
Efficient KNN Search for Large-Scale Applications: Methodologies and Insights

TDT4501 - Computer Science, Specialization Project

Author:
Kristoffer Seyffarth

Advisor:
Knut Magne Risvik

Autumn 2024

Table of Contents

List of Figures	ii
List of Tables	iii
1 Introduction	1
1.1 Traditional K Nearest Neighbour (KNN)	1
1.2 Approximate K Nearest Neighbour (ANN)	1
1.2.1 Tree-datastructures	1
1.2.2 Graph-datastructures	2
1.2.3 Hashing for ANN optimization	2
1.2.4 Clustering for ANN optimization	3
1.2.5 Quantization for ANN optimization	3
1.3 Scaling ANN Algorithms	3
1.3.1 Billion Scale ANN algorithms	4
1.4 Further Reading	4
1.4.1 Metric Learning	4
1.4.2 DEC	5
1.4.3 Learned Indexes	5
1.5 Usage Areas	5
1.5.1 Retrieval-Augmented Generation (RAG)	6
1.5.2 Other Applications	6
1.6 Research question	6
2 Background	7
2.1 SOT ANN-BENCHMARKS	7
2.1.1 SCANN- Scalable Nearest Neighbour	7
2.1.2 HNSW	8
2.1.3 VAMANA (DiskANN)	8
2.1.4 Neighborhood Graph and Tree (NGT)	8
2.1.5 Pynndescent (NN-Descent)	8
2.1.6 Overview	8
2.2 SOT BIG-ANN-BENCHMARKS	9
2.2.1 DiskANN	9
2.2.2 FRESHDiskANN	9
2.2.3 Pinecone-ood	9

2.2.4	Zilliz	10
2.2.5	Puck	10
3	Proposal	10
3.1	Hypothesis	10
3.2	Implementation	10
3.2.1	Training	11
3.2.2	Index generation	11
3.2.3	Querying	11
3.3	Testing	11
3.3.1	Benchmark	12
3.4	Results	12
3.4.1	Analysis	13
3.4.2	Conclusion	14
3.4.3	Future Work	14
3.4.4	Code-space	14
3.5	Similar work	14
3.5.1	Matryoshka Representation	14
3.5.2	Adaptive ANN Search (AdANNs)	15
4	Summary	15
	References	16
	Appendix	17
A	AI Declaration	17

List of Figures

1	KD-tree partitioning	2
2	HNSW search illustration	2
3	Partitioning based on features	4
4	Contrastive loss function	5
5	The current ANN-Benchmarks leaderboard for the sift 128 dimensional dataset . .	7
6	The proposed implementation and the benchmark for dynamic and static datasets	12
7	<i>ANNANN</i> with and without <i>DEC</i> (<i>HNSW</i> added for context)	13

List of Tables

1	The runtime of the individual query stages	13
2	The memory required by each part	13

1 Introduction

K-Nearest Neighbour (KNN) algorithms are simple and widely applicable for many different types of tasks. They work by searching through datasets for similar cases, and evaluating the result based on the k nearest neighbors.

Typical use cases are classification and regression, however they are also applicable to many other kinds of tasks such as recommendation systems, RAG systems, and error detection. The main ability they have is to provide neighborhood information directly from a dataset, without generalizing the output such as Neural Networks do. For tasks that require actual points and not approximations, KNN algorithms might be the way to go.

1.1 Traditional K Nearest Neighbour (KNN)

Also known as linear scan, works by simply calculating a distance measure over N instances and D dimensions, and returning the closest. Through hardware optimization reasonable results can be achieved, however for large datasets, particularly in large dimensions, runtime is slow. This is because by nature they require all-to-all computation for each entry over all dimensions. This results in linear scaling for larger datasets making the runtime:

$$O(ND) \tag{1}$$

1.2 Approximate K Nearest Neighbour (ANN)

To improve efficiency you can use ANN algorithms, which approximate the locations of neighbours, sacrificing some accuracy for a great increase in efficiency. To do this they utilize different indexing methodologies that incorporate data structures such as trees, graphs, or clusters. These solutions often require storing the index in main memory for efficient search, but in return provide much faster search times over linear scan. However, the indexing structures only approximate the nearest neighbors, meaning they are not fully accurate as with linear scan. Often more recall can be achieved with larger and more complex indexing structures, at the cost of speed and storage. Therefore the most common way to compare these algorithms are the amount of queries per second for a given precision, with a recall ≥ 0.95 often considered satisfactory for most tasks.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

1.2.1 Tree-datastructures

Tree based approaches was first introduced by [1]. with a way to partition dimensions into tree data-structures called KD-trees. It worked by creating a tree where each layer corresponds to a dimension, and each node is a partition based on the values of an instance in the dataset. By doing this you could search a tree for close neighbours over all dimensions, resulting in the true nearest neighbour for low dimensions.

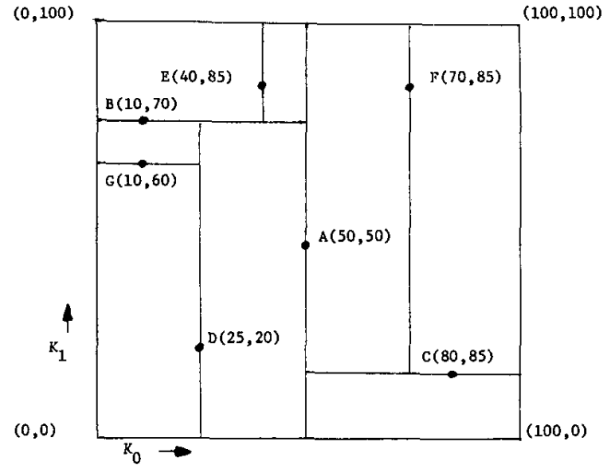


Figure 1: KD-tree partitioning

However, for higher dimensions partitioning becomes less impactful as there are a greater chance the query is on the borders of one of the partitions. This means for accurate searches, the KD-trees algorithm must backtrack across the tree and search more leaf nodes in order to ensure precision. This reduces efficiency for high dimensions, but KD-trees is still used for precise searches over low dimensions when backtracking is not needed. Newer tree based approaches are generally optimized for less backtracking, making them more efficient but less accurate.

1.2.2 Graph-datastructures

Graph based approaches construct graphs, often with multiple layers of complexity called a Hierarchical Navigable Small World, or *HNSW*. During search you initially query a random point, and find neighbors from the graph. Then you greedily select the closest of the neighbours, while increasing the complexity of the selected graph by diving into deeper layers 2, until the closest neighbour is approximated. Currently these types of ANN algorithms are generally considered the most efficient for most datasets, as they are able to achieve high recall and efficiency with little overhead [2]. Recent advances have also utilized lower amounts of more optimized layers, reducing memory related drops in efficiency for large indexes [3].

Vector Search

Hierarchical Navigable Small Worlds (HNSW)

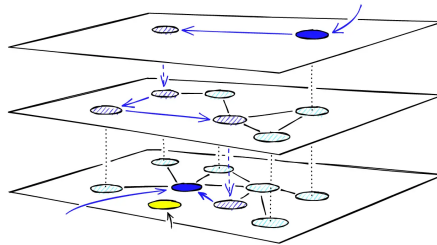


Figure 2: HNSW search illustration

1.2.3 Hashing for ANN optimization

Hash based approaches work by bucketing similar hashed indexes together, thereby reducing the search space substantially as you would only need to search the closest hashed bucket of entries. The

key is to generate accurate hashing functions, which have higher probability to hash close instances equally. To improve accuracy, multiple hash functions can be used, however at the expense of speed and storage. Hash functions are often used as a supplementary search mechanism.

1.2.4 Clustering for ANN optimization

Similar to how hashing works, you can also cluster similar datapoints together, and search only the closest clusters drastically reducing the amount of points needed to query. A common way to do this is with a clustering algorithm like K-folds, or Gaussian Mixture Models (GMM), which cluster the closest neighbours together. Some of the currently leading ANN algorithms utilize efficient dot product clustering 2.2.3.

1.2.5 Quantization for ANN optimization

Product Quantization (PQ) is a technique used in approximate nearest neighbor (ANN) search to efficiently handle high-dimensional data, especially in large-scale datasets [4]. PQ operates by dividing the original high-dimensional space into multiple lower-dimensional subspaces. Each subspace is then quantized separately using a dedicated codebook of representative vectors. This approach allows for compact storage of vectors and facilitates rapid distance approximations during queries.

During a query, PQ approximates the distance between the query vector and dataset vectors by combining precomputed distances from each subspace’s quantized representations. This method significantly accelerates similarity computations without the need to reconstruct full high-dimensional vectors. PQ is particularly effective for large datasets with high-dimensional features, as it enables scalable storage and fast ANN searches. However, the quantization process introduces approximation errors, which can affect retrieval accuracy. To address these challenges, several enhancements have been developed:

- **Optimized Product Quantization (OPQ):** Enhances PQ by applying a rotation to the data, better aligning it with the subspaces and improving quantization accuracy.
- **Additive Quantization (AQ):** Extends PQ by allowing more flexible combinations of codewords, further refining approximation quality.

Despite its advantages, maintaining high accuracy in PQ can be challenging as dataset sizes grow. To mitigate this, modern ANN algorithms often combine PQ with other indexing structures, such as tree-based or graph-based methods. This hybrid approach leverages the strengths of PQ in providing compact and efficient representations while utilizing additional structures to organize and navigate the search space effectively, resulting in improved query performance and scalability.

1.3 Scaling ANN Algorithms

All approaches for KNN search have certain drawbacks. They might excel for specific implementations, while not doing so in others. A common thread is their ability to improve recall at the expense of efficiency, and vice versa. Another constraint they all have is the additional size of the data required by the generated index. Searching through larger datasets does in other words not only require more time, but also more memory. Modern ANN algorithms are often constrained by available Random Access Memory for large indexes, which is expensive and therefore not freely available. For large billion scale datasets, the indexes will not fit within a single computer. This means you will either have to run multiple machines concurrently, creating overhead and requiring additional hardware, or you can read the index from disk which might reduce the efficiency of the algorithm substantially.

This is a problem in a world where recent advancements in Deep Learning and Generative AI seem to generate better result on ever increasing amounts of data [5]. To be able to stay relevant and to enable large scale implementations, KNN algorithms must be able to efficiently and effectively search large amounts of data, without requiring specialized hardware to do so. Recent advancements use multiple different techniques in order to reduce the memory footprint of the algorithms, while utilizing smart memory storage techniques and Product Quantization 1.2.5, optimizing for faster query times both theoretically and practically. However these approaches also have their limits, and are often somewhat dependent upon the type of data and hardware being used. Often combining different approaches, combining strengths, results in better algorithms. Therefore alternative approaches for extremely large scale KNN that can be combined with existing ones, would be highly beneficial.

1.3.1 Billion Scale ANN algorithms

The simplest way to solve the constrained main memory issue is to simply read from disk instead. The main problem is that most algorithms rely on multiple reads to memory during search, which reduces efficiency when memory reads are expensive. To solve this problem, researchers from Microsoft proposed in 2019 *DiskANN* [3], which could search billion scale data with 1000 queries per second and recall 0.95, requiring only 64GB main memory, storing the rest of the index on disk. To do this they proposed *VAMANA*, which creates highly efficient one or low layer graph structures, which are optimized for faster disk read times. It works by placing neighbours in the index more closely together in disk, which results in lower read times as sequential reads perform significantly better than random access [6]. Every two years there is a competition called *NeurIPS* for large scale ANN search, and in 2001 the competition was aimed at billion scale where they utilized *DiskANN* as one of the baselines [7]. The results were varied, but on most datasets no algorithm outperformed *DiskANN* by a significant margin.

One problem with *DiskANN* is how it does not support dynamic datasets, requiring alterations to the index structure for every alteration in the data. To solve this, some researchers behind *DiskANN* proposed [8], which was aimed at dynamically changing datasets. It achieves sufficient speed and recall, while being able to dynamically alter the index at runtime. They utilize many different techniques, but the main one is to label changes in the graph structure with the most recent change. During search only the most recently changed nodes are visited, increasing recall. During the most recent *NeurIPS* competition in [9], *FreshDiskANN* was used as the baseline for the dynamically changing dataset track, which as of this paper still has not concluded with results. Unfortunately while it is designed to handle dynamic datasets, it is still computationally heavy to alter the index at runtime, and might cause delay issues for heavy loads.

1.4 Further Reading

1.4.1 Metric Learning

Distance metric learning is a branch of machine learning that aims to learn distances from the data, which enhances the performance of similarity-based algorithms [10]. It is most commonly used for a subset of classification tasks, where the instances given to predict are not the same as the ones given during training. The goal is to create a distance function which more closely corresponds to the distribution of categories.

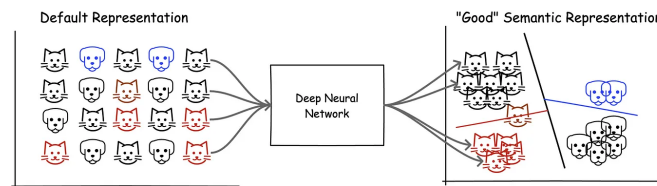


Figure 3: Partitioning based on features

Similarly you have *deep metric learning*, which essentially does the same as *Distance Metric Learning*, only within a learned embedding space. It works by inputting the labels of the input in addition with the vectors for the embedding. By inputting multiple points at the same time, and with the information saying either "these are similar" or "these are different", the embedding model learns not only the vector representations, but also the similarity and dissimilarity of instances. An example of this is contrastive loss [10].

$$\mathcal{L}_{\text{contrastive}} = (1 - y)\frac{1}{2}d^2 + y\frac{1}{2}[\max(0, m - d)]^2$$

Figure 4: Contrastive loss function

It works by comparing two entries, giving a higher loss function for similar entries far apart, and dissimilar entries close together. However for multi class classification task, contrastive loss lose accuracy, and it is therefore better used for binary classification [10]. Furthermore it can be subject to something called *model collapse*, where not enough features are extracted resulting in all points clustered together.

Another deep metric loss function is triplet loss, which works similar to contrastive loss, however by comparing three instances, two positive and one negative. That way it better clusters close entries together while keeping dissimilar entries further away. All of these methods are supervised, meaning you have a label for each instance, and this label is used for the distance metric.

1.4.2 DEC

Unsupervised Deep Embedding for Clustering Analysis was first proposed in 2016 [11]. It works by training an autoencoder to embed into a lower dimension, and then use centroids produced by KMeans in the embedded space for further aligning the autoencoder to those clusters. More specifically it generates a soft assignment over the clusters, for each entry using student-t distribution. Then during training it tries to minimize the Kullback-Leibler divergence, which is the overall entropy or difference between two distributions. The result is a vector space more closely correlating the inputted clusters. As a result better clustering performance over KMeans was achieved. In some regards it can be seen as an unsupervised deep metric learning technique, that instead of using labels use the clustering assignments.

1.4.3 Learned Indexes

An area of current research is learned index structures [12]. It has been shown to be able to efficiently generalize tree structures into neural networks and outperform traditional tree structures in some regards, especially regarding accuracy. Still, there are many drawbacks with this approach. Generally, utilizing neural networks instead of more rigid structures introduces the possibility of tuning and over fitting errors. Furthermore there is a certain training overhead, especially for large models. For large datasets, the model also has to be able to differentiate all entries, meaning the model complexity increases alongside the size of the data its meant to index, reducing efficiency. Furthermore, altering the dataset its meant to index requires retraining of the index, adding overhead and computational cost.

1.5 Usage Areas

Traditional usage areas for KNN search is for regression and classification tasks, where labels are generated based on the labels of the K nearest neighbours. As Machine Learning has been able to excel for those kinds of tasks, KNN search is now more applicable in other areas.

1.5.1 Retrieval-Augmented Generation (RAG)

RAG is an area that showcases one of the major benefits of KNN algorithms. It works by enhancing pre-trained models with additional continuously updated information. For example, when using LLMs, queries could be parameterized into a vector space where a dataset of similarly parameterized text documents could be searched using KNN, and then the most similar documents could be inserted into the LLM together with the query providing updated information while still utilizing the linguistic knowledge of LLMs. One could therefore look at this as the LLM containing the constant linguistic knowledge, and the corresponding dataset containing the relevant, dynamically changing, domain knowledge.

Through a system like this, the language model could be reused across domains, not requiring post-training for a certain domain. Other types of models could also be used, such as for computer vision. The one constant is the KNN algorithm, which could be used for any one of these implementations.

Furthermore such systems have proven to give similar results as extremely large LLMs such as GPT4, but requiring far smaller language models and computational power [13]. This is because models such as GPT have all their knowledge inside a black box consisting of billions of parameters, which all need to be queried in order to obtain the relevant information. Compare this to a model powered by RAG, which would only need enough parameters to accurately represent a format of communication such as text, images, or speech, and getting the other information through a KNN search in a relevant database. This is far more energy efficient, and showcases how RAG and KNN could not only solve the issue of continuously updated information for generative models, but possibly also the ever increasing power and GPU consumption of GenAI.

1.5.2 Other Applications

As KNN algorithms are versatile, they have multiple different applications. One example is for recommendation systems, where similar cases to previous history can be used for recommendation. For example Spotify have been using the tree based ANN algorithm *ANNOY* for the past decade, however recently switched over to the *HNSW* based *VOYAGER* [14]

Other applications could be anomaly detection, where anomalies often will find themselves isolated from neighbors. This can again not only be used for processing of data, but also fraud detection in cybersecurity, disease detection in medical services, or optimization of maintenance for anything from construction to web-services. You also have data imputation, where missing values in a dataset can be filled based on the nearest neighbors.

1.6 Research question

As we have seen, KNN algorithms are widely applicable, simple, easy to use and understand, and provide many of the capabilities that you lose when generalizing data such as through machine learning. We have seen that the application of KNN algorithms serve a purpose, and that there are areas where efficient KNN algorithms can outperform more advanced and computationally heavy machine learning approaches. We have also seen that the field of KNN search is continuously changing, as new research and better algorithms are continuously made. However there are a set of constraints around these types of algorithms, particularly around scaling. In order to open up new possible areas of application, and to stay competitive for implementations of large amounts of data, new search-mechanism could be explored.

How can dynamic dataset handling and memory efficiency be improved for billion-scale KNN algorithms while maintaining high recall and query speed?

2 Background

To better understand the differences and nuances of ANN algorithms, the strengths and weaknesses of different implementations, a subsample of the highest ranked algorithms for different datasets have been selected for further exploration.

2.1 SOT ANN-BENCHMARKS

ANN-Benchmarks [2] is a github repository containing benchmarking tools capable of comparing multiple different KNN algorithms to eachother over many different configurations and datasets. It also maintains an updated leaderboard of the best performing algorithms for different datasets.

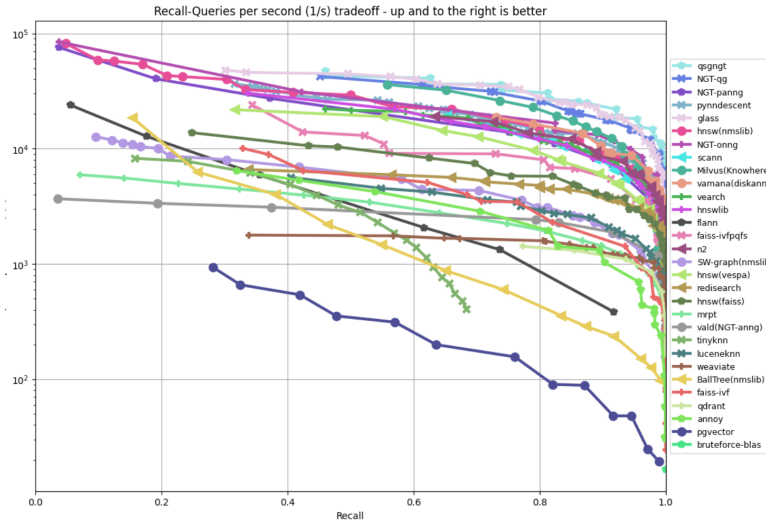


Figure 5: The current ANN-Benchmarks leaderboard for the sift 128 dimensional dataset

2.1.1 SCANN- Scalable Nearest Neighbour

SCANN is an ANN algorithm created by Google for high dimensional vector similarity search [15]. It can be described as a three step process:

1. **Partitioning** (optional)
2. **Scoring**
3. **Rescoring** (optional)

Partitioning is used when the dataset is large, and essentially clusters the data into subsets, where only the closest clusters are queried when used. For smaller datasets partitioning is not necessary.

Scoring Is the process of assigning a distance measure to all the queries in the clusters selected after partitioning. For small datasets brute force works best, however for large sets an approximate hashing function can be used to create approximate values.

Rescoring is the final step, often used in conjunction with the approximate hashing functions as they can provide the K approximate closest entries, for then to use brute force on this subset only and returning the $k \leq K$ closest entries.

2.1.2 HNSW

HNSW is the baseline for most other graph-based implementation due to its efficiency, scalability, flexibility of implementation, and reliability across datasets. It has been implemented and tested in multiple libraries, such as *nmslib*, *hnswlib*, and *faiss* 5.

2.1.3 VAMANA (DiskANN)

A newer and more efficient graph-based implementation is *VAMANA*, which is the index generation algorithm used in *DiskANN* [3]. It is designed to generate a single layer graph structure, instead of multiple as *HNSW* do. *Vamana* was created as the researchers behind *DiskANN* noticed switching layers during search required additional reads, and therefore for implementations where the index structure is on disk the multilayer graph structure was inefficient.

Furthermore by carefully assigning neighbors in the index structure closely together in memory, more efficient memory optimization could be achieved, even for implementations with the index in main memory, due to hardware optimization such as caching. because of this, *VAMANA* is generally seen as a more efficient implementation over *HNSW*, however with less flexibility.

2.1.4 Neighborhood Graph and Tree (NGT)

NGT [16] is optimized for large scale datasets up to millions of instances. It utilizes both a graph and a tree datastructure during search. The tree can exclude large parts of the datasets for search, then to switch to the graph structure for more fine grain control.

Multiple configurations are implemented and can be selected depending on the specific usecase. For example quantization, which is used to lower the bitwise resolution of the datapoints, decreasing the overall memory required by the index, and slightly increasing efficiency with only a marginal cost in accuracy.

2.1.5 Pynndescent (NN-Descent)

Another graph based ANN algorithm is *Pynndescent*, a version of NN-Descent implemented in Python [17]. It works similarly to *HNSW*, however with a single layer graph structure. During creation of the graph, multiple iterations are used in order to optimize the structure for efficiency and accuracy.

This makes the structure highly efficient, pruning edges not contributing to the efficiency of the search. Utilizing only one layer graph structure also reduces the amount of memory required by the index. However it also means that for dense datasets with nodes close together, additional computations are needed during search, as it is not able to take larger initial steps like *HNSW* is. Still it seems to slightly outperform *HNSW* on the Sift dataset 5

2.1.6 Overview

All of these algorithms primarily utilize graph structures for efficient search. Their implementations differ, and the result are slightly different scores for this particular dataset 5. For other sets the ordering are slightly different, however generally these algorithms outperform the others.

One important distinction is that the implementation of an algorithm matter for this kind of comparison, especially because the top performers are so clustered. For example *HNSW* has multiple entries, but their positions are spread out. The best performing one is *nmslib*, but the header-only version of the same library *hnswlib* perform slightly worse. There could be many reasons for this, however most likely *nmslib* is slightly more optimized for efficiency for example through hardware cache optimization. Even further down the list you find *HNSW* through *vespa*

and faiss, which prove that the individual implementations of the different algorithms can affect their standing on the leaderboard.

Another important aspect to consider involves metrics not depicted in this chart, specifically the scalability of the algorithms, their memory requirements, and the impact on recall in dynamic settings where the index structure is continuously updated. For instance, *VAMANA* 2.1.3 is optimized for scalability; however, its recall and efficiency may decline over time as the index is modified. This degradation is attributable to *VAMANA*'s reliance on hardware memory optimizations, such as disk allocation. Another example is *NGT*, which is currently the top-performing algorithm as shown in Figure 5. *NGT* employs both graph and tree structures in combination with quantization techniques. This dual-structure approach introduces additional overhead when altering the index, as both the graph and tree must be updated simultaneously.

2.2 SOT BIG-ANN-BENCHMARKS

BIG-ANN-BENCHMARKS is a github repository created by the NeurIPS competition track for large scale KNN algorithms [7] [9]. It mainly concerns scaling algorithms over simple efficiency as with ANN-BENCHMARKS 2.1.

2.2.1 DiskANN

Proposed in 2019 [3] *DiskANN* made a significant leap for large scale KNN algorithms. It showed that it was possible to efficiently index and search vast billion scale datasets on normal hardware, opening up new possibilities and areas of implementation for KNN algorithms not previously attempted.

The major component of *DiskANN* is *VAMANA* 2.1.3, which is designed as a highly optimized one layered graph structure indexing algorithm. It enables *DiskANN* to store the majority of the index on disk, while main memory can be reserved for caching the recently used parts of the index in order to speed up the search.

2.2.2 FRESHDiskANN

A major problem with *DiskANN* is how it does not support dynamic datasets. This is a problem when the search is highly dependent upon the overall structure of the index. To solve this, the researchers behind *DiskANN* proposed [8], which is aimed at dynamically changing datasets, also called streaming. It achieves sufficient speed and recall, while being able to dynamically alter the index at runtime. They utilize many different techniques, but the main one is to label changes in the graph structure with the most recent change.

2.2.3 Pinecone-ood

The current leader for OOD track on [9] is *Pinecone-ood*. It's not open source, so there is limited information about its implementation. Based on the available information, it utilizes three main steps:

1. Cluster entries with high dot product
2. Train a graph structure based on the k-nearest neighbours within a test set
3. Quantize the entries optimized for SIMD operations

During search select the clusters with the highest dot product, and add the corresponding neighbours from the graph, before doing an exhaustive search returning the k closest entries

At the time of writing this paper, Pinecone-ood is currently outperforming *DiskANN* at recall > 0.9 with a factor of about 9.5. It should be mentioned the 2023 competition [9] is meant for large ann algorithms only, in order to enable more researchers to compete. The largest dataset is 30 million entries, with the OOD track having only 10. Compared to the billion-scale competition in 2021 [7] this is quite small. It is therefore unknown how it would scale to larger datasets.

2.2.4 Zilliz

Zilliz is another competitor with an algorithm that is not open source. It is closely trailing *Pinecone* for two tracks, and leading one. However even less information is given about the specific implementation of this algorithm.

2.2.5 Puck

Also successfully able to query billion scale datasets efficiently, *PUCK* achieved good performance at billion scale benchmarks in 2021 [7], and the winner of the 2023 streaming track for practical vector search [9], increasing their own performance by 70%

3 Proposal

The proposed implementation attempts to reduce computational time, optimize memory efficiency, and support dynamic data adjustments in KNN search. Many of the most efficient ANN algorithms utilize lower dimensionality or memory reduction techniques like Product Quantization for more efficient computations. The proposal is to use an autoencoder for a learned lower dimensional search, which is a type of neural network used to learn efficient data embeddings in an unsupervised manner, and will for the sake of this paper be named *ANNANN*.

3.1 Hypothesis

The idea is to generate an index structure for an ANN algorithm that utilizes the general patterns of the individual dataset as a feature for better placement, search, and memory optimization. It could potentially solve issues related to scaling indexes for large datasets, together with problems related to decreasing recall for streaming tasks. The theory is that generating an index structure based on patterns in the dataset rather than a random distribution would make the structure itself more relevant, even after heavy modification of the dataset. This assumes that entries in the dataset are somewhat correlated over time.

To be able to test this theory, while ensuring it still reflects the original idea of being able to scale for large datasets, it has to work within some constraints.

- **Insertion and deletion-** It should support simple insertions and deletions, without requiring alteration of the index structure.
- **Memory constraint-** It should consider the current limitations of large scale ANN algorithms, that the index structure might exceed the capabilities in memory of the machine. Therefore it should be able to store parts of the index in disk, while maintaining as few reads as possible.

3.2 Implementation

Instead of generating a tree or graph structure that fits these constraints, and that is aimed at the patterns of the dataset, a simpler approach could be to generate a vector space which is based on

these patterns, and then create a regular structure within it. Additionally, instead of utilizing *PQ* for instance based memory reduction, an *autoencoder* could reduce the dimensionality by generating a learned lower dimension, reducing the dimensional memory required. It is an unsupervised neural network which is trained on the entries in a dataset in order to as efficiently as possible reduce the dimensionality of the entry and revert it back, with as little loss as possible. This gives the model an hourglass shape. After training the model is cut in half in the middle, where the first part becomes the encoder, and the second the decoder.

Therefore, *ANNANN* utilizes three main components.

- **Autencoder**, to generate a learned lower dimensionality search space.
- **clustering**, using *KMeans* for small or *MiniBatchKMeans* for larger datasets.
- **ANN algorithm**, using *HNSW* 2.1.2 for the initial cluster search

3.2.1 Training

The *autoencoder* is trained over two stages, where the first simply learns the embedded space, and the second adds clustering information using DEC 1.4.2. A *KMeans* or *MiniBatchKMeans* is used to *cluster* within the embedded space, where the clusters are used as a reference during the second stage to better place learned entries within the clusters.

3.2.2 Index generation

Each entry is encoded in order to find the closest embedded cluster. Then the original vector is added to the batch corresponding to that cluster. To increase efficiency for large scale cluster search, *HNSW* is fitted on the cluster centroids. This means that the main search component of *ANNANN* should not only perform equally well after extensive edits to the dataset, but the search itself is also in a lower dimension. This should increase the efficiency and reduce the memory required by the *HNSW* index, in addition to the reduction in size as a result of clustering.

3.2.3 Querying

The query has three main steps.

1. Encode the query and use *HNSW* to find the $C * 10$ closest clusters in the embedded space
2. Use linear scan to find the C closest clusters in the input space
3. Use linear scan on all entries within the C to find the k closest neighbours

A problem with the second stage is that it requires the cluster centroids in the input space, which requires using decoder on all $C*10$ clusters. Using the encoder for a singly query did not result in a major contribution to the query time, however $C*10$ computations for each query is substantial. The solution to this is to precompute all the decoded cluster centroids when generating the index, and store them for quick lookup during search. This requires marginally more memory, but results in much better efficiency. Potentially for situations where memory is constrained, only the most recent clusters could be cached in memory, and queries requiring additional centroids could either use the decoder or read them from disk, depending on the implementation.

3.3 Testing

In order to test the algorithm, it will be compared to a Benchmark not utilizing a lower dimensionality or DEC for search. Furthermore a simulated dynamic instance is used, where the autoencoder

and cluster centroids are generated utilizing only the first half of the dataset. This will simulate an instance where the dataset have doubled in size since the last time the underlying index structure has been changed, other than simple insertions and deletions.

Dataset - sift1m (128 dimensions, 1 million entries)

Parameters:

- **N_Clusters** - 50 000
- **Autoencoder** - 4 Layers
- **Encoding Dimensions** - 32 (75% reduction)
- **Clustering Algorithm** - *MiniBatchKMeans*
(Max_iter:100,batch_size:10000, max_no_improvement:10)
- **ANN Algorithm** - *HNSW* (ef_build:200, ef_search:100, M:16)

3.3.1 Benchmark

As the algorithm is written in Python, even with Tensorflow and Numpy optimization, the implementation language could still affect the results. Additionally it is the theory that should be tested, not necessarily the implementation itself, so testing against SOT algorithms is not as relevant. Instead a similar benchmark algorithm is constructed which utilizes the same index generation and querying over the first and second step, only in the original space not the embedded. This will better show the differences in search resulting from a learned vector space.

3.4 Results

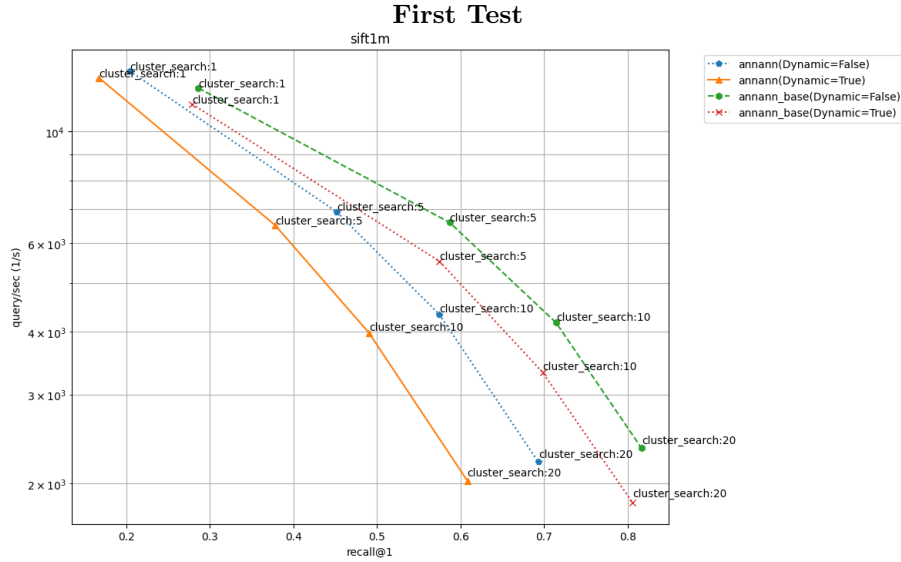


Figure 6: The proposed implementation and the benchmark for dynamic and static datasets

Second Test

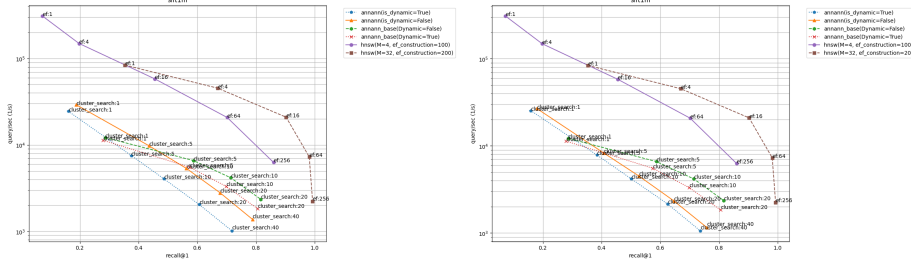


Figure 7: ANNANN with and without DEC (HNSW added for context)

Runtime

	Encode Queries	Cluster Search	Linear Scan
ANNANN	0.003	0.33	0.667
ANNANN_BASE	0	0.20	0.80

Table 1: The runtime of the individual query stages

Memory Consumption

	Index (MB)	HNSW (MB)	Other (MB)	Sum
ANNANN	525	13	31	569
ANNANN_BASE	525	31.5	24.4	581
percentage %	100	0.41	1.27	0.98

Table 2: The memory required by each part

3.4.1 Analysis

As expected the ANNANN algorithm is both less efficient and accurate than the baseline 6. This is probably in some part due to information loss after dimensionality reduction, however mostly because of the computational overhead of encoding the queries and multiple cluster search stages. The majority of time is spent on linear scan 1, which is expected. Furthermore both algorithms have a small drop in recall for the dynamic instance, which means ANNANN does not provide the expected benefit of recall retention theorized. It actually seems like the discrepancy between the static and dynamic sets for ANNANN is slightly larger than the for the base case, which is surprising. Additionally it looks like ANNANN retains efficiency when increasing the search area, however the recall drops somewhat. Meanwhile the base case retains the recall, while the efficiency drops.

This is interesting as the only difference between the static and dynamic implementation of the base is the amount of entries available while generating the index clusters. The most likely explanation is that the variation of entries in each cluster increase for a dynamic set, and the largest clusters are more often searched, reducing the efficiency without increasing recall. Why recall drops for ANNANN is uncertain, however most likely it is a result of a less accurate autoencoder. Training the encoder and decoder on the entire vector space seems to make it more accurate at predicting the correct query embeddings, and the correct cluster positions in the original space. Hence we can say that ANNANN is better at distributing new entries among the different clusters, while less accurate placing and localizing the nearest neighbors in those clusters.

It is possible that reducing the amount of entries available for DEC, reduces the predictive capabilities of the autencoder which is why we experience a drop in recall when multiple clusters are selected. To test this theory a second trial was run 7, this time with two versions of ANNANN, one with DEC and one without. Its clear that using DEC improves the performance in the static instance, and reduces it in the dynamic one, without changing the efficiency. This means that DEC does not work well on unseen data, as it seems to over-fit the clustering based on the initial data, and not generalize as well as expected.

3.4.2 Conclusion

The original theory that the discrepancy between dynamic and static recall for *ANNANN* would be less than for a base case, seems to be false. Instead it would seem like clustering using *DEC* within the learned embedded space produces a better distribution of entries. However it also seems like it reduces the predictive capabilities for dynamic sets, at least for this current implementation. One more interesting observation is that historically while implementing and testing *ANNANN*, the efficiency and recall actually seemed to increase when reducing the embedding dimension. That this would increase queries per second is obvious, however why the recall also increases is unknown. It is possible that *DEC* clustering works better in lower dimensions for the selected dataset, which could be something worth looking into as lower dimensions are beneficial for KNN search. Nonetheless the benefits of utilizing *DEC* for this kind of search is minimal.

3.4.3 Future Work

The major drawback of the current implementation of *ANNANN*, is how it relies on linear scan in the original space for the second and third query stage. This not only reduces the efficiency substantially, but also means the size of the index remains the same as for the base case. The reason is because it is not able to effectively search individual entries within the embedded space, meaning it ultimately requires storing and searching the original vectors. This is a result of *DEC*, as its main objective is to cluster entries together, not to make sure those entries remain proportional to each-other. A possible solution to this is to generate a custom loss function based on *DEC*, which is more oriented around KNN search rather than just clustering. Theoretically you could have a loss function which increases when the k closest entries are not in the same cluster, as this would help during search. Furthermore it could also better capture local structure of each cluster, meaning the closest entries remain close even after embedding. The result could be a lower dimensionality search space capable of being queried directly, without storing and searching the original space. This would increase the efficiency substantially, while lowering the amount of memory required by the index, which was the main goal of this research.

3.4.4 Code-space

<https://github.com/pilotCapp/annbench-annann-ext>

3.5 Similar work

After the implementation, some similar concepts have been revealed. These should probably have been explored before drafting the proposal, however we now get to compare our approaches, and analyze some key differences considering benefits and drawbacks.

3.5.1 Matryoshka Representation

The Matryoshka Representation Learning *MRL* method, proposed in 2022 [18], suggests learning a dynamic representation where different partitions of the embedding can be used at different stages of a task. For example, in a KNN search, you could start with a lower-dimensional prefix of the query embedding, which reduces the initial computation cost. As the search narrows, you progressively increase the dimensionality for finer details, allowing for more precise results. This approach reduces computation during the early broad search, enabling efficient refinement while reducing the scope of the search.

One important difference to the proposed implementation is that while *MRL* reduces the computational overhead during the search by using lower-dimensional representations in the initial stages, it does not reduce the overall memory consumption of the embeddings. The full embeddings are still stored in memory. However, the impact on efficiency for large datasets remains unclear, as it

might require accessing different parts of the embedding from disk for large datasets, which could affect real-time search performance.

3.5.2 Adaptive ANN Search (AdANNs)

Utilizing Matryoshka Representation Learning (MRL), the authors of the original paper proposed AdANN [19], an algorithm designed for KNN search that incorporates learned product quantization (1.2.5) and integrates with DiskANN [3] for improved efficiency. This approach achieves similar recall to baseline methods while offering significantly better computational efficiency.

Despite its advantages, AdANN shares certain drawbacks with the proposed solution. For instance, it requires extensive training to generate the index, which can be computationally expensive. However, this training is a one-time cost and does not impact query-time performance. Additionally, AdANN necessitates embedding query vectors into the MRL format during the search process, which slightly reduces efficiency. Nevertheless, embedding a single query vector adds only a marginal cost to the overall runtime 1.

To implement a search across multiple dimensions in the proposed solution, either multiple models would need to be queried, or embeddings of different dimensions would have to be extracted from a single model. Both approaches introduce additional computational overhead. *AdANN* is more able to reduce the dimensionality for each step, providing more fine grained control during search, and to efficiently create learned dimensions more suited for the search process. Still it does not seem likely to handle extremely large datasets, or dynamic instances very well. Further research towards these goals seems highly promising.

In summary, *AdANN* demonstrates a promising balance of efficiency and recall by leveraging MRL for hierarchical embeddings and adaptive search. However, it might come with specific limitations, particularly when applied to dynamic datasets where frequent updates may require re-training or re-indexing. This is supported by the fact that *ANNANN* performs worse in the dynamic over the static set compared to the baseline.

4 Summary

We have seen how some of the most efficient KNN algorithms work and where they are used 2.1. How they utilize different optimization techniques for efficiency, and how some algorithms work well for larger datasets while others don't. We have seen the pitfalls of some algorithms in this regard, particularly around index size constraints, and how some algorithms handle this 2.2. We have also briefly explained other interesting concepts such as *Metric Learning*, *DEC*, and *Learned Indexing* 1.4. Different use-cases for KNN algorithms have also been discussed 1.5, and how these cases could potentially benefit from algorithms with capabilities for larger datasets, before formulated a research question that captures this 1.6.

The proposal 3 was an attempt at incorporating discussed concepts for a KNN search 1.4, that could potentially help scale future algorithms to larger datasets. Even though it did not achieve the theorized results, it also proved that such an algorithm is possible. Furthermore the reduction in performance was not big enough to give a certain conclusion about the possible results for similar implementations in the future. Research solving the remaining problems, in particular being able to efficiently search individual instances within the embedded space, could potentially improve performance while greatly reducing the memory footprint. Additionally, related work in this domain has highlighted promising avenues for future research, emphasizing the potential for further advancements in scalable and memory-efficient search algorithms.

References

- [1] J. L. Bentley, ‘Multidimensional binary search trees used for associative searching’, Stanford University, Technical Report TR-75-06, 1975.
- [2] E. Bernhardsson. ‘Ann benchmarks’. (2024), [Online]. Available: <https://github.com/erikbern/ann-benchmarks/> (visited on 14th Nov. 2024).
- [3] M. Research. ‘Diskann: Fast accurate billion-point nearest neighbor search on a single node’. (2019), [Online]. Available: <https://www.microsoft.com/en-us/research/project/diskann/> (visited on 1st Nov. 2024).
- [4] Pinecone. ‘Product quantization’. (2024), [Online]. Available: <https://www.pinecone.io/learn/series/faiss/product-quantization/> (visited on 25th Nov. 2024).
- [5] OpenAI. ‘Gpt-4’. (2024), [Online]. Available: <https://openai.com/index/gpt-4/> (visited on 27th Nov. 2024).
- [6] StoreDBits. ‘Sequential vs random read/write performance in storage’. (2024), [Online]. Available: <https://storedbits.com/sequential-vs-random-data/> (visited on 6th Nov. 2024).
- [7] *Neurips big ann benchmark*, 2021. [Online]. Available: <https://big-ann-benchmarks.com/neurips21.html> (visited on 1st Nov. 2024).
- [8] M. Research. ‘Freshdiskann: A fast and accurate graph-based ann index for streaming similarity search’. (2021), [Online]. Available: <https://arxiv.org/abs/2105.09613> (visited on 1st Nov. 2024).
- [9] *Neurips big ann benchmark*, 2023. [Online]. Available: <https://big-ann-benchmarks.com/neurips23.html> (visited on 1st Nov. 2024).
- [10] Jyotsana, *Deep metric learning - fundamentals*, <https://example.com/deep-metric-learning>, Accessed: 2024-11-01, 2023.
- [11] J. Xie, R. Girshick and A. Farhadi, ‘Unsupervised deep embedding for clustering analysis’, *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 478–486, 2016. DOI: 10.5555/3045118.3045142.
- [12] T. Kraska, A. Beutel, E. H. Chi, J. Dean and N. Polyzotis, *The case for learned index structures*, <https://example.com/learned-index-structures>, Accessed: 2024-11-01, 2023.
- [13] Pinecone. ‘Rag study’. (2024), [Online]. Available: <https://www.pinecone.io/blog/rag-study/> (visited on 27th Nov. 2024).
- [14] P. Sobot. ‘Introducing voyager: Spotify’s new nearest neighbor search library’. (2023), [Online]. Available: <https://engineering.atspotify.com/2023/10/introducing-voyager-spotifys-new-nearest-neighbor-search-library/> (visited on 27th Nov. 2024).
- [15] G. Research. ‘Scann algorithms documentation’. (2024), [Online]. Available: <https://github.com/google-research/google-research/blob/master/scann/docs/algorithms.md> (visited on 14th Nov. 2024).
- [16] Y. Japan. ‘Ngt wiki’. (2024), [Online]. Available: <https://github.com/yahoojapan/NGT/wiki> (visited on 14th Nov. 2024).
- [17] L. McInnes. ‘How pynndescent works’. (2024), [Online]. Available: https://pynndescent.readthedocs.io/en/latest/how_pynndescent_works.html (visited on 14th Nov. 2024).
- [18] Kusunapati, Bhatt, W. Rege *et al.*, ‘Matryoshka representation learning’, University of Washington, Google Research, Harvard University, Tech. Rep., 2022.
- [19] Rege, R. S. Kusunapati, C. Fan, J. Kakade and Farhadi, ‘Adanns: A framework for adaptive semantic search’, University of Washington, Google Research, Harvard University, Tech. Rep., 2023.

Appendix

A AI Declaration

Deklarasjon om KI-hjelpemidler

Har det i utarbeidningen av denne rapporten blitt anvendt KI-baserte hjelpemidler?

☐ Nei

☒ Ja

Hvis ja: spesifiser type av verktøy og bruksområde under.

Tekst

☒ **Stavekontroll.** Er deler av teksten kontrollert av:
Grammarly, Ginger, Grammarbot, LanguageTool, ProWritingAid, Sapling, Trinkai.ai eller lignende verktøy?

☒ **Tekstgenerering.** Er deler av teksten generert av:
ChatGPT, GrammarlyGO, CopyAI, WordAi, WriteSonic, Jasper, Simplified, Rytr eller lignende verktøy?

☒ **Skriveassistanse.** Er en eller flere av ideene eller fremgangsmåtene i oppgaven foreslått av:
ChatGPT, Google Bard, Bing chat, YouChat eller lignende verktøy?

Hvis ja til anvendelse av et tekstverktøy - spesifiser bruken her:

chatgpt har blitt brukt for informasjonsfinning, teordiskutering, implementering av kode, samt rettskriving og omformuleringer. All tekst er skrevet manuelt, hvor enkelte deler er omformulert med tekstgenerering for presisering.

Kode og algoritmer

☒ **Programmeringsassistanse.** Er deler av koden/algoritmene som i) fremtrer direkte i rapporten eller ii) har blitt anvendt for produksjon av resultater slik som figurer, tabeller eller tallverdier blitt generert av: *GitHub Copilot, CodeGPT, Google Codey/Studio Bot, Replit Ghostwriter, Amazon CodeWhisperer, GPT Engineer, ChatGPT, Google Bard* eller lignende verktøy?

Hvis ja til anvendelse av et programmeringsverktøy - spesifiser bruken her:

Copilot har blitt brukt for implementering av diskutert algoritme, samt med innvendinger fra chatgpt


Bilder og figurer

☐ **Bildegenerering.** Er ett eller flere av bildene/figurene i rapporten blitt generert av:
Midjourney, Jasper, WriteSonic, Stability AI, Dall-E eller lignende verktøy?

Hvis ja til anvendelse av et bildeverktøy - spesifiser bruken her:

☐ **Andre KI verktøy.** har andre typer av verktøy blitt anvendt? Hvis ja spesifiser bruken her:

☒ Jeg er kjent med NTNUs regelverk: *Det er ikke tillatt å generere besvarelse ved hjelp av kunstig intelligens og levere den helt eller delvis som egen besvarelse.* Jeg har derfor redegjort for all anvendelse av kunstig intelligens enten i) direkte i rapporten eller ii) i dette skjemaet.

 6/12/24 Trondheim
Underskrift/ Dato/Sted