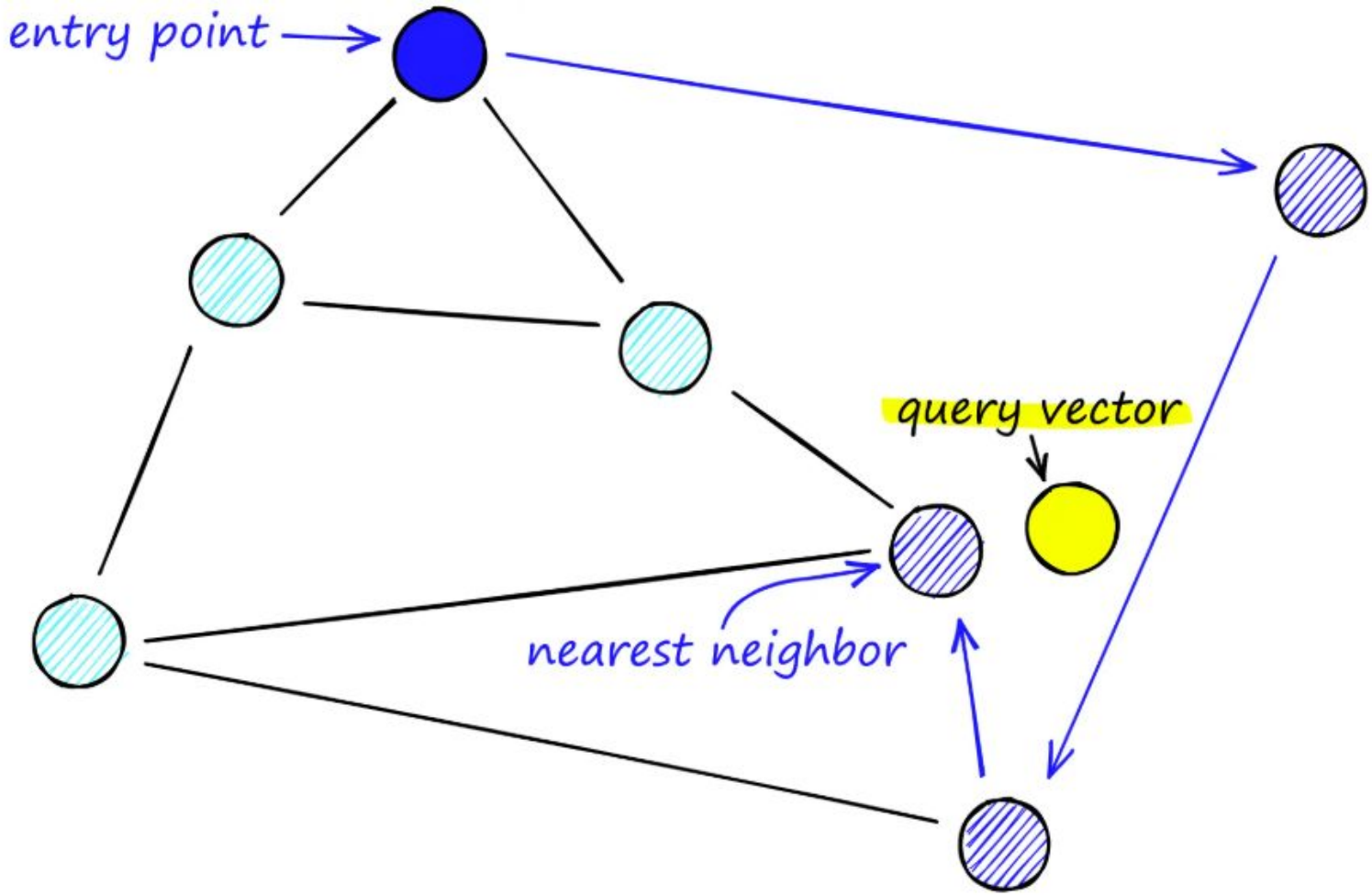


A probability skip list structure, we start on the top layer. If our current key is greater than the key we are searching for (or we reach end), we drop to the next layer.

- Skip-list is sparse on the top layer (few connections, large jumps)
- Gets denser as you go deeper
- Adjusting the number of layers and their density adjusts the memory vs lookup time tradeoff

Navigable small world

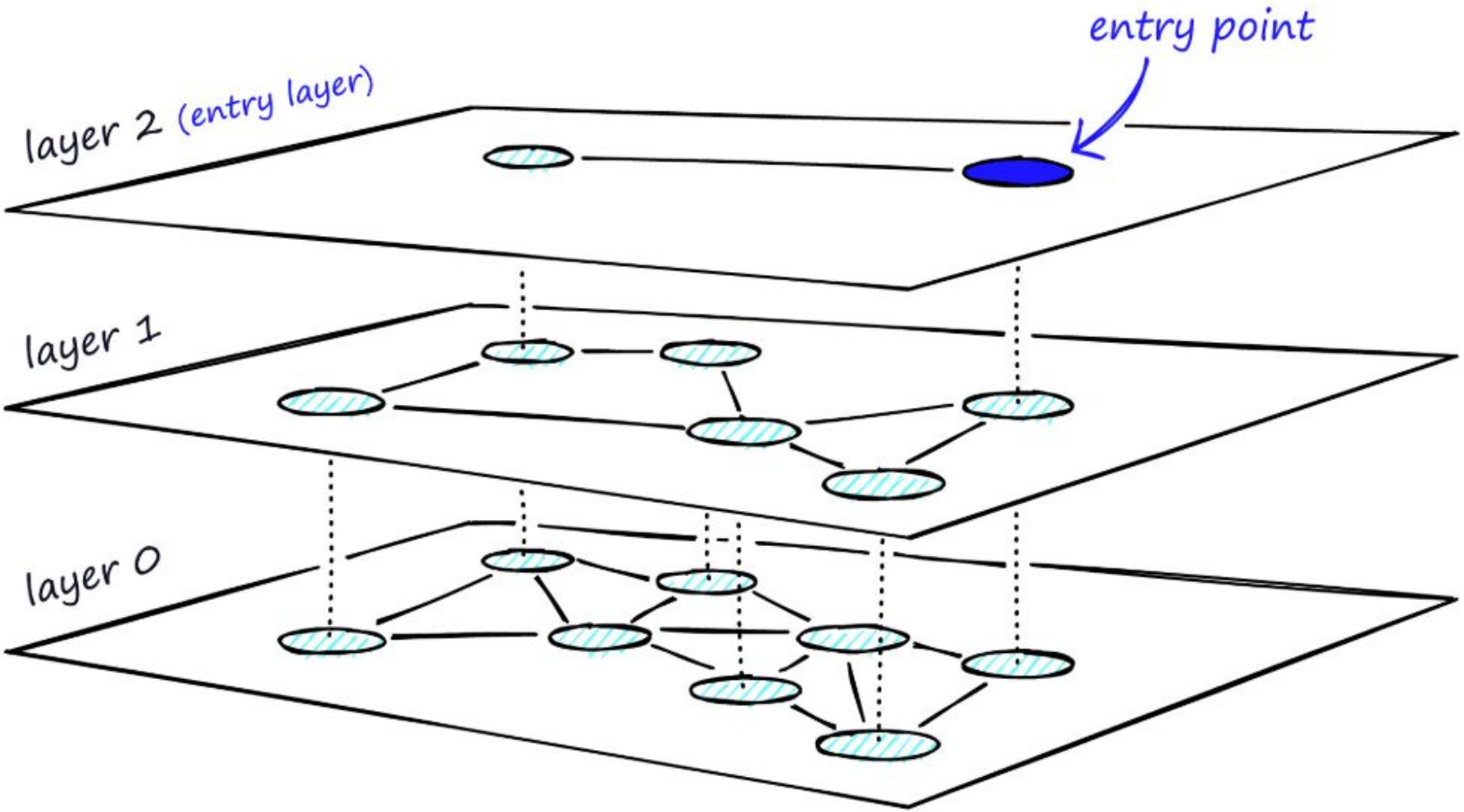
- The skip-list idea can be extended into a navigable graph
- The graph has a fixed entry point and you walk it as long as the distance between the vectors and your query vector decreases
- Get to a point, look at its neighbors, and proceed if you find a neighbor which is closer than the present point
- Eventually, you will get stuck in a local minimum
- The density of the graph controls the length of the “jumps”; sparse connections lead to large jumps; dense connections allow for a finer search, but increase memory consumption and may lead to local minima more often



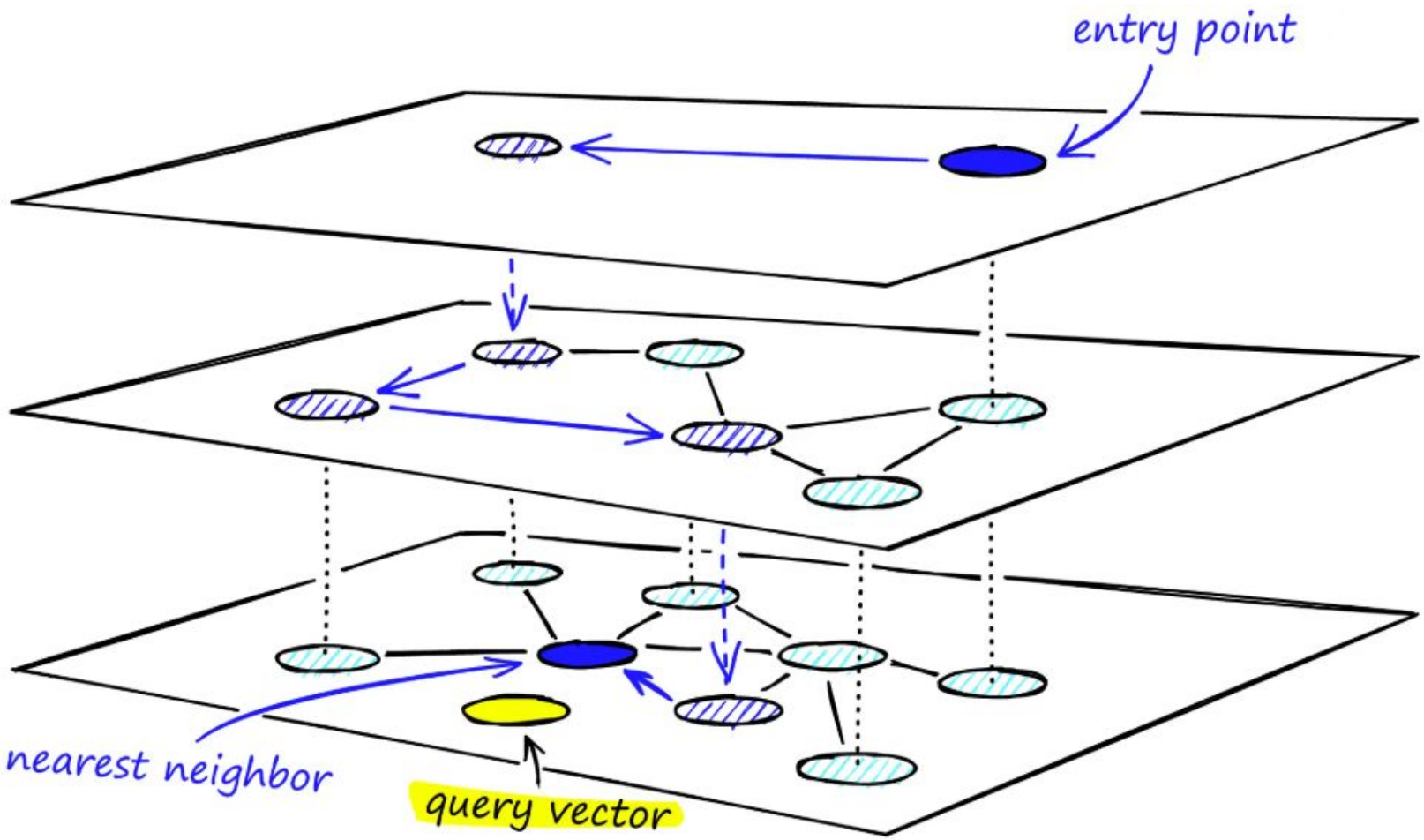
The search process through a NSW graph. Starting at a pre-defined entry point, the algorithm greedily traverses to connected vertices that are nearer to the query vector.

HNSW

- Combines the navigable small world idea with the skip-list hierarchy idea
- Start from the top level, entry point, walk till distance no longer improving, drop one level lower and repeat
- The construction algorithm ensures that a node is present in all layers below the present level as well
- Eventually you will reach the bottom-most level and stop at a local minimum and gather your nearest neighbors at that level



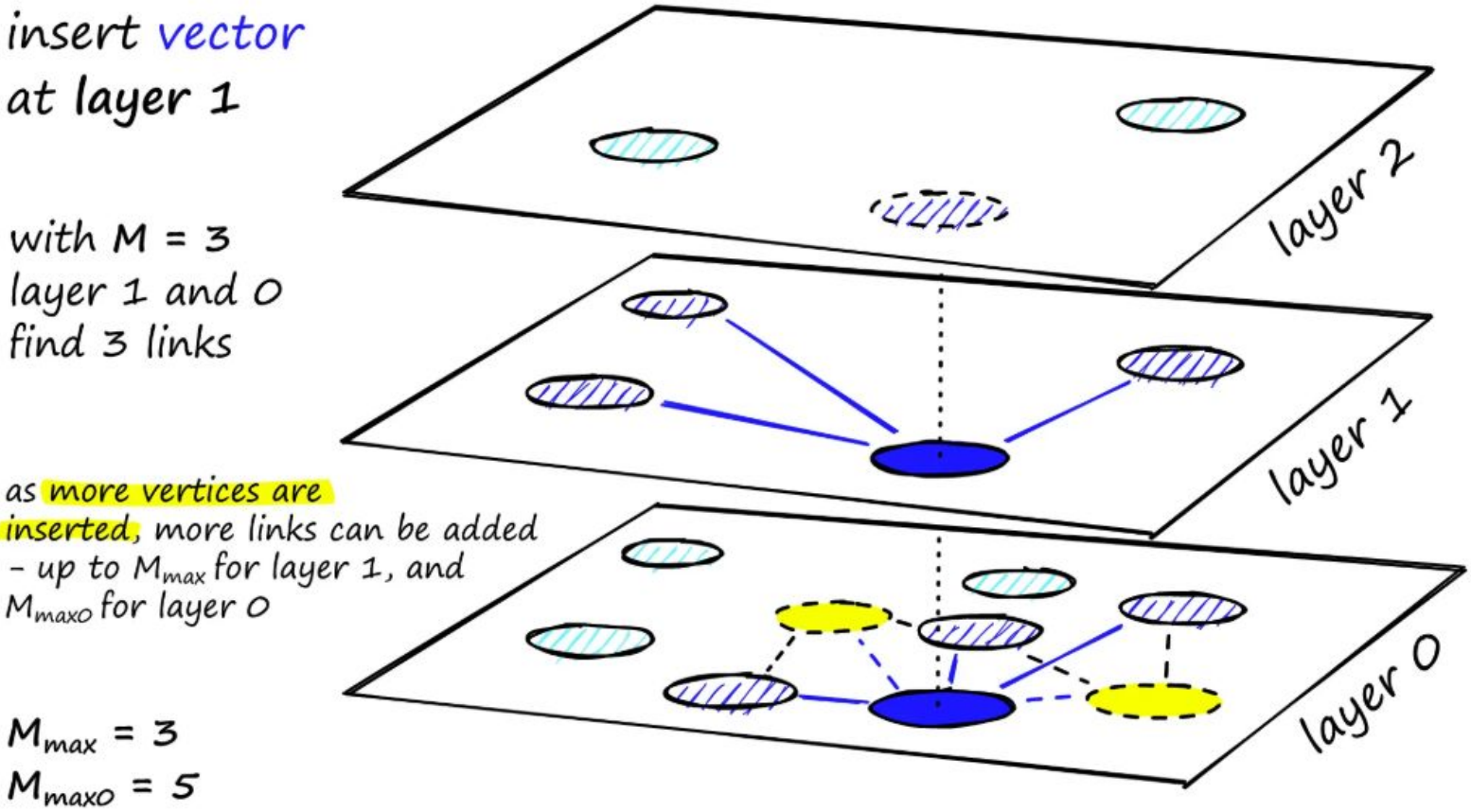
Layered graph of HNSW, the top layer is our entry point and contains only the longest links, as we move down the layers, the link lengths become shorter and more numerous.



The search process through the multi-layer structure of an HNSW graph.

Building HNSW

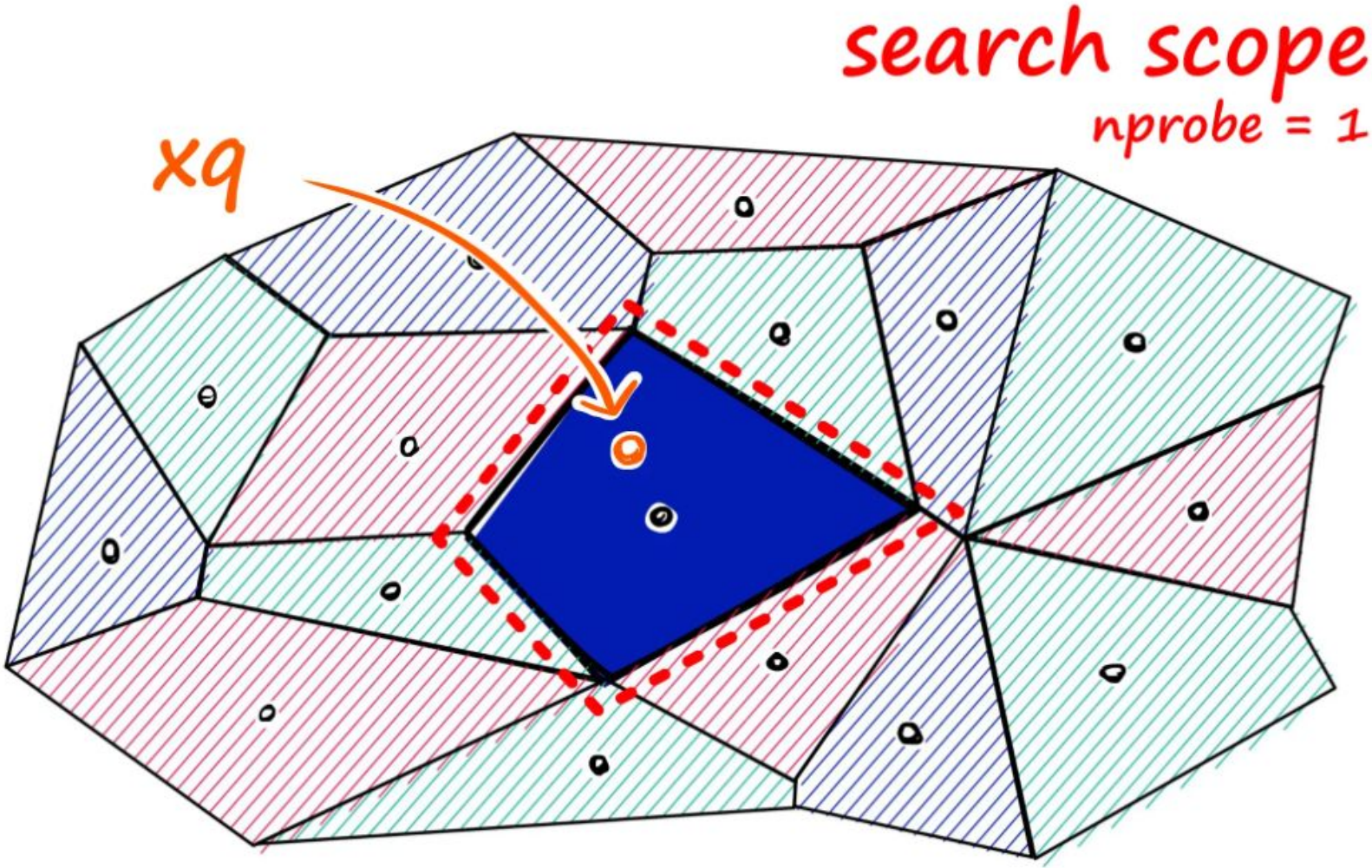
- Vectors are inserted one at a time
- An insertion layer is chosen randomly (such that the layers sparsify as we go up)
- Run normal lookup for the new vector up until the insertion layer
- On the insertion layer (and all below) choose a suitable number of nearest neighbors to build links to (and control the number of links through a parameter)
- This is a heavy process to run for very large vector collections
- But empirically HNSW gives very good results

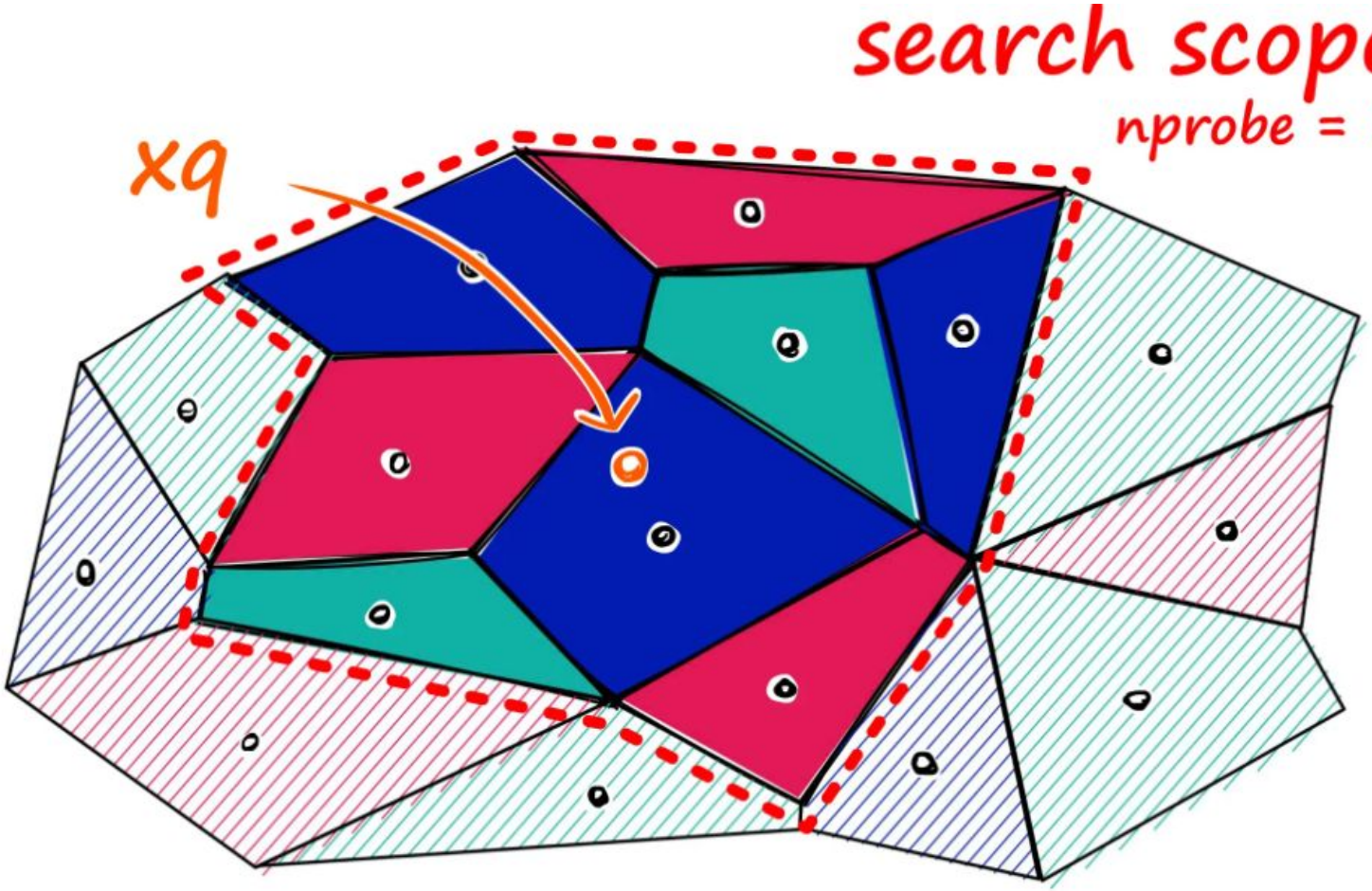


Explanation of the number of links assigned to each vertex and the effect of M, M_max, and M_max0.

Pre-cluster/Voronoi Cells (IVF)

- Faster alternative to HNSW
- Pre-cluster the set of vectors, effectively subdividing the space into partitions, Voronoi Cells
- Each partition holds a subset of our full collection
- On query time, we find the closest partition to the query, and narrow the search down to that partition, basically ruling out the rest of the search space entirely
- Alternatively, we can expand to (a small number of) neighboring partitions to increase our reach
- Within one partition, one might use an exhaustive search, but of course other indices are also possible, in theory
- If you have very many clusters/cells (tens of thousands), you can use HNSW to find the right cell for the query


search scope
nprobe = 1
xq



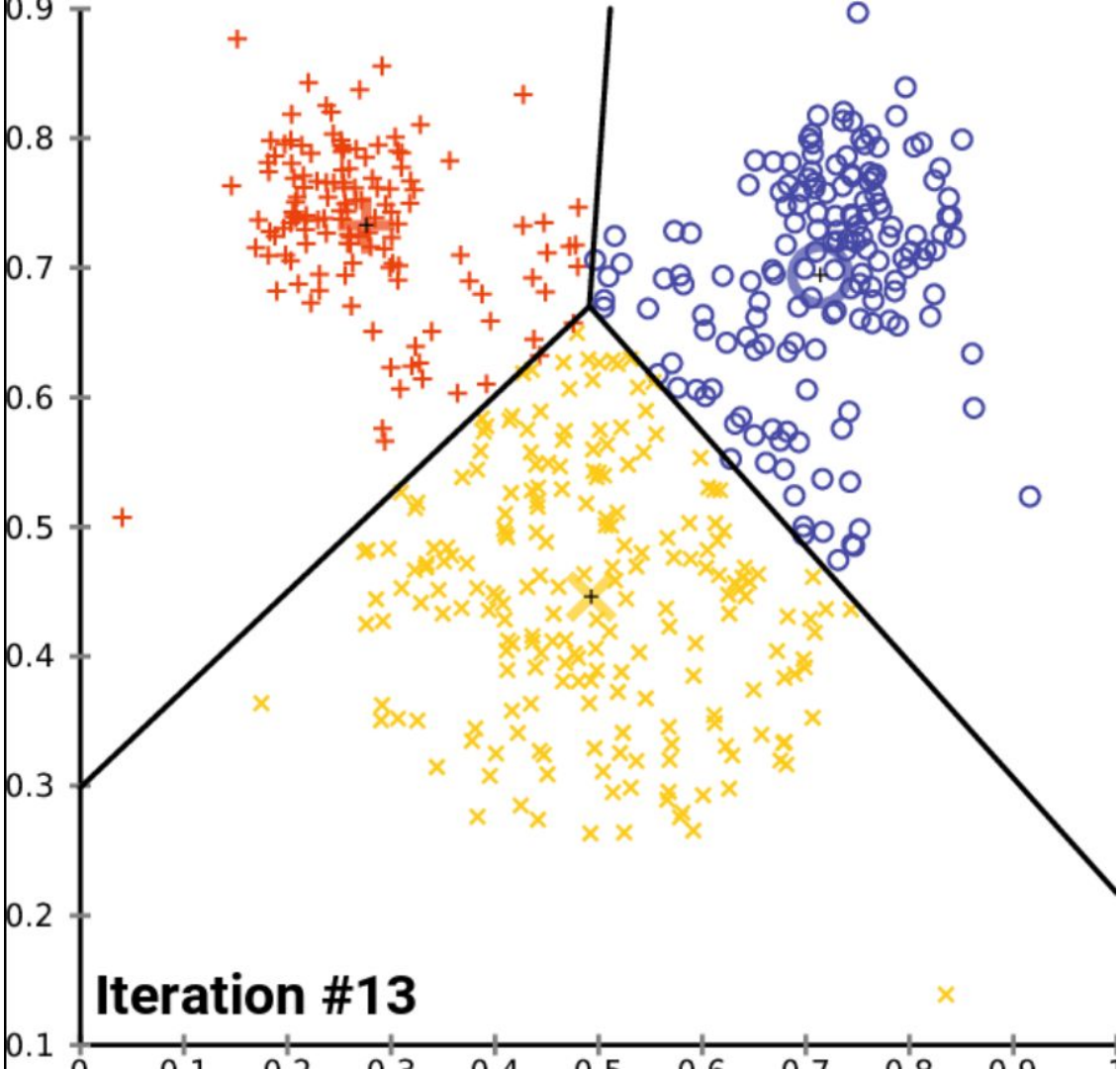
Increasing nprobe increases our search scope.

Product quantization

- HNSW search is efficient as search, but does not alleviate memory footprint in any way
- We are still remembering every vector
- 1B (10^9) vectors, each 8KB in size (2000-long at 4B float) is 8TB of memory just for the raw data … that's kinda plenty :D
- Quantization is a handy memory footprint reduction technique
  - There are many variants and nuances
  - Let's look at the basic idea
- (I will be taking illustrations from [this Pinecone page] - thanks and acknowledged!)

Little detour: K-Means

- K-means is a *clustering* algorithm
- Given data as set of vectors and the value of K, K-means will assign each datapoint to one of K clusters, each cluster represented by its *centroid* (middle point)
- Each point is assigned to the cluster whose centroid is nearest
- The centroids are found by a simple iterative algorithm (which is not hugely relevant to us here):
  - Start with K random centroids
  - Then:
    - Assign each point to its nearest centroid
    - Calculate new centroids
    - Repeat till convergence



Source: Wikipedia

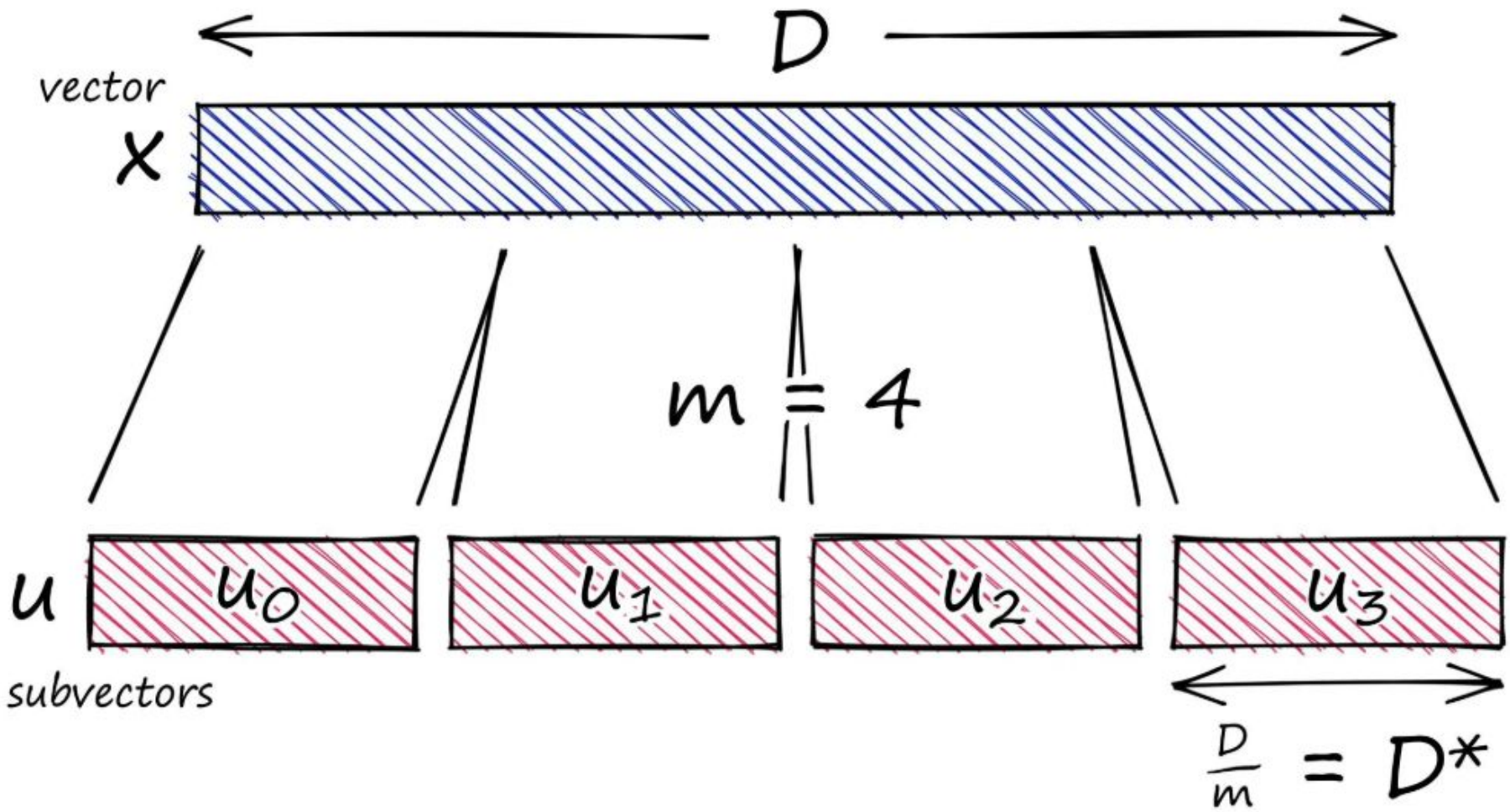
Product quantization (cont.)

Quantization

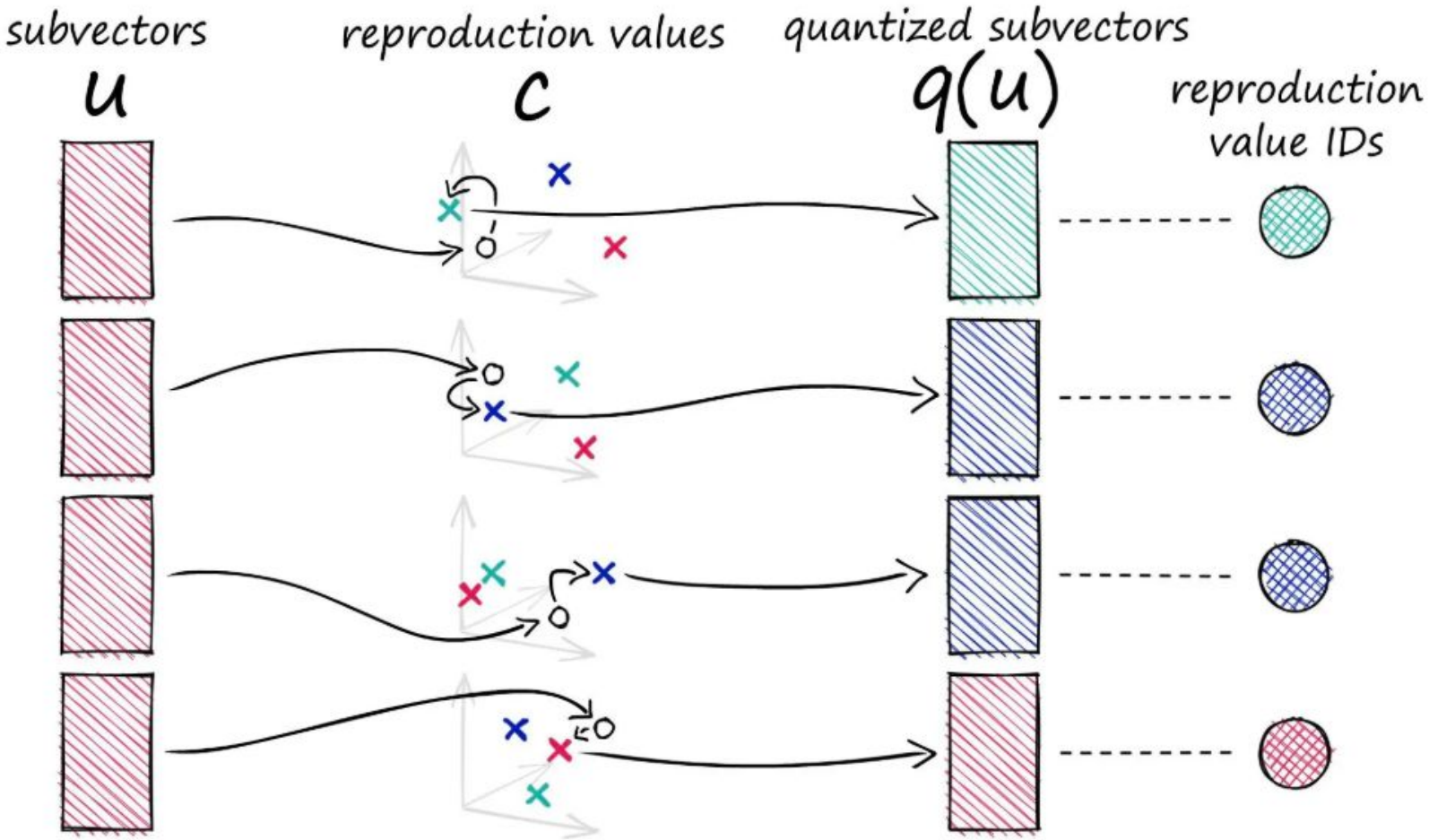
- Let us say you take your vectors and run K-means for K=4096
- Then you represent each vector by the ID of the cluster to which it belongs
- That way you only need to store 4096 cluster centroids plus one ID for each vector
- For 4096 clusters, the ID would be 12-bit integer
- That is most likely substantially less than the storage of the original vector, whatever it was
- Of course you lose information, since you can now only represent 4096 different unique vectors (which is why we call it quantization)

Product quantization

- The full idea is to quantize partitions of our vectors
- Divide the vector into m partitions (of course the number of dimensions D must be divisible by m)
- Have a separate k-means for each partition
- The quantized vector is then a sequence of D/m integers, each integer being an ID of a k-means cluster of that particular partition
- That is a lot, lot more compact than the full vector, but very precise



Here we are splitting our high-dimensional vector x into several subvectors u_j.



Our subvectors are replaced with a specific centroid vector — which can then be replaced with a unique ID specific to that centroid vector.

PQ and HNSW/IVF combined

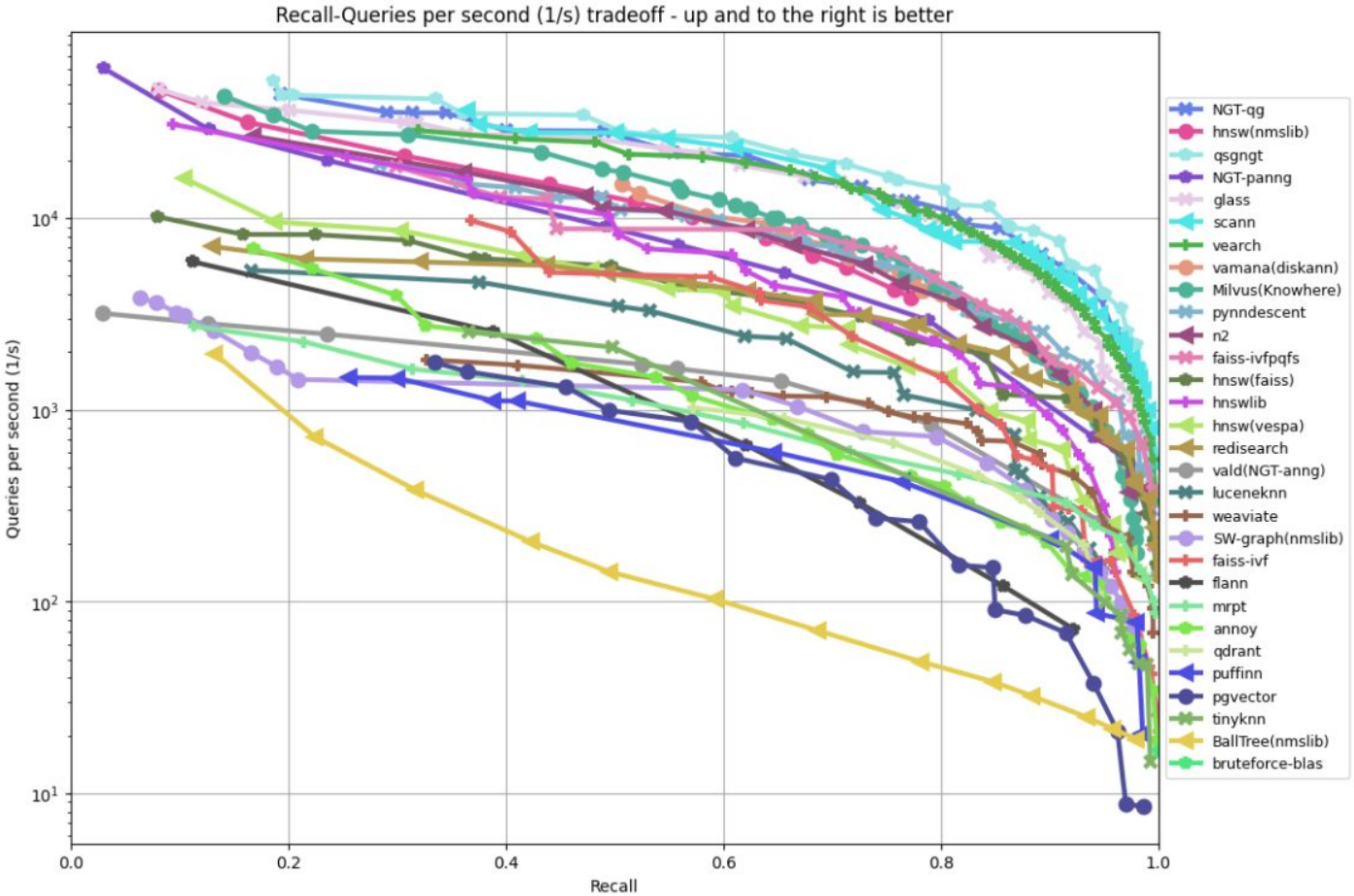
- These are two orthogonal techniques: one deals with fast, non-exhaustive search, the other makes your vectors smaller
- The HNSW index can be built using the quantized vectors
- When searching, the quantized vectors can be “reconstructed” (not losslessly, though!) by replacing the partition IDs with the centroid vectors
- These can be compared with the query vector as needed
- Warning! There are levels of complexity I am not getting into
  - Performant variations of these techniques
  - Their hyperparameters
  - Their various combinations
  - Residual quantization
- For the curious mind: the FAISS paper

Training an index

- This may not be immediately obvious:
- PQ and IVF must be “trained”
- Both are based on pre-clustering the data to obtain the centroids
- So it is necessary to train the indices (i.e. build the centroids based on a sample of the data)
- For very massive indices, where say the number of IVF partitions is very large (tens of thousands) you naturally also need substantial amount of data to train the index

FAISS and other libraries

- FAISS is a popular library implementing all of the techniques mentioned so far (and then some!)
  - Implements HNSW
  - Implements the pre-clustering, called IVF (inverted file) in FAISS terminology
- Includes GPU acceleration for some indices
- Popular, but not the only choice
- Basis for a number of popular vector databases (pinecone, elastic, …) which hide the complexity of FAISS
- Check out https://ann-benchmarks.com/
- In general: HNSW is better than IVF but IVF starts becoming the only choice for very massive indices to keep memory and indexing time in check



https://ann-benchmarks.com/

Some example uses of these techniques

Retrieval component of RAG

- RAG - retrieval-augmented generation
  - Given a query (usually a question) retrieve K most relevant documents from a collection
  - Catenate these documents together with the question to form a new prompt
  - Use LLM to generate from that prompt, hopefully using the retrieved information
- Note: here we somewhat stretch the notion of semantic similarity
- Question is not particularly similar to its answer if we take “semantic similarity” to mean “similar in meaning / synonymous”
- Recall, that the recent embedding models are trained on text segment pair datasets
- Many are explicitly training for Question-Answer similarity

Translation mining

- Remember that the embedding models are cross-lingual
- Similarity holds also across languages
- A body of documents in one language can be compared to a body of documents in another language
- Assumes that translation pairs lead to a high similarity that can be thresholded and detected

Translation mining from books