B.Sc. in Computer Science and Engineering Thesis

A Comparative Study of Approximate Nearest Neighbor Search Algorithms in High Dimensions

Submitted by

Md Sabbir Rahman
201705076

Kazi Md Irshad Farooqui
201705094

Supervised by

Dr. Rifat Shahriyar

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

May 2023

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis, titled, "A Comparative Study of Approximate Nearest Neighbor Search Algorithms in High Dimensions", is the outcome of the investigation and research carried out by us under the supervision of Dr. Rifat Shahriyar.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Md Sabbir Rahman
201705076

Kazi Md Irshad Farooqui
201705094

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our supervisor, Professor Dr. Rifat Shahryar, for his invaluable guidance and support throughout the thesis. We are incredibly fortunate to have had the opportunity to work with such an inspiring mentor and are forever grateful to him for his patience in tolerating all our shortcomings and continuously encouraging us.

We would like to thank honorable members of the examination board, for their valuable comments.

We would also like to extend our thanks to Tahmid Hasan, who provided us with valuable insights and guidance throughout our thesis. Without his feedback and suggestions, this work would not have been possible. We are grateful for his mentorship and support throughout the process.

Finally, we would like to acknowledge the contributions of all those who supported us in this endeavor, including our families and friends, who provided us with the emotional support and encouragement we needed to see this project through to completion.

Dhaka
May 2023

Md Sabbir Rahman

Kazi Md Irshad Farooqui

Contents

List of Figures

ABSTRACT

Nearest Neighbor Search (NNS) is a critical component of many applications, such as information retrieval, pattern recognition, and recommender systems. This problem entails finding the closest vectors to a query vector within an input vector set. However, solving NNS exactly becomes exceedingly difficult when the dimensionality of the vectors is in the hundreds or thousands. As such, literature has focused on Approximate Nearest Neighbor Search (ANNS). Numerous ANNS algorithms have been proposed, ranging from converting data to low dimensionality to using heuristics to prune to a subset of the input subset. However, no single algorithm has universal superiority, with each having tradeoffs to consider. In this study, we investigate different ANNS algorithms, emphasizing their core ideas and comparing them.

We begin by introducing the NNS problem and various distance metrics used to determine "nearness". We provide established results on using the Euclidean distance metric in place of other well-known metrics. Next, we categorize existing NNS methods into four classes and study each in detail. Finally, we employ FAISS, a dedicated library for NNS, to compare representative algorithms of different types. We evaluate several metrics on real-life datasets of varying purposes and discuss the tradeoffs of different algorithms. By studying and comparing different ANNS algorithms, this work provides valuable insights and recommendations for selecting the appropriate NNS algorithm for a given problem.

Chapter 1

Introduction

Nearest neighbor search (NNS) is a fundamental component of numerous information systems that deal with semantic representations of objects. Its application spans across domains like information retrieval [1], pattern recognition [2], machine learning [3], and recommender systems [4]. In all of these applications, NNS involves embedding objects into high-dimensional vector spaces, where the proximity between vectors translates to similarity between objects. To find similar objects, applications require finding the k-nearest vectors to a particular query vector according to some specified metric. The metric used in NNS varies from problem to problem. The most commonly used metric is Euclidean distance or L^2-norm, but some specific uses require cosine similarity between vectors, while others require the inner product.

Despite the apparent simplicity of the problem requirement, NNS cannot be easily tackled and poses significant challenges. In practical applications, the data scales explosively [5], while the dimensions of the embedded vectors can be as large as nearly a thousand [6]. Due to such high dimensions, not only are traditional approaches unsatisfactory [7], but exact nearest neighbor finding in itself becomes a computationally challenging problem. Instead, Approximate Nearest Neighbor Search (ANNS) is used [8]. In ANNS, we do not prioritize the most nearest neighbors; rather, we are happy with results that are not exact but are close to exacts. This trade-off of accuracy allows us to tackle the problem efficiently, allowing for approximate but close results.

The approaches to ANNS in recent times can be categorized into four main groups: tree-based [9], hashing-based [10], quantization-based [11], and graph-based [12]. However, not all approaches fall precisely into these groups, and sometimes it is possible to combine approaches to get better results.

In our study, we present an overview of the nearest neighbor problem, along with a comprehensive background study of the main approaches. Several libraries have been built to tackle the problem of ANNS, such as FAISS [13], ANNOY [14], Hnswlib [15], and others. We particularly use FAISS as the primary library, which supports all of the approaches discussed thereof,

to compare them. We use several representative methods on real-life datasets, and compare their tradeoffs from various perespectives such as approximation, intialization overhead, query efficiency, and memory overhead. Using these discussions, we highlight the limitations of the experimented algorithms and provide recommendations for selecting appropriate methods based on on task-specific requirements.

Chapter 2

Motivation for Nearest Neighbour Search

Nearest Neighbours are essential components of fundamental algorithms such as clustering and machine learning, including the widely-used KNN algorithm [16]. However, in such cases, exact nearest neighbours are required. In information retrieval and related systems, approximate nearest neighbours are often sufficient. ANNS is employed in various use cases with unique applications, some of which will be explained. But in all cases, ANNS involves working with embeddings of information, where embedded vectors are associated with a particular distance metric that approximates the similarity between the information they represent. Finding the nearest neighbours then translates to identifying the most similar candidate information. As the meaning of the distance metric becomes less sensitve with high dimensions [7], ANNS is used to find the most similar candidates with some leniency, allowing for efficient search. We list several applications in the sections below:

2.1 Recommender Systems

Recommender systems are widely used in industry to put forth content to users based on their past behavior. The key idea is to learn enough patterns or trends from a user's history to predict what kind of content will make them engage more. A popular approach [17] is to learn embeddings for user activity and content together, where users are assigned a vector u_i and content is assigned vectors v_j. The compatibility of user i and content j can be represented as

$$r_{i,j} = \mu + u_i'^T v_j' \tag{2.1}$$

where u_i' and v_j' are some known simple transformations on the embeddings and μ is a learned constant. To find the most compatible content, we aim to find content j such that

$$j = \arg\max_k r_{i,k} = \arg\max_k u_i'^T v'_k \tag{2.2}$$

In practice, recommender systems aim to find not just one, but k contents with high compatibility. This is an instance of nearest neighbor search (NNS) with the distance metric being the inner product. However some recommender systems work with huge content databases [18] and the complexity of content results in high dimensions. So ANNS is used in such cases.

2.2 Information Retrieval

Information retrieval (IR) encompasses the retrieval of various types of content such as images, texts, and videos, among others. Popular approaches [19] to these tasks involve transforming the information into a latent space where the cosine similarity between vectors represents similarity among the information.

For instance, an information search engine might index existing contents with a unique vector u_i for each content. Then, given a query content, the engine maps it correspondingly to v_j and finds the vectors closest to it in its database:

$$i = \arg\max_k \cos(\angle(u_k, v_j)) \tag{2.3}$$

For instance, the Microsoft Azure Cognitive Services uses this approach for their image retrieval APIs[1].

Recently, joint embedding of multiple domains has been employed so that, for example, text can be used to search for images or vice versa. In this case, given a query vector, the engine might search only one of the datasets.

2.3 Language Models

Language models are used to assign probabilities to sentence contexts, to predict the next most probable sentence. In a recent work [20], researchers transformed pre-trained language models into a nearest neighbour problem, by finding the nearest neighbors to a vector instead of maximizing a probability expression. The metric used in this new latent space depends on the model,

[1] see Do image retrieval using vectorization (version 4.0 preview)

but L^2-distance has been mostly successful. Due to the size of these language models, ANNS is the only efficient method to solve this problem.

An older but similar approach [21] used sample memorization and nearest neighbour to achieve better results with fewer parameters. In this method, observed samples are used to build a latent space, where nearest neighbour search produces an existing sample. Although it is a limited approach, it can be useful when data is scarce.

Chapter 3

Preliminaries

3.1 Problem Definition

The problem of nearest neighbor search (NNS) can be formally defined as follows:

Definition 1. *Given a finite dataset X in Euclidean space $\mathbb{R}^d$, a query vector q also in $\mathbb{R}^d$, and a positive integer k, the KNNS problem is to find the set of k nearest neighbors of q from X according to a distance metric δ. Formally, we have to find the set S such that*

$$S = \arg\min_{\substack{S \subset X \\ |S|=k}} \sum_{x \in S} \delta(x, q) \tag{3.1}$$

However, as the size of X and d grows, exact KNNS becomes computationally expensive. Therefore, approximations are used in practice. There are formal definitions of approximate nearest neighbor search (AKNNS) that use an error bound ϵ [8]. But most practical works avoid it as the mathematical analysis can be complex. In practice, the following semi-formal definition is often used:

Definition 2. *Given a finite dataset X in Euclidean space $\mathbb{R}^d$, a query vector q also in $\mathbb{R}^d$, and a positive integer k, if KNNS outputs the set S, then AKNNS outputs an approximation $\tilde{S}$ such that the recall $\frac{S \cap \tilde{S}}{k}$ is reasonable.*

The level of recall that is considered "reasonable" depends on the specific use case and can be a trade-off between recall, speed, and memory efficiency [22]. The challenge in AKNNS is to find an approximation method that strikes a balance between these factors.

3.2 Distance Metrics

The distance metric δ used in nearest neighbor search is not always the same. Depending on how the embeddings are produced, the metric used changes. Some of these distance metrics are Euclidean distance, inner product, cosine similarity, manhattan distance, hamming distance etc. Among these the first three are most used and they possess a form of equivalence among them. We describe these below:

3.2.1 Euclidean Distance

Definition 3. *In Euclidean space $\mathbb{R}^d$ the Euclidean distance between two vectors $u, v \in \mathbb{R}^d$ is defined as:*

$$\delta(u, v) = |u - v| = \sqrt{\sum_{i=1}^{d} (u_i - v_i)^2} \tag{3.2}$$

where $u = (u_1\ u_2\ \cdots\ u_d)^T$ and $v = (v_1\ v_2\ \cdots\ v_d)^T$

Euclidean distance is widely used in various applications such as machine learning, computer vision, and pattern recognition, because it has desirable mathematical properties such as satisfying the triangle inequality. In literature, Euclidean distance is also referred to as L^2 metric, which is a special case of a family of distance metrics called L^p metrics.

In nearest neighbor search, we often use the squared Euclidean distance, which is just the term inside the square root in the Euclidean distance formula. This allows us to avoid computing costly square roots, and still preserve the order of distances.

3.2.2 Inner Product

Definition 4. *In Euclidean space $\mathbb{R}^d$ the inner product distance between two vectors $u, v \in \mathbb{R}^d$ is defined as:*

$$\delta(u, v) = u \cdot v = \sum_{i=1}^{d} u_i v_i \tag{3.3}$$

where $u = (u_1\ u_2\ \cdots\ u_d)^T$ and $v = (v_1\ v_2\ \cdots\ v_d)^T$

While the inner product is not technically a distance metric, it is commonly used as a similarity metric. Unlike other metrics, a higher inner product value represents higher similarity between the vectors. Despite its limitations, the inner product is widely used in embedding learning techniques due to its linearity properties.

One advantage of the inner product over other distance metrics is that it is computationally efficient to compute, as it simply involves a dot product. This makes it a popular choice in large-scale applications.

3.2.3 Cosine Similarity

Definition 5. *In Euclidean space $\mathbb{R}^d$ the cosine distance between two vectors $u, v \in \mathbb{R}^d$ is defined as:*

$$\delta(u, v) = \cos(\angle(u, v)) = \frac{u \cdot v}{|u||v|} = \frac{\sum_{i=1}^{d} u_i v_i}{\sqrt{\sum_{i=1}^{d} u_i^2}\sqrt{\sum_{i=1}^{d} v_i^2}} \tag{3.4}$$

where $u = (u_1\ u_2\ \cdots\ u_d)^T$ and $v = (v_1\ v_2\ \cdots\ v_d)^T$

Like inner product, Cosine similarity also represents higher similarity with higher value. It can be seen as an extension of inner product where the vectors are scaled to be of unit magnitude. Cosine similarity is a popular choice for measuring similarity between two vectors because it is scale-invariant. This property is particularly useful when dealing with high-dimensional data where the magnitude of the vectors may not be meaningful.

3.2.4 Equivalence of the three metrics

From [23], it can be shown that above 3 metrics are reducible to each other albeit with some extra dimensions. In particular, the following are true (the exact details is in the reference paper)

Theorem 1. *Given $q \in \mathbb{R}^d$ and $X \subset \mathbb{R}^d$, there is a transformations $f_1 : \mathbb{R}^d \to \mathbb{R}^{d+1}$ and $g_1 : \mathbb{R}^d \to \mathbb{R}^{d+1}$ such that $|f_1(q) - g_1(x_i)|^2 \leq |f_1(q) - g_1(x_j)|^2$ implies $q \cdot x_i \geq q \cdot x_j$.*

Theorem 2. *Given $q \in \mathbb{R}^d$ and $X \subset \mathbb{R}^d$, there is a transformations $f_2 : \mathbb{R}^d \to \mathbb{R}^d$ and $g_2 : \mathbb{R}^d \to \mathbb{R}^d$ such that $f_2(q) \cdot g_2(x_i) \geq f_2(q) \cdot g_2(x_j)$ implies $\frac{q \cdot x_i}{|q||x_i|} \geq \frac{q \cdot x_j}{|q||x_j|}$.*

Theorem 3. *Given $q \in \mathbb{R}^d$ and $X \subset \mathbb{R}^d$, there is a transformations $f_3 : \mathbb{R}^d \to \mathbb{R}^{d+1}$ and $g_3 : \mathbb{R}^d \to \mathbb{R}^{d+1}$ such that $\frac{f_3(q) \cdot g_3(x_i)}{|f_3(q)||g_3(x_i)|} \geq \frac{f_3(q) \cdot g_3(x_j)}{|f_3(q)||g_3(x_j)|}$ implies $|q - x_i|^2 \leq |q - x_j|^2$.*

So, knowing algorithm for only one metric translates to ability to solve for other two metrics. In particular, most works focus on nearest neighbour with Euclidean distance and reduce other two to it.

Chapter 4

Approaches to Approximate Nearest Neighbor Search

The brute-force method of solving nearest neighbor search is to simply iterate over the data and keep track of the k best candidates seen so far. However, this approach is computationally expensive and is not practical for large datasets. As stated before, methods for nearest neighbor search can be broadly categorized into four groups: tree-based, hashing-based, quantization-based, and graph-based methods.

It should be noted that tree-based algorithms are generally not suitable for high-dimensional data. Therefore, we will briefly discuss tree-based methods while placing more emphasis on the other three approaches. The following sections provide a detailed discussion of each of these methods.

4.1 Tree-based Methods

Tree-based or partition-based methods build a rooted tree over data such that each data point is represented by a leaf of the tree. Given a query point, these methods explore the tree to find leaves closest to the query point. While most methods follow the same idea, the variation comes from how to build the tree and how to explore it. However, due to the curse of dimensionality, in most cases, these methods become almost brute force.

The most classic tree method for KNN is the KD tree. However, KD tree is part of a larger family known as binary space partition trees (BSP trees). Every node of a KD tree is divided into two children by choosing an axis and dividing along the median of the data in that axis. The choice of axis is suggested to be random. In fact, there are variations of KD tree called random projection trees where, along with axes, the partition among points is also random.

Instead of trying a hyperspace partition, some methods attempt curved partitions. One such example is the ball tree [24]. In ball trees, each node represents points within a hypersphere. The intuition is that with hyperspheres, maximum distances between groups of points are easier to tackle. There are also other tree-based methods such as cover trees [25] and VP trees [26].

Another approach is to use several trees together in parallel or sequentially. One such example is the rpforest [27], which combines multiple KNN-sensitive trees. A big advantage of forests is that they can be parallelized.

Most tree-based methods fall short in their exploration to answer a query. For any query point, tree exploration needs to be able to prune branches of the tree so that only relevant leaves are found. However, this becomes nearly impossible in high dimensions. Despite this, recent research [28] has been done on tree methods for high-dimensional data.

4.2 Hashing-based Methods

Hashing-based methods aim to reduce the dimensionality of data by hashing all the points in the dataset from the original data space to a lower-dimensional space. In this reduced space, nearest neighbor search is easier due to lower dimensionality, and pruning of points becomes possible, resulting in lower memory requirements for the data structure. However, inexact distances are a trade-off for these benefits.

The main component of hashing-based methods is a simple hashing function h. Typically, several such functions $h_1, h_2, \cdots, h_m$ are concatenated to form the hash code $y = h(x) = (h_1(x)\ h_2(x)\ \cdots\ h_m(x))^T$. The parameter m is a parameter of the "hash index", and lower values of m make the index more efficient but with high loss of information. On the other hand, higher values of m result in less efficiency but better information retention.

There are two strategies used with hashing approaches. The first strategy involves keeping hash tables of all hashed vectors. In this approach, the query vector is hashed to transform it into the reduced hash space, and all hash collisions with existing points are found. These points are then checked, resulting in a reduced search space. However, this method does not perform well because using only one hash function may prune away nearest points, so multiple hash functions are used to improve recall. But due to the curse of dimensionality, nearer points may still get pruned.

The second and more prevalent strategy involves using all the data points as candidates. Instead of using their original dimensions to compute distances, we use their compact hash codes to compute distances. This strategy trades efficiency for accuracy but generally gives better results. Sometimes the result set is reranked to improve the output.

Hashing-based methods use particular hashes that fall into two categories [29]: locality-sensitive

hashing (LSH) and learning to hash. Both types of methods are discussed below.

4.2.1 Locality-Sensitive Hash

Locality-Sensitive Hash (LSH) are hash functions that try to assure that nearer vectors are hashed into same bucket i.e. collide more. This is different from traditional hash functions which try to make collision rare.

To make locality sensitive hash, we require LSH functions $h_1, h_2, \cdots, h_m$. These are then applied on a vector and the results are concatenated to form a compound LSH function which results in both randomization and locality preservation. Formally LSH function are defined as follows

Definition 6. *Given a metric space and its distance metric δ, a threshold $t > 0$, an LSH function h should satisfy for any two points u, v in the space following two things:*

1. *If $\delta(u, v) < t$ then $h(u) = h(v)$ with probability at least p_1.*
2. *If $\delta(u, v) \geq ct$ then $h(u) = h(v)$ with probability at most p_2.*

$c > 1$ being an approximation factor and $p_1 > p_2$.

There has been extensive study in what LSH functions to use for different distance metrics including L^p distances [30] and cosine similarity. In particular for L^2 distance LSH functions is

$$h(u) = \left\lfloor \frac{w \cdot u + b}{r} \right\rfloor \tag{4.1}$$

Here w is a vector with each coordinate an independent sample from p stable distribution, r is a scaling term chosen arbitrarily, b is a bias chosen randomly from $[0, r]$.

For cosine similarity, LSH function is

$$h(u) = \operatorname{sgn}(w \cdot u) \tag{4.2}$$

Here w is a sample from the standard Gaussian distribution.

Using LSH as a building block, many variations have been invented. Multi-probe LSH [31] tries to generate various perturbations of one hashcode to find several candidate hash buckets. This results in pruning of search space. Query adaptive LSH [32] works with offline querying where knowing the query points, hashes are generated tuned to the known query points. SortingKeys-LSH [33] tries to define a special order on the hashcodes so that nearer vectors are ordered sequentially, so that retrieving nearer vectors to hashed query vector is easier.

4.2.2 Learning to Hash

While LSH uses hash functions that are independent of data, learning to hash is a technique where the hash fucntions are generated based on the input set of points. The goal is to "learn" a hash funciton such that the hashed space will preserve relative distance information of vectors and computing distance over this hashed space will be efficient. There are generally three components [34] to learning to hash. They are:

Hash function First, we need to come up with a family of hash functions with some number of parameters. The final hash function is generated by specifying these parameters. For example, a simple family of hash functions is the following:

$$h(x) = \text{sgn}(w \cdot x + b) \tag{4.3}$$

Here, the vector w and the bias b are to be learned. More complicated hash functions can also be chosen.

Similarity measures While the similarity measure in the original space is relevant but out of our control, the similarity measure to use in the hashed space is set by us. Traditionally, Hamming distances or a variation of Euclidean distance is used. The metric chosen should be computationally inexpensive. Spectral hashing [35] is an example which uses Hamming distances.

Optimization With the hash function and similarity measure chosen, we have to come up with an optimization criterion. A simple such criterion could be to minimize the differences between distances in the original space and in the hashed space. More complicated approaches could try to also ensure that the original vectors are distributed approximately equally to hash codes.

There have been extensive studies in learning to hash. Methods include both supervised and unsupervised learning techniques. Recent works have tried to use deep learning in this scope. For example, Deep Hashing proposed by [36], where input vectors are passed through a deep network which result in hash codes. Some approaches, like NINH [37], try to learn the embedding from the original information domain together with the hash codes.

The biggest drawback of learning to hash techniques is the large-scale data volume. Thus, weakly supervised techniques have also been researched. While learning to hash has mostly been used in image retrieval, it shows promise in other domains too.

4.3 Quantization-based Methods

Quantization [38] is a classical technique for enhancing the efficiency of data storage and processing while losing very little information. In the context of nearest neighbor search, quantization involves mapping data vectors to a reduced subset such that distance computations with converted vectors can be performed using lookup or other methods. The fundamental idea of quantization-based methods is to choose a codebook of a fixed size, consisting of "centroids" $c_1, \cdots, c_b$. Then, each vector in the input data is assigned to one of these centroids using a simple rule:

$$Q(x) = \arg\min_{c_i} |x - c_i|^2 \tag{4.4}$$

This approach offers memory efficiency as we can store indices to $Q(x_i)$ instead of x_i. However, it introduces quantization error. Unfortunately, the method becomes expensive when the data volume scales to millions or billions of dimensions. So, an improved approach, product quantization, is used in practice

In product quantization [11], the input vector space's dimension d is split into m disjoint contiguous subvectors. Then, for each subspace of dimension d^*, quantization is performed individually. The mapping function can be expressed as:

$$Q(x) = Q((\underbrace{x_1 \ \cdots \ x_{d^*}}_{u_1(x)} \ \cdots \ \underbrace{x_{d-d^*+1} \ \cdots \ x_d}_{u_m(x)})^T) = (Q_1(u_1(x)) \ \cdots \ Q_m(u_m(x)))^T \tag{4.5}$$

Where $Q_1, \cdots, Q_m$ are quantization functions for each subspace, as defined above. If we use b centroids per subspace, then we need to use a lookup table of mb vectors of d^* dimensions, and the number of distinct representations for full vectors becomes b^m.

There are several components involved in using quantization-based methods for nearest neighbor search, which are explained in detail below:

4.3.1 Quantization in Distance Computation and Error

Given a query vector q, we compute the distance between q and a database vector x_i by using their quantized representations $Q(q)$ and $Q(x_i)$. This introduces quantization error, which is the difference between the true distance and the approximated distance. But it also gives us ability to use lookup tables for efficient computation. There are two main ways to compute the distances:

Symmetric Distance Computation (SDC): Both q and x_i are quantized via Q, and their distance is computed as:

$$\tilde{\delta}(q, x_i) = \delta(Q(q), Q(x_i)) = \sqrt{\sum_j |Q_j(u_j(q)) - Q_j(u_j(x_i))|^2} \tag{4.6}$$

Here, $\delta(c_s, c_t)$ denotes the distance between centroids c_s and c_t, and $u_j(q)$ and $u_j(x_i)$ are the jth subvectors of q and x_i, respectively. Since the number of centroids per subspace b is generally not large, we can precompute all $\delta(c_s, c_t)$ and keep them in memory after applying quantization to the input data.

Asymmetric Distance Computation (ADC): Only x_i is quantized via Q, and q is used as it is, resulting in the distance computation:

$$\tilde{\delta}(q, x_i) = \delta(q, Q(x_i)) = \sqrt{\sum_j |u_j(q) - Q_j(u_j(x_i))|^2} \tag{4.7}$$

In ADC, the distance computation can be optimized by computing each of $\delta(u_j(q), c_s)$ for a subspace j before searching with one q.

Generally, ADC is the superior approach since SDC introduces more error due to two approximations $Q(q)$ and $Q(x_i)$, whereas ADC only has the error introduced by $Q(x_i)$. A mathematical analysis of the error due to product quantization is shown in [11]. The only advantage of SDC is that it requires mb^2 values to be stored for all queries, whereas ADC requires mb values per query.

4.3.2 Searching method

Product Quantization reduces memory usage and shows significant improvement over the naive brute search. However, performing a complete brute search over the full dataset is still expensive. To address this issue, non-exhaustive search techniques are applied to search over only a fraction of the input set. One common technique in information retrieval is the inverted file system [39], where searching is done by first dividing the data into disjoint lists. During a query, a number of lists are chosen to search through.

A data structure combining inverted file system and product quantization, is called IVFADC [11]. First, a coarse quantizer Q_C is used on the full input data to map each vector to a fixed number of quantized centroids, similar to hash buckets. Then, for each of these clusters, we compute the residual $r(x) = x - Q_C(x)$, and apply a fine product quantization Q_P on $r(x)$. The resulting

data structure consists of the representation

$$\tilde{x} = Q_C(x) + Q_p(r(x)) \tag{4.8}$$

During search, we probe only some of the coarse centroids by using $\delta(q, Q_C(x_i))$ to select candidate lists. Then, for each list, we loop over all vectors in the candidate lists, comparing $\delta(q - Q_C(x_i), Q_P(r(x)))$. The number of coarse centroids to probe, the number of coarse and fine centroids, and the number of subspaces are parameters that control IVFADC.

Several variation of IVFADC have been researched in literature. One such variation is the inverted multi-index [40], where the coarse quantization is another product quantization. In the coarse quantizer of the inverted multi-index, first, nearest subspace centroids are found in each subspace, and then, using a priority queue, the nearest centroids are found in the original space. Another search algorithm utilizing product quantization is PQTable [41], which uses a hash table instead of an inverted file system to non-exhaustively search for nearest neighbors.

4.3.3 Improving Quantization

One of the main challenges of quantization is choosing the centroids so that the quantization error is minimized. [39] proposed solution is using the k-means algorithm to choose the centroids. While this approach can yield satisfactory results, there have been other works aimed at improving the quantization process.

An approach that does not change the quantization method is Optimized Product Quantization (OPQ) [42], which involves learning a rotation that is applied to each input and query vector before using product quantization. Another novel approach is Distance Encoded Product Quantization [43], where coarse and fine quantizations are utilized in combination, such that clusters with more variance are assigned more bits to the fine quantization.

Several works have also explored the use of deep neural networks to improve quantization. For example, the Deep Quantization Network (DQN) [44] learns both a vector embedding and an optimal product quantizer for the original data. Similarly, the Deep Product Quantization (DPQ) [45] approach uses a joint loss function and a different network structure to achieve improved quantization. Deep Progressive Quantization [46] is another approach that uses several quantization blocks to achieve quantization of different code lengths. However, these approaches can suffer from the same challenges as deep learning approaches in hashing-based methods, such as large volumes of data.

4.4 Graph-based Methods

Graph-based methods are currently considered the most promising approaches for approximate nearest neighbor searching. Although these methods generally follow a similar graph creation and search process, there is significant variation in the structures and properties of various methods.

A common graph-based NNS method involves creating a graph on the input data, where each node corresponds to an input vector and the edges between nodes are determined by the distances between the corresponding points.

In a comprehensive review of various graph-based ANNS methods by [47], four types of graphs were identified as the basis for NNS methods: Delaunay Graph (DG) [48], Relative Neighborhood Graph (RNG) [49], K-Nearest Neighbor Graph (KNNG) [50], and Minimum Spanning Tree (MST). Many methods are based on one or more of these four graphs, but all methods typically consist of two parts: graph construction and graph search.

In the following sections, we will provide a brief overview of various graph-based algorithms and their two parts.

4.4.1 Different Types of Graph-Based Methods

DG and RNG are two similar graph bases that have been used in several methods with good results. Both of these graphs ensure that if there is an edge between nodes corresponding to vectors u and v, then a restriction is applied on other nodes or edges with other nodes. However, these graphs suffer from high memory usage, making it difficult to use them in practice. Researchers have attempted to reduce the memory usage by advanced techniques.

Navigable Small World (NSW) [51] Graph is a method that tries to approximate Small World graphs, a graph similar to DG. It builds a graph by incrementally inserting vectors while ensuring connectivity. An improved version of NSW is Hierarchical NSW (HNSW) [52], which imposes a hierarchy in the created graph, similar to skip-lists, where upper layers have smaller size than lower layers. HNSW is one of the most promising methods among various recent methods, as it has presented superior results that are only achieved by few others.

Fast Approximate Nearest Neighbor Graphs (FANNG) [53] tries to approximate RNG by pruning edges between nodes. However, its construction can be very slow depending on the method used. Neighborhood Graph and Tree (NGT) [54] tries to create an approximate graph to RNG by offering methods similar to NSW.

KNNG is a graph where each node has directed edges to its k nearest nodes. Since the number of edges is limited, it has a low memory requirement. However, there is no guarantee

of connectedness in KNNG, which can badly prune out nearest neighbors for a query point. Space Partition Tree and Graph (SPTAG) [55] tries to create an exact KNNG by using divide and conquer. Connectivity can then be ensured by propagating the neighborhood of nodes. Other methods such as KGraph [56], EFANNA [57], IEH [58] use different approaches to build KNNG and modify it.

Some methods try to incorporate both KNNG and RNG. Examples of such methods are Diversified Proximity Graph (DPG) [59], Navigating Spreading-out Graph (NSG) [60], Navigating Satellite System Graph (NSSG) [61], and Vamana [62]. These methods show promise and, in some cases, show considerable results in billion-scale applications. There is also an MST-based Hierarchical Clustered Nearest Neighbor Graph (HCNNG) [63], which uses MST to build a tree and uses SPTAG-like methods to build the graph.

4.4.2 Graph Construction

The first component of any graph-based ANNS method is to construct a graph $G(V, E)$ from the input vectors, with some specified properties. [47] classifies graph construction process into three main categories: divide and conquer, refinement, and incremental. While there is variation even among these three categories, many steps are similar.

Divide and conquer methods typically initialize the construction by dividing the input vectors into several subdivisions, and then recursively constructing a graph on each subdivision. Refinement methods usually start with an initial graph that is created by some other ANNS algorithm, such as a KD-tree, and then iteratively refine the graph structure to improve accuracy. Incremental methods do not have any specific initialization, but may use some "seed" nodes to start the graph construction process.

Neighbor selection is a common task performed by many graph-based ANNS methods. Some algorithms first choose a candidate neighbor set for each node, which may not include the exact nearest neighbors but will be refined later. Algorithms such as SPTAG and HCNNG take the entire vertex set as candidate neighbors. Other algorithms use the algorithm itself as an ANNS to prune the candidate neighbor set. To select the final neighbor set $N(u)$ for node u, refinement methods are used which could depend on the distance with candidate neighbors or some other metrics. However, not all algorithms perform an explicit neighbor selection step, and some rely on the graph structure to compute approximate nearest neighbors.

Some algorithms also have extra steps such as seed selection, which is used during query to improve efficiency. The method of selecting these seeds can be deterministic, such as the k-means algorithm, or probabilistic, such as Skip Lists. A final step in graph construction is generally ensuring connectivity, which can be performed in various ways depending on the algorithm. Some methods rely on the graph construction algorithm to ensure connectivity,

while others use post-processing steps or perform a traversal to ensure connectivity.

4.4.3 Graph Traversal to Find Nearest Neighbors

To find the approximate nearest neighbors of a query point q in a graph-based ANNS algorithm, the first step is to choose a seed set $\hat{S}$, which is a subset of the original input set S. This seed set can be either dynamically selected for each query or fixed after graph creation via seed preprocessing. In some cases, the algorithm may even modify the seed set to a limited extent.

Once the seed set is fixed, the graph is traversed from $\hat{S}$ to points close to q. This is typically done using a form of greedy best-first search. The algorithm keeps two sets, a set of seen nodes C and a set for the result set R, both initialized to $\hat{S}$. At each iteration, the algorithm finds the node $\hat{x} = \arg\min_{u \in C} \delta(q, u)$, which is the node in C that is closest to q, and checks its neighbors. The neighbors are then inserted into sets C and R. However, the sizes of these sets are limited to improve efficiency. The result set R is restricted to k elements for k nearest neighbors, while C is restricted to some specified parameter. If the insertion of a new neighbor causes the set sizes to grow beyond the restrictions, then the node with maximum distance to q is erased. When C is no longer updated, the process ends and R is returned as output.

Despite the restricted size of C and R, the traversal can still visit too many nodes, which can be computationally expensive. Therefore, algorithms generally use heuristic-based pruning in the search method to make the algorithm more efficient. HCNNG proposes variation to the search where instead of picking all neighbors of a node, a guided search is done to avoid redundant revisits. Another disadvantage of best first search is that the search can get stuck in local optima. To mitigate this issue, some algorithms perform the search with different seed sets to increase the chances of finding better solutions. HNSW performs the seach in every layer of its graph using previous layers results set as seed set for next layer. FANNG provides backracking to make sure other nodes can be searched in case of local optima.

Chapter 5

Experimental Evaluation

Due to nearest neighbor search being a vital part of many modern day problems, there have been many libraries and frameworks built explicitly for approximate nearest neighbor search. Some of these libraries were listed in the first chapter. Each library has its own tradeoffs and focus on particular algorithms. To compare the various methods of ANNS, we used Facebook AI Similarity Search (FAISS) which is a robust tool for nearest neighbor search supporting hashing-based, quantization-based and graph-based methods.

In the following sections we will describe the datasets, evaluation metrics and algorithms used for our experiments.

5.1 Datasets

We use three real-world datasets which are embeddings of several problem domains. The first two datasets, SIFT and GIST are standard datasets for nearest neighbor search [64]. The third, MNIST, is used for various other purposes including nearest neighbor search. Each of these are based on Euclidean distance as we have previously seen, other two major metric distances can be converted to Euclidean metric. We briefly describe the datasets below:

SIFT: SIFT dataset consists of one million images of covering a wide range of objects [65]. The images are represented as a set of scale-invariant feature transform features which are expressed as a 128 dimension floating point vectors. The dataset provides 10000 query vectors to use for querying with ground truth information of 100 nearest neigbor for each query vector.

GIST: Similar to SIFT, GIST consists of descriptions of one million images for object recognition [6]. The images are represented by 960 dimension floating point vectors using a different approach than SIFT. The dataset also provides 1000 query vectors to query with them. Ground truth is provided for 100 nearest neighbor for the queries.

Table 5.1: Summary of datasets

Dataset	Vector Dimension	Input Vector Count	Query Vector Count	Nearest Neighbor Count
SIFT	128	1000000	10000	100
GIST	960	1000000	1000	100
MNIST	784	60000	10000	100

MNIST: MNIST dataset [66] is a popular benchmark dataset for various machine learning tasks. It consists of 60000 images of handwritten digits. Although the images are presented as 28×28 grayscale images, they can be used for nearest neighbor search by using the images as a 784 dimension floating point vectors. 10000 images are provided as test images, which we use as query vectors. The ground truth is computed via brute force for nearest 100 neighbors.

We used the above three databases, with $k = 50, 100$, to generate our results. Table 5.1 shows summary of the datasets.

5.2 Evaluation Metrics

Nearest neighbor search has several possible factors to consider depending on the application. Generally four metrics are used to describe the possible tradeoffs. They are described below:

Train Time: Each algorithm is associated with creating a "index" structure. These structures are created, index is trained (all algorithms might not need training) and input vectors are added. The time taken for this whole preprocessing step is train time. Algorithms should have a small enough train time so that they can be used with minimal startup delay.

Memory: Most libraries including FAISS allows the index structures to be written to disk and loaded later. The memory size of this write is the memory required, also called index size. While memory is generally cheap, structures should not have so large memory footprint that it is outside machine capabilities.

Query Speed: We generally call NNS algorithms with a large number of query vectors. The average number of query vectors processing in one second is the query speed or queries per second. This can vary drastically across algorithms. We generally want an algorithm with high speed.

Recall: Recall means the proportion of true outputs returned by an algorithm. In kNNS, if the true output set is R and our algorithm returns the set S, then recall is $\frac{|R \cap S|}{|R|} = \frac{|R \cap S|}{k}$. Recall is the measure of approximation in our algorithms. High recall is preferred.

It is impossible to determine an algorithm which is best in all of these metrics. We show the tradeoffs through scatter plots.

5.3 Algorithms used

FAISS provides several "index" structures that are used to compute the nearest neighbors. We have selected some of these representing the methods described in chatper 4. The chosen indexes are briefly described below:

FlatL2: The naive brute algorithm which stores the input vectors without any change. This is used mainly to create a base to compare to.

LSH: Hashing method, in particular LSH. A random rotation is first performed on the input vectors then the vectors are converted to hashsed code which are used to compute distance. The index depends on a parameter `nbits`, which is the number of bits the hashed codes should have.

IVFADC: IVFADC method as described in [11]. Inverted file is kept with several clusters mapping to coarse centroid. Each cluster has fine product quantization. The cluster lookup is done via naive loop. This index depends on several parameters: `ncentroid`, number of centroids in coarse quantization; `pqparam`, number of subspaces in residual vectors; `nprobe`, number of clusters used for lookup. We have fixed number of bits used in subspace code to be 8, but FAISS allows it to be changed too.

IMIADC: Inverted Multi Index as described in [40]. Similar to IVFADC, but the first coarse quantization is another product quantization. The first quantization uses fixed 2 subspaces with 8 bits each. The remaining part is determined by two parameters: `pqparam` and `nprobe`. These are same as IVFADC. The number of bits in fine quantization is also fixed to 8. Additionally, we have used OPQ as a preprocessing step.

HNSW: Hierarchical Navigable Small World Graph. HNSW generally shows most satisfactory results among graph-based methods. This index depends on three parameters: `maxdegree`, the maximum degree of any node in the graph built; `efconstruction`, maximum number of neighbors to explore when constructing the graph; `efsearch`, maximum number of neighbors to explore when searching nearest neighbors of a query vector.

IVFHNSW: A composition of IVFADC and HNSW methods. The coarse quantization step is done by building a HNSW graph on the centroids. The index depends on three parameters: `maxdegree`, maximum degree of nodes in HNSW; `nprobe`, `ncentroid`, both same as other IVF variants.

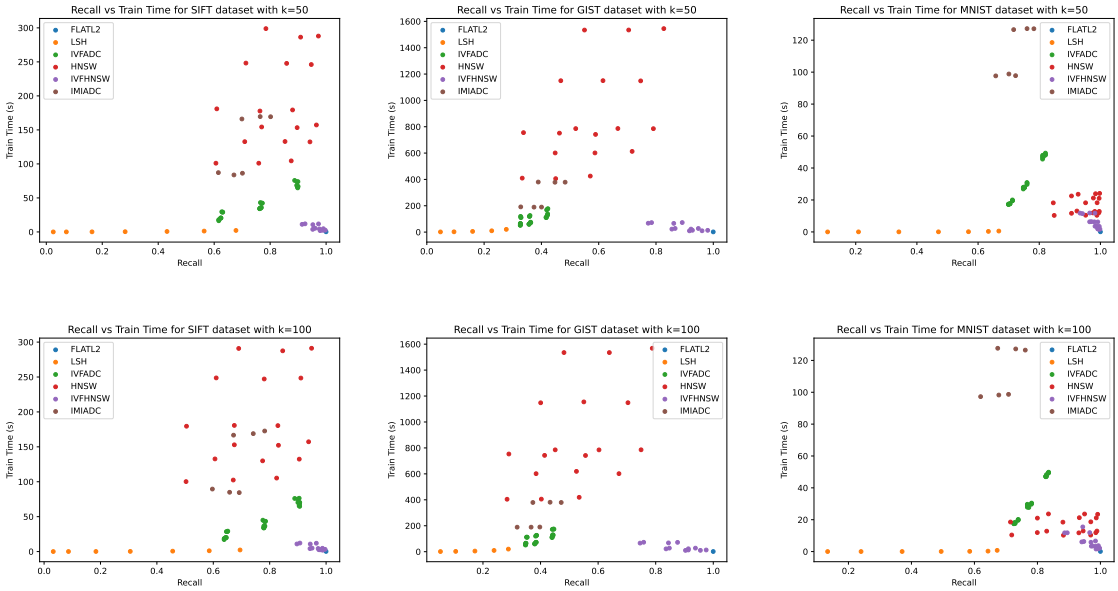

Figure 5.1: Recall vs Train Time for three datasets and $k = 50, 100$. Points toward right and down are better.

The chosen indexes are run on all three datasets two times, once for $k = 50$, once for $k = 100$.

5.4 Results

We have run the chosen algorithms for several different combination of parameters on each of the three datasets for $k = 50, 100$. The machine used was equipped with an Intel Core i5-9400F CPU, featuring 6 cores and a base clock frequency of 2.9 GHz. The system also had 31 GiB of 2133 MHz RAM along with an Nvidia RTX 2070 GPU with 8GB of memory. The evaluation metrics have been plotted in scatter plots with two metric on two axes. Since correctness of output is a significant value to consider, we have compared each of train time, queries per second, index size to recall.

For the algorithms, it is noted that for hashing methods, increasing `nbits` results in higher recall. This is intuitive as more bits in hashed space means more information is retained. For inverted file based algorithms, increase in `nprobe` increases recall as it controls the number of centroid clusters searched. keeping `nbits` fixed, increasing `ncentroid` decreases recall as vectors per centroid decreases. In HNSW based methods, higher `efconstruction` and `maxdegree` results in superior graphs with higher recall. `efsearch` impacts searching and increasing it also increases recall. Finally, `pqparam` controls the number of subspace in product quantization, increasing it gives more granular quantization and returns higher recall.

In fig. 5.1, we notice that HNSW and IMIADC suffer from high train time. In particular HNSW can take a considerable train time to return satisfactory recall. LSH, Flat indexes have almost no

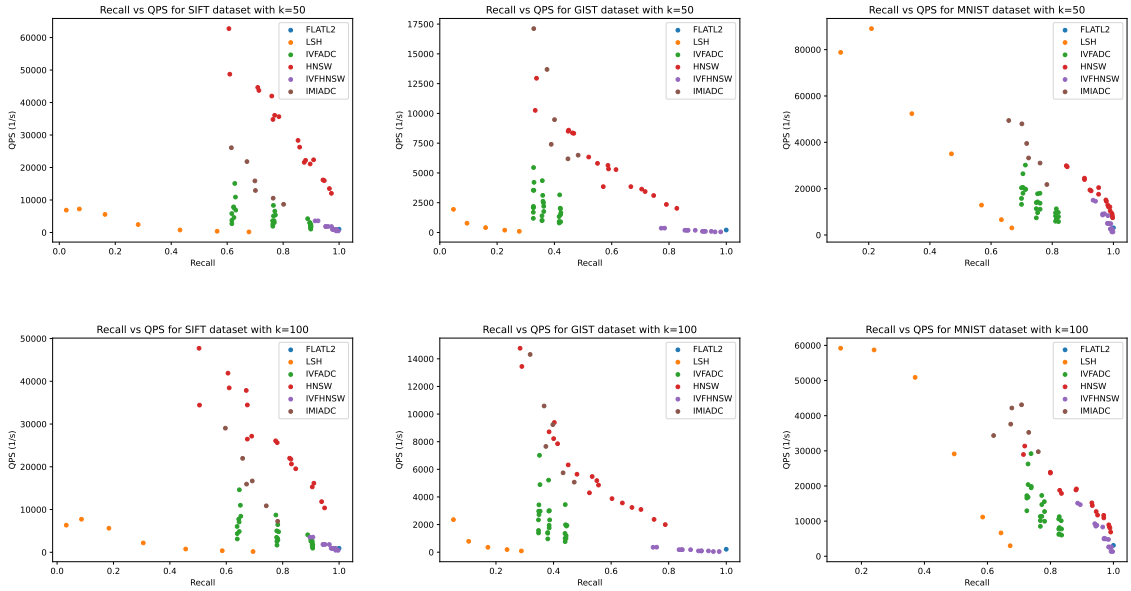

Figure 5.2: Recall vs Queries Per Second for three datasets and $k = 50, 100$. Points toward right and up are better

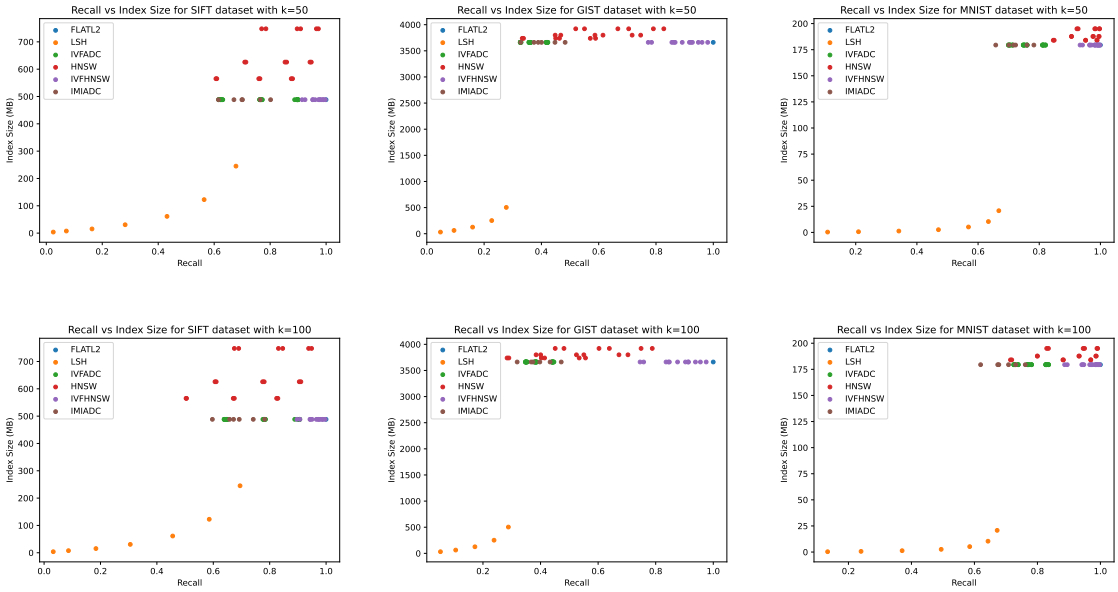

Figure 5.3: Recall vs Index Size for three datasets and $k = 50, 100$. Points toward right and down are better

train time overhead. IVFHNSW has little overhead, while IVFADC has increasing time taken with respect to higher recall.

In fig. 5.2, we can see that LSH generally provides unsatisfactory query speed and recall. IVFHNSW also provides similar speed but at high recall. HNSW provides best query speed with respect to high recall. IMIADC, IVFADC give similar speed but IMIADC is relatively better.

Finally, we consider fig. 5.3. All three inverted file based algorithms generally have similar fixed memory size unaffected by parameters. LSH requires increasing memory, lower than inverted files, but is unsatisfactory in other metrics. HNSW has the largest varying index size. This hinders application of HNSW in datasets of huge size.

Chapter 6

Discussion and Future Work

As we have previously stated, there is no one algorithm that is best suited for solving the approximate nearest neighbor search problem. However, our experiment results provide an overview of the tradeoffs offered by different methods. In particular, hashing-based methods do not produce satisfactory metrics in query speed or recall, which are the two most important factors. Nonetheless, the ideas related to hashing can still be useful in future works. Product quantization and HNSW graph are the most promising algorithms to use.

Of these, HNSW graph shows considerable speed while ensuring high recall. However, it has two disadvantages: a long training time and large memory size. If the application has memory restrictions or needs to have a short initialization time, it is better to consider inverted multi-index, IVFADC or the composite IVFHNSW. IVFHNSW is useful if application requires low initialization time. But other two will provide better query speed.

Another thing to consider is the scale of data and hardware. Billion scale data combined with GPU acceleration can provide very different results. Performance of various methods in billion scale could be a potential research topic. However, for datasets similar to the ones we use, HNSW graph or inverted multi-index are possibly the best algorithms to consider.

While our experiments have provided valuable insights into the performance of various approximate nearest neighbor search algorithms, there is still much work to be done in this field. One direction of future research could focus on usage of deep learning in the methods described. While some recent works do propose deep learning to improve ANNS, there is yet to be a library or framework utilizing them with wide range of algorithms. Experiments with algorithms augmented by Deep learning could provide even better results.

References

[1] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker, "Query by image and video content: the QBIC system," *Computer*, vol. 28, pp. 23–32, Sept. 1995. Conference Name: Computer.

[2] T. Hastie and R. Tibshirani, "Discriminant adaptive nearest neighbor classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, pp. 607–616, June 1996. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[3] S. Cost and S. Salzberg, "A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features," *Machine Learning*, vol. 10, pp. 57–78, Jan. 1993.

[4] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based Collaborative Filtering Recommendation Algorithms," *Proceedings of ACM World Wide Web Conference*, vol. 1, Aug. 2001.

[5] H. Lejsek, B. Jónsson, and L. Amsaleg, *NV-Tree: Nearest Neighbors at the Billion Scale*. Apr. 2011. Journal Abbreviation: Proceedings of the 1st ACM International Conference on Multimedia Retrieval, ICMR'11 Publication Title: Proceedings of the 1st ACM International Conference on Multimedia Retrieval, ICMR'11.

[6] A. Oliva and A. Torralba, "Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope," *International Journal of Computer Vision*, vol. 42, pp. 145–175, May 2001.

[7] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is "Nearest Neighbor" Meaningful?," in *Database Theory — ICDT'99* (C. Beeri and P. Buneman, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 217–235, Springer, 1999.

[8] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, (New York, NY, USA), pp. 604–613, Association for Computing Machinery, May 1998.

[9] E. Gómez-Ballester, L. Micó, and J. Oncina, "Some approaches to improve tree-based nearest neighbour search algorithms," *Pattern Recognition*, vol. 39, pp. 171–179, Feb. 2006.

[10] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 518–529, Morgan Kaufmann Publishers Inc., Sept. 1999.

[11] H. Jégou, M. Douze, and C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, pp. 117–28, Jan. 2011.

[12] P.-C. Lin and W.-L. Zhao, "Graph based Nearest Neighbor Search: Promises and Failures," June 2019. arXiv:1904.02077 [cs].

[13] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[14] E. Bernhardsson, *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. Python package version 1.13.0.

[15] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[16] P. Cunningham and S. Delany, "k-Nearest neighbour classifiers," *Mult Classif Syst*, vol. 54, Apr. 2007.

[17] Y. Koren, R. Bell, and C. Volinsky, "Matrix Factorization Techniques for Recommender Systems," *Computer*, vol. 42, pp. 30–37, Aug. 2009. Conference Name: Computer.

[18] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, *Large-Scale Parallel Collaborative Filtering for the Netflix Prize*. June 2008. Pages: 348.

[19] D. Lucarella, "A document retrieval system based on nearest neighbor searching," *Journal of Information Science*, vol. 14, pp. 25–33, Feb. 1988.

[20] S. Trotta, L. Flek, and C. Welch, "Nearest Neighbor Language Models for Stylistic Controllable Generation," Oct. 2022. arXiv:2210.15762 [cs].

[21] U. Khandelwal, O. Levy, D. Jurafsky, L. Zettlemoyer, and M. Lewis, "Generalization through Memorization: Nearest Neighbor Language Models," Feb. 2020. arXiv:1911.00172 [cs].

[22] S. Arya, T. Malamatos, and D. Mount, "Space-Time Tradeoffs for Approximate Nearest Neighbor Searching," *J. ACM*, vol. 57, Nov. 2009.

[23] Y. Bachrach, Y. Finkelstein, R. Gilad-Bachrach, L. Katzir, N. Koenigstein, N. Nice, and U. Paquet, "Speeding up the Xbox recommender system using a euclidean transformation for inner-product spaces," in *Proceedings of the 8th ACM Conference on Recommender systems*, RecSys '14, (New York, NY, USA), pp. 257–264, Association for Computing Machinery, Oct. 2014.

[24] M. Dolatshah, A. Hadian, and B. Minaei-Bidgoli, "Ball*-tree: Efficient spatial indexing for constrained nearest-neighbor search in metric spaces," Nov. 2015. arXiv:1511.00628 [cs].

[25] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd international conference on Machine learning*, ICML '06, (New York, NY, USA), pp. 97–104, Association for Computing Machinery, June 2006.

[26] *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, Jan. 1993. Google-Books-ID: a3BFcnmGM4oC.

[27] D. Yan, Y. Wang, J. Wang, H. Wang, and Z. Li, "K-Nearest Neighbor Search by Random Projection Forests," *IEEE Transactions on Big Data*, vol. 7, pp. 147–157, Mar. 2021. Conference Name: IEEE Transactions on Big Data.

[28] P. Ram and K. Sinha, "Revisiting kd-tree for Nearest Neighbor Search," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, (New York, NY, USA), pp. 1378–1388, Association for Computing Machinery, July 2019.

[29] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for Similarity Search: A Survey," Aug. 2014. arXiv:1408.2927 [cs].

[30] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, (New York, NY, USA), pp. 253–262, Association for Computing Machinery, June 2004.

[31] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, (Vienna, Austria), pp. 950–961, VLDB Endowment, Sept. 2007.

[32] H. Jegou, L. Amsaleg, C. Schmid, and P. Gros, "Query adaptative locality sensitive hashing," in *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 825–828, Mar. 2008. ISSN: 2379-190X.

[33] Y. Liuz, J. Cuiz, Z. Huang, H. Liz, and H. Shenx, "SK-LSH : An efficient index structure for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 7, pp. 745–756, Jan. 2014.

[34] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen, "A Survey on Learning to Hash," Apr. 2017. arXiv:1606.00185 [cs].

[35] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Proceedings of the 21st International Conference on Neural Information Processing Systems*, NIPS'08, (Red Hook, NY, USA), pp. 1753–1760, Curran Associates Inc., Dec. 2008.

[36] V. E. Liong, J. Lu, G. Wang, P. Moulin, and J. Zhou, "Deep hashing for compact binary codes learning," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2475–2483, June 2015. ISSN: 1063-6919.

[37] H. Lai, Y. Pan, Y. Liu, and S. Yan, "Simultaneous Feature Learning and Hash Coding with Deep Neural Networks," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3270–3278, June 2015. arXiv:1504.03410 [cs].

[38] R. Gray and D. Neuhoff, "Quantization," *IEEE Transactions on Information Theory*, vol. 44, pp. 2325–2383, Oct. 1998. Conference Name: IEEE Transactions on Information Theory.

[39] J. Sivic and A. Zisserman, "Video Google: A Text Retrieval Approach to Object Matching in Videos," pp. 1470–1470, IEEE Computer Society, Oct. 2003.

[40] A. Babenko and V. Lempitsky, "The inverted multi-index," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3069–3076, June 2012. ISSN: 1063-6919.

[41] Y. Matsui, T. Yamasaki, and K. Aizawa, "PQTable: Fast Exact Asymmetric Distance Neighbor Search for Product Quantization Using Hash Tables," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1940–1948, Dec. 2015. ISSN: 2380-7504.

[42] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized Product Quantization," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, pp. 744–755, Apr. 2014. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[43] J.-P. Heo, Z. Lin, and S.-E. Yoon, "Distance Encoded Product Quantization," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2139–2146, June 2014. ISSN: 1063-6919.

[44] Y. Cao, M. Long, J. Wang, H. Zhu, and Q. Wen, "Deep Quantization Network for efficient image retrieval," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, (Phoenix, Arizona), pp. 3457–3463, AAAI Press, Feb. 2016.

[45] B. Klein and L. Wolf, "End-To-End Supervised Product Quantization for Image Search and Retrieval," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5036–5045, June 2019. ISSN: 2575-7075.

[46] L. Gao, X. Zhu, J. Song, Z. Zhao, and H. T. Shen, "Beyond Product Quantization: Deep Progressive Quantization for Image Retrieval," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Aug. 2019. arXiv:1906.06698 [cs].

[47] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search," May 2021. arXiv:2101.12631 [cs].

[48] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, pp. 345–405, Sept. 1991.

[49] J. Jaromczyk and G. Toussaint, "Relative neighborhood graphs and their relatives," *Proceedings of the IEEE*, vol. 80, pp. 1502–1517, Sept. 1992. Conference Name: Proceedings of the IEEE.

[50] R. Paredes and E. Chávez, "Using the k-Nearest Neighbor Graph for Proximity Searching in Metric Spaces," in *String Processing and Information Retrieval* (M. Consens and G. Navarro, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 127–138, Springer, 2005.

[51] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, Sept. 2014.

[52] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," Aug. 2018. arXiv:1603.09320 [cs].

[53] B. Harwood and T. Drummond, "FANNG: Fast Approximate Nearest Neighbour Graphs," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5713–5722, June 2016. ISSN: 1063-6919.

[54] Masajiro Iwasaki, "NGT (Graph and tree-based method)," May 2023. original-date: 2016-09-01T07:36:57Z.

[55] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang, "SPTAG: A library for fast approximate nearest neighbor search," May 2023. original-date: 2018-09-12T10:42:51Z.

[56] Wei Dong, "KGraph: A Library for Approximate Nearest Neighbor Search," Mar. 2023. original-date: 2015-05-29T12:38:24Z.

[57] C. Fu and D. Cai, "EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph," Dec. 2016. arXiv:1609.07228 [cs].

[58] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He, "Fast and Accurate Hashing Via Iterative Nearest Neighbors Expansion," *IEEE Transactions on Cybernetics*, vol. 44, pp. 2167–2177, Nov. 2014. Conference Name: IEEE Transactions on Cybernetics.

[59] W. Li, Y. Zhang, Y. Sun, W. Wang, W. Zhang, and X. Lin, "Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement (v1.0)," Oct. 2016. arXiv:1610.02455 [cs].

[60] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph," Dec. 2018. arXiv:1707.00143 [cs].

[61] C. Fu, C. Wang, and D. Cai, "High Dimensional Similarity Search With Satellite System Graph: Efficiency, Scalability, and Unindexed Query Compatibility," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, pp. 4139–4150, Aug. 2022. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[62] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node," in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019.

[63] J. Vargas Muñoz, M. A. Gonçalves, Z. Dias, and R. da S. Torres, "Hierarchical Clustering-Based Graphs for Large Scale Approximate Nearest Neighbor Search," *Pattern Recognition*, vol. 96, p. 106970, Dec. 2019.

[64] Anonymous, "Evaluation of Approximate nearest neighbors: large datasets." http://corpus-texmex.irisa.fr/, n.d. Accessed on: 2023-05-09.

[65] D. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, p. 91, Nov. 2004.

[66] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," *IEEE Signal Processing Magazine*, vol. 29, pp. 141–142, Nov. 2012. Conference Name: IEEE Signal Processing Magazine.

Generated using Undegraduate Thesis LaTeX Template, Version 2.2. Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

This thesis was generated on Thursday 30th January, 2025 at 7:12pm.