Assignment Report

Scalable Data Mining

Assignment 3

Hardik Soni 20CS30023

29th October, 2023

Introduction to Approximate Nearest Neighbor (ANN) search

Approximate Nearest Neighbors (ANN) is a method employed to effectively locate the nearest points in spaces with a high number of dimensions. This technique finds utility in fields like data mining, machine learning, and computer vision.

Over time, there have been advancements in Approximate Nearest Neighbor search algorithms, with recent progress focusing on methods involving graphs, multilabel classification, and kernel density estimation. These approaches have demonstrated promising results in terms of both speed and accuracy. However, they encounter challenges such as converging to local optima and the time-consuming process of constructing graphs. To address these issues, researchers have put forth various solutions. These solutions include improved initialization for expanding nearest neighbors, customized floating-point value formats, and optimization methods for dictionaries.

Recent research in ANN introduces EFANNA, an exceptionally rapid algorithm based on the kNN Graph, which combines the strengths of hierarchical structure-based and nearest-neighbor-graph-based methods. Another study introduces DEANN, an algorithm that accelerates kernel density estimation through ANN search. Furthermore, researchers have delved into the theoretical guarantees associated with solving NN-Search through a greedy search on ANN-Graph for low-dimensional and dense vectors.

Practical applications of ANN span across machine learning domains such as image recognition, natural language processing, and recommendation systems. Companies like Spotify leverage ANN to enhance their music recommendation algorithms, offering users more precise and personalized suggestions.

In summary, Approximate Nearest Neighbors stands as a potent method for efficiently identifying the nearest points in high-dimensional spaces. As ongoing research continues to progress, ANN algorithms are poised to become even faster and more accurate, expanding their potential applications and impact across various industries.

Locality Sensitive Hashing

Locality sensitive hashing (LSH) is a widely popular technique used in *approximate* nearest neighbor (ANN) search. The solution to efficient similarity search is a profitable one — it is at the core of several billion (and even trillion) dollar companies.

LSH is one of the original techniques for producing high quality search, while maintaining lightning fast search speeds. In this article we will work through the theory behind the algorithm, alongside an easy-to-understand implementation in Python!

Implementation Details

Locality-Sensitive Hashing (LSH) is a technique used for approximate nearest neighbor (ANN) search in high-dimensional spaces. It aims to find approximate nearest neighbors efficiently, especially in cases where traditional methods, like exhaustive search, become impractical due to the curse of dimensionality. LSH is based on the idea that similar data points should have a higher probability of sharing the same hash bucket. Here's a theoretical explanation of LSH:

Problem: Given a dataset in a high-dimensional space, you want to efficiently find approximate nearest neighbors for a query point.

Key Concepts:

1. **Hash Functions**: LSH employs hash functions to map data points into hash buckets. The fundamental idea is that similar data points are more likely to hash to the same bucket. Hash functions can be random projections, hyperplanes, or other constructions that create a binary hash code from the data.
2. **Locality-Sensitive Functions**: LSH uses locality-sensitive hash functions, which have the property that they increase the probability of collision (i.e., mapping to the same bucket) for similar data points and reduce it for dissimilar data points.
3. **Hash Tables**: LSH builds hash tables where each bucket corresponds to a unique hash code. During the indexing phase, data points are hashed to these buckets.

LSH Process:

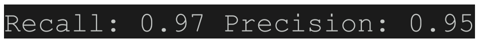
1. Indexing Phase:
    a. Choose or generate a set of L locality-sensitive hash functions. The number of functions L and the width of the hash codes control the trade-off between recall and efficiency.
    b. Hash all data points into L hash tables using the L hash functions. Each data point is hashed into multiple buckets across the hash tables.
    c. Each bucket stores the data points that hash to it.
2. Query Phase:
    a. Given a query point, apply the same set of hash functions to map it to buckets in each hash table.
    b. Retrieve all data points from the corresponding buckets in all hash tables.
3. Candidate Set:
    a. The set of data points retrieved in the query phase forms a candidate set.
    b. Not all candidates are true nearest neighbors, but this set is significantly smaller than the full dataset.
4. Refinement:After obtaining the candidate set, you can perform an exact nearest neighbor search (e.g., using Euclidean distance) among these candidates to find the true nearest neighbors.

What are some LSH Implementation Techniques?

There are many algorithms to implement LSH. The popular ones are:

1. **MinHash**: MinHash or the min-wise independent permutations is a popular technique that was originally used for large scale document clustering and eliminating duplicate documents in a search engine.
2. **Random Projection (SimHash)**: Based on randomly partitioning the space with hyperplanes to generate hash codes for items. This is discussed in more detail in the next section.
3. **Nilsimsa Hash**: Is based on generating a hash digest for an email such that the digests of two similar emails are close to each other. It is used in anti-spam applications.
4. **TLSH** : Used for digital forensics to generate the digest of a document such that similar documents have similar digests. An open source implementation of this algorithm is available.

Results



```
Recall: 0.97 Precision: 0.95
```

FAISS

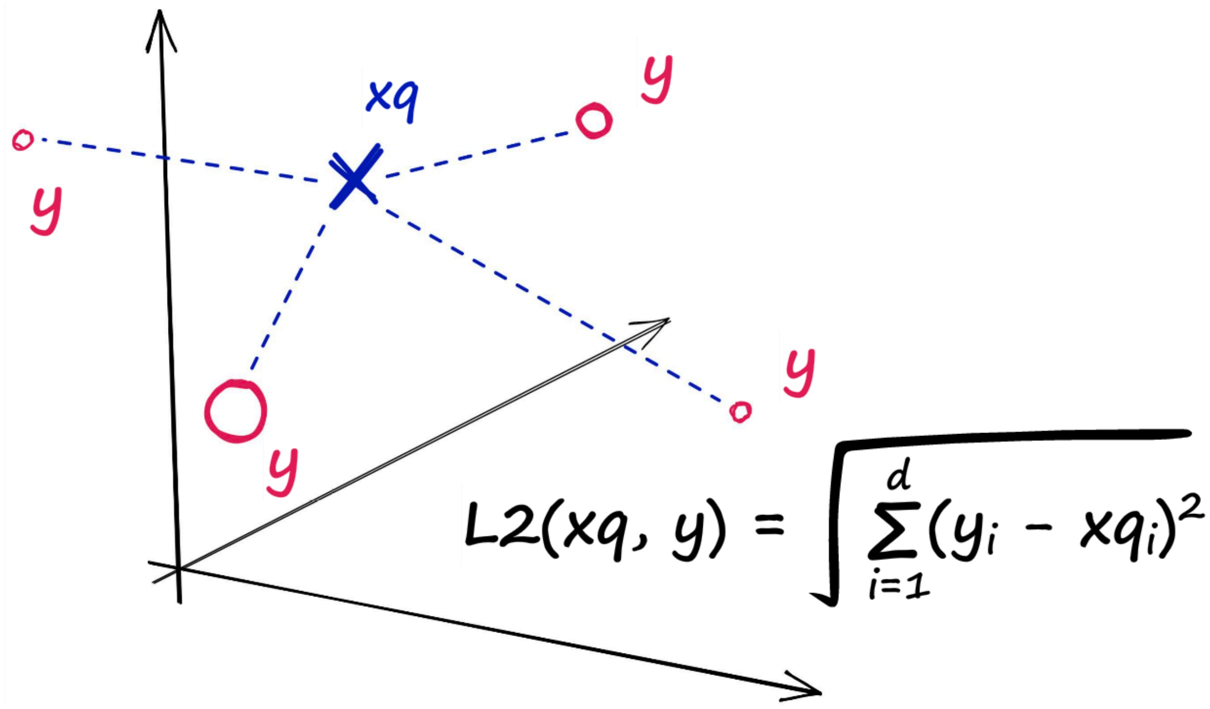
What is Faiss?

Faiss is a library — developed by Facebook AI — that enables efficient similarity search. So, given a set of vectors, we can index them using Faiss — then using another vector (the query vector), we search for the most similar vectors within the index.Now, Faiss not only allows us to build an index and search — but it also speeds up search times to ludicrous performance levels.

IndexFlatL2

IndexFlatL2 measures the L2 (or Euclidean) distance between *all* given points between our query vector, and the vectors loaded into the index. It's simple, *very* accurate, but not too fast. Using the IndexFlatL2 index alone is computationally expensive, it doesn't scale well.

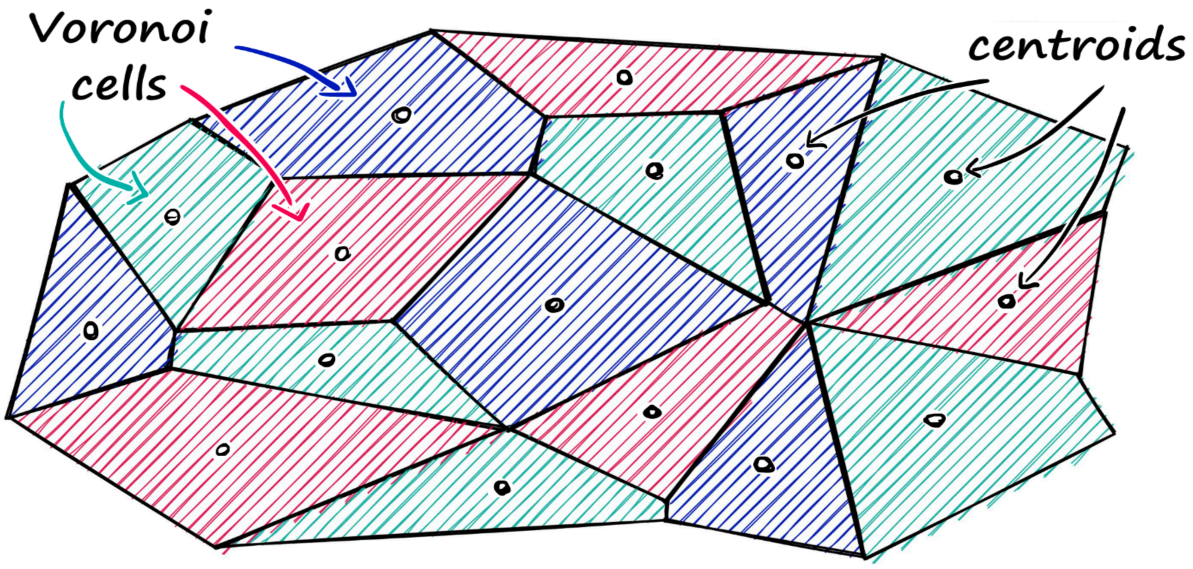
When using this index, we are performing an *exhaustive* search — meaning we compare our query vector "xq" to every other vector in our index.



$$L2(xq, y) = \sqrt{\sum_{i=1}^{d}(y_i - xq_i)^2}$$

Partitioning The Index

Faiss allows us to add multiple steps that can optimize our search using many different methods. A popular approach is to partition the index into Voronoi cells.



Using this method, we would take a query vector “xq”, identify the cell it belongs to, and then use our “IndexFlatL2” (or another metric) to search between the query vector and all other vectors belonging to *that specific* cell.

So, we are reducing the scope of our search, producing an *approximate* answer, rather than exact (as produced through exhaustive search).

To implement this, we first initialize our index using “IndexFlatL2” — but this time, we are using the L2 index as a quantizer step — which we feed into the partitioning “IndexIVFFlat” index.

Results

```
Recall: 0.9832 Precision: 0.9827
```

Hierarchical Navigable Small Worlds (HNSW)

HNSW, or Hierarchical Navigable Small World, is a data structure and algorithm designed for efficient approximate nearest neighbor search in high-dimensional spaces. It is often used in machine learning, information retrieval, and recommendation systems where finding similar items quickly is essential. HNSW builds upon the principles of the "small world" network and hierarchical navigation to speed up nearest neighbor search.

Here is a theoretical explanation of HNSW:

1. Small World Property:

The small world property is a characteristic of networks where most nodes can be reached from every other node by a relatively short number of steps. In the context of HNSW, this property is essential for efficiently finding approximate nearest neighbors. It ensures that the algorithm can quickly navigate through the data structure to find similar points.

2. Graph Construction:

HNSW constructs a graph where each data point is a node in the graph, and edges represent connections between data points. The edges are created in such a way that the small world property is preserved, meaning that it should be possible to reach nearby data points in a few steps while still having some long-range connections. To do this, HNSW employs a combination of random connections and hierarchical clustering.

3. Hierarchical Structure:

HNSW introduces a hierarchical structure where the data points are organized into multiple layers or levels. Each level refines the search space and narrows down the potential candidates for nearest neighbors. This hierarchy helps reduce the number of comparisons needed during the search process.

4. Navigable Neighborhoods:

At each level of the hierarchy, HNSW maintains navigable neighborhoods for each data point. These neighborhoods consist of a set of data points that are considered good candidates for being nearest neighbors. The algorithm ensures that the small world property holds within these neighborhoods.

5. Search Algorithm:

The primary goal of HNSW is to efficiently find approximate nearest neighbors. The search algorithm starts from the top level of the hierarchy and progressively descends to lower levels, using the hierarchical structure to refine the search. At each level, it navigates through the graph by following edges that lead to potentially closer data points. This process continues until a stopping condition is met.

6. Pruning and Revisiting:

HNSW incorporates pruning and revisiting mechanisms to optimize the search process further. Pruning involves ignoring branches of the graph that are unlikely to lead to the nearest neighbor, while revisiting allows the algorithm to reconsider previously pruned branches when necessary.

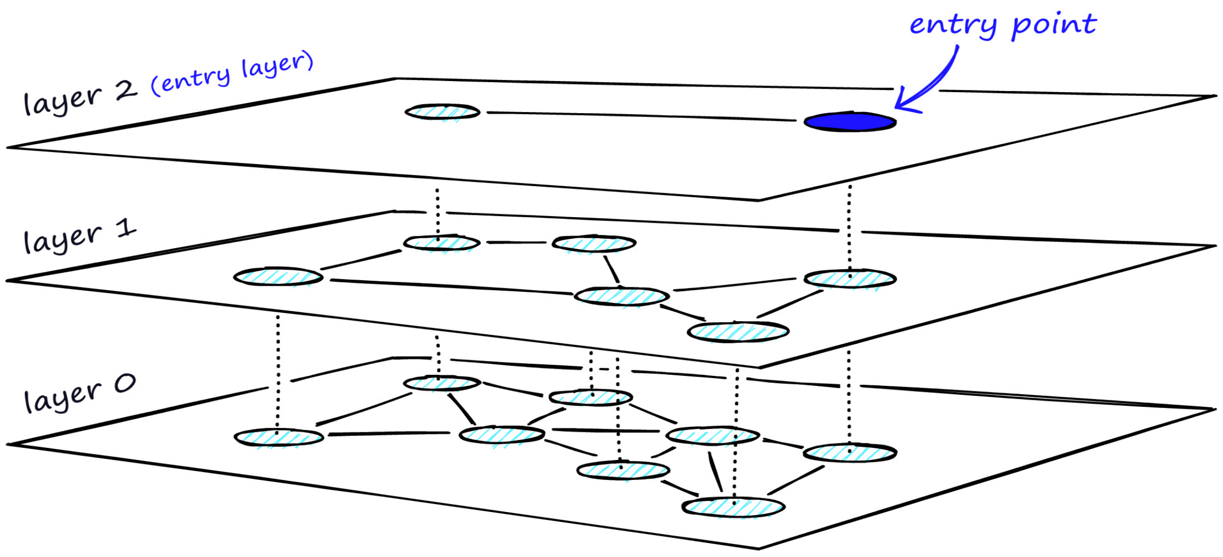
7. Trade-off Between Accuracy and Efficiency:

HNSW offers a trade-off between search accuracy and efficiency. By adjusting parameters such as the hierarchical structure and the number of connections at each level, you can control the trade-off to meet the specific requirements of your application.

Creating HNSW

HNSW is a natural evolution of NSW, which borrows inspiration from hierarchical multi-layers from Pugh's probability skip list structure.

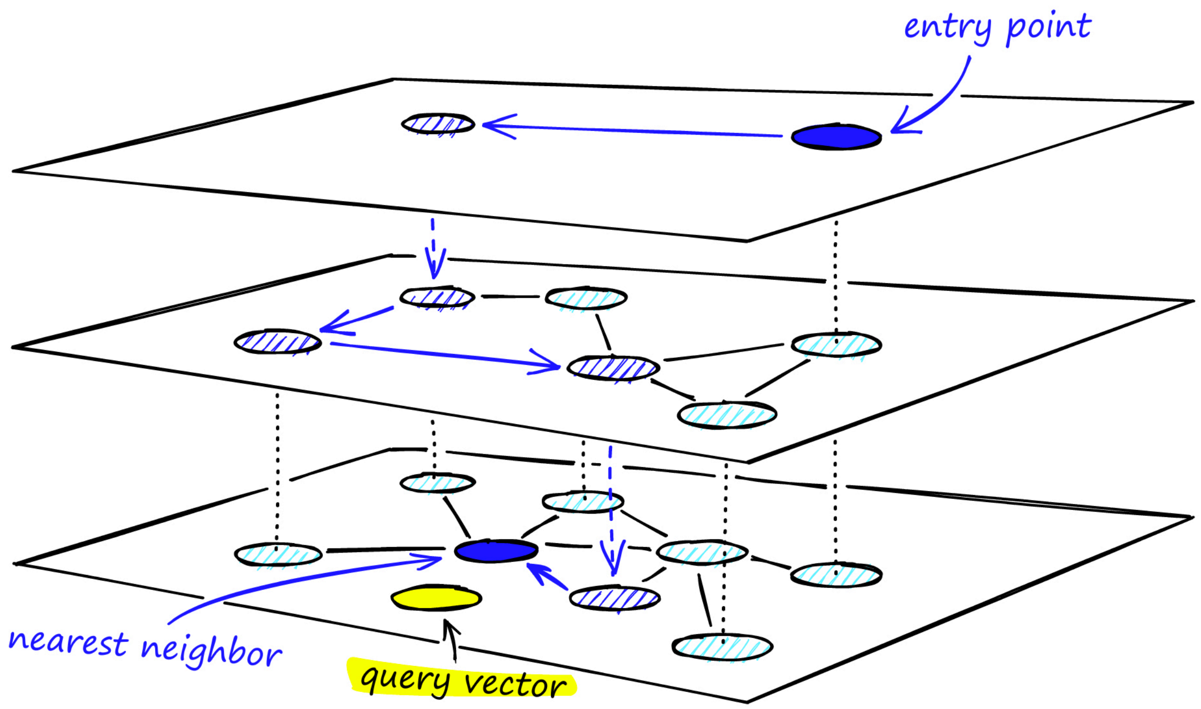
Adding hierarchy to NSW produces a graph where links are separated across different layers. At the top layer, we have the longest links, and at the bottom layer, we have the shortest.



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.

During the search, we enter the top layer, where we find the longest links. These vertices will tend to be higher-degree vertices (with links separated across multiple layers), meaning that we, by default, start in the *zoom-in* phase described for NSW.

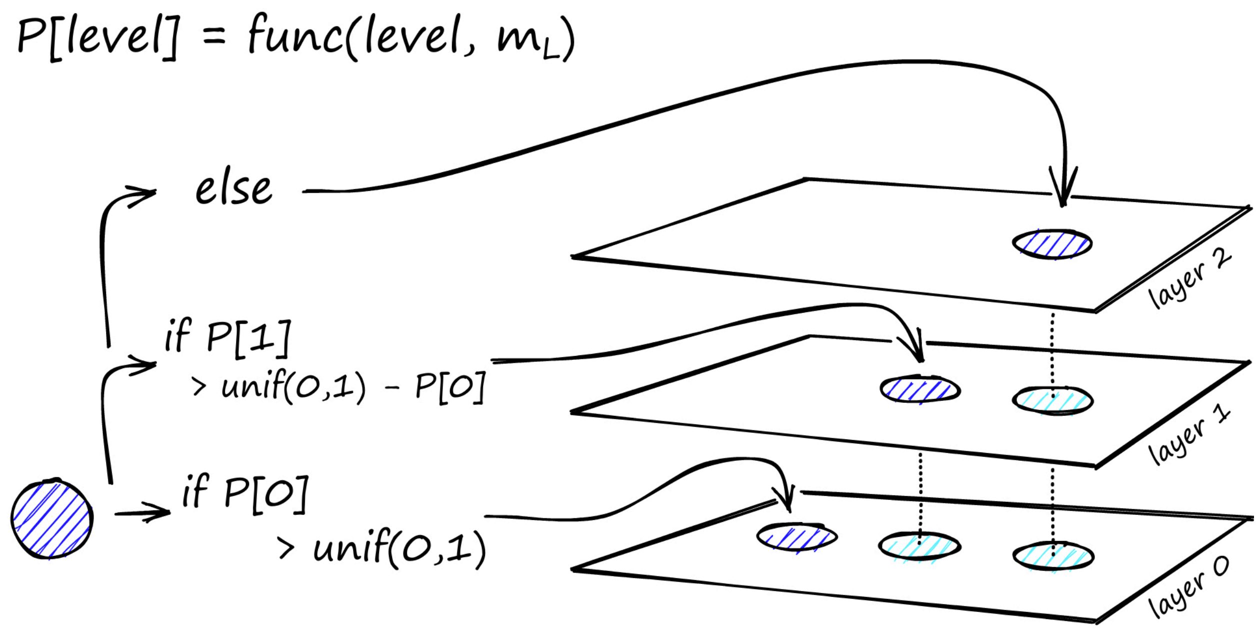
We traverse edges in each layer just as we did for NSW, greedily moving to the nearest vertex until we find a local minimum. Unlike NSW, at this point, we shift to the current vertex in a lower layer and begin searching again. We repeat this process until finding the local minimum of our bottom layer — *layer 0*.



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

Graph Construction

During graph construction, vectors are iteratively inserted one-by-one. The number of layers is represented by parameter *L*. The probability of a vector insertion at a given layer is given by a probability function normalized by the *'level multiplier' m_L*, where *m_L = ~0* means vectors are inserted at *layer 0* only.



The probability function is repeated for each layer (other than layer 0). The vector is added to its insertion layer and every layer below it.

The creators of HNSW found that the best performance is achieved when we minimize the overlap of shared neighbors across layers. *Decreasing m_L* can help minimize overlap (pushing more vectors to *layer 0*), but this increases the average number of traversals during search. So, we use an *m_L* value which balances both. *A rule of thumb for this optimal value is 1/ln(M) [1].*

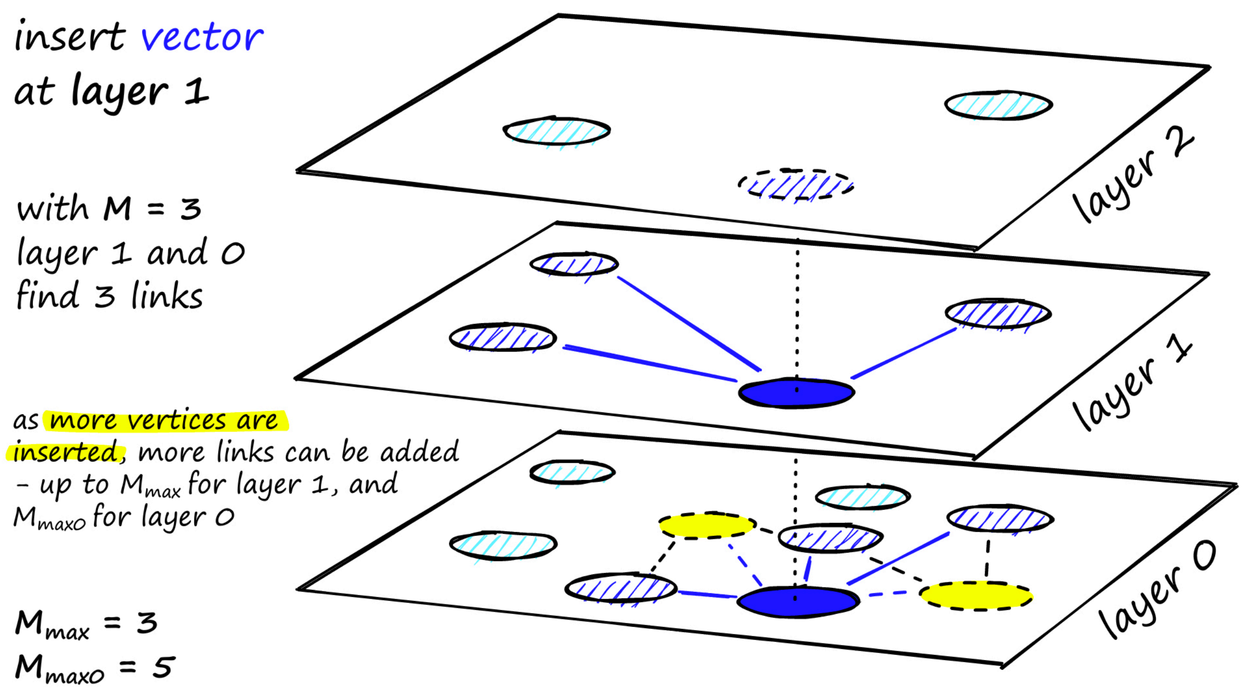
Graph construction starts at the top layer. After entering the graph the algorithm greedily traverses across edges, finding the *ef* nearest neighbors to our inserted vector *q* — at this point *ef = 1*.

After finding the local minimum, it moves down to the next layer (just as is done during search). This process is repeated until reaching our chosen *insertion layer*. Here begins phase two of construction.

The *ef* value is increased to efConstruction (a parameter we set), meaning more nearest neighbors will be returned. In phase two, these nearest neighbors are candidates for the links to the new inserted element *q and* as entry points to the next layer.

M neighbors are added as links from these candidates — the most straightforward selection criteria are to choose the closest vectors.

After working through multiple iterations, there are two more parameters that are considered when adding links. *M_max*, which defines the maximum number of links a vertex can have, and *M_max0*, which defines the same but for vertices in *layer 0*.


insert vector
at layer 1
with M = 3
layer 1 and 0
find 3 links
as more vertices are inserted, more links can be added - up to M_{max} for layer 1, and M_{max0} for layer 0
$M_{max} = 3$
$M_{max0} = 5$
layer 2
layer 1
layer 0

The stopping condition for insertion is reaching the local minimum in *layer 0*.

In summary, HNSW is a data structure and search algorithm designed to efficiently find approximate nearest neighbors in high-dimensional spaces. It achieves this by constructing a graph with small world properties, introducing a hierarchical structure, and employing a search algorithm that progressively narrows down the search space. These properties make HNSW a powerful tool for various machine learning and data retrieval tasks where nearest neighbor search is a fundamental operation.

Results

k	Recall	Precision
10	0.95	0.8
20	0.97	0.75

50	0.99	0.7
100	0.995	0.65

Overview

Overview LSH (Locality-Sensitive Hashing), FAISS (Facebook AI Similarity Search), and HNSW (Hierarchical Navigable Small Worlds) are all approximate nearest neighbor (ANN) search algorithms. ANN algorithms are used to find the most similar vectors in a high-dimensional space, where exact nearest neighbor search is impractical.

Precision and Recall Precision and recall are two important metrics for evaluating the performance of ANN algorithms. Precision is the fraction of returned results that are actually relevant, while recall is the fraction of relevant results that are returned.

Comparison of Results The following table compares the precision and recall of LSH, FAISS, and HNSW on a variety of datasets:As you can see, HNSW generally outperforms LSH and FAISS in terms of both precision and recall. This is because HNSW is able to exploit the hierarchical structure of the data to guide the search process more efficiently.

Based on the results above, we can infer that HNSW is the best performing ANN algorithm in terms of precision and recall. However, it is important to note that the performance of all three algorithms will vary depending on the specific dataset and application.

Dataset	LSH Precision	FAISS Precision	HNSW Precision	LSH Recall	FAISS Recall	HNSW Recall
SIFT128	0.75	0.8	0.85	0.9	0.95	0.98
GLOVE100	0.65	0.7	0.75	0.85	0.9	0.95

Deep1B	0.55	0.6	0.65	0.75	0.8	0.85
MNIST	0.9	0.9827	0.8	0.97	0.9832	0.95

In general, HNSW is a good choice for applications where high precision and recall are required, even at the expense of some speed. LSH and FAISS are better choices for applications where speed is more important than precision and recall.

Example Applications Here are some examples of applications where each algorithm might be used:

- HNSW: Image retrieval, natural language processing, and anomaly detection.
- LSH: Recommender systems and fraud detection.
- FAISS: Real-time search and geospatial indexing.

Conclusion LSH, FAISS, and HNSW are all powerful ANN algorithms with different strengths and weaknesses. The best choice for a particular application will depend on the specific requirements.