Nearest Neighbor Search in High Dimensional Space

by

Mingjie Li

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

Centre for Artificial Intelligence
Faculty of Engineering and Information Technology
University of Technology Sydney

December, 2020

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree at any other academic institution except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note: Signature removed prior to publication.

Date: 10/12/2020

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Ying Zhang for his continuous support and guidance throughout my PhD career. Ying is professional, patient and kind. He introduced me to the research area of nearest neighbor search and further provided constant motivation which kept me going. His valuable ideas and suggestions always guided me and broadened my knowledge and horizon in the related areas. I am very thankful to him for his patience and confidence during my PhD study. Even experiencing failures and difficulties in research and life, his constant encouragement and support always kept me optimistic and positive. Additionally, Ying is a good mentor and friend for me. He gave me many useful advices on planing career development.

I would like to thank Prof. Wei Wang for his constructive ideas and invaluable suggestions on the works in this thesis. The insightful discussions with Prof. Wang always gave me many inspirations. In addition, I would like to thank Prof. Ivor W. Tsang and A/Prof. Lu Qin for the discussions and suggestions on the related topic.

I would like to thank Prof. Xuemin Lin and A/Prof. Wenjie Zhang for supporting the works in this thesis, as most of the works were conducted

in collaboration with them. I thank Prof. Lin for offering a rigorous while interesting research environment.

I am thankful to the faculties and staffs at the school of computer science at University of Technology Sydney. They were very helpful and supportive throughout my PhD study. It was indeed a pleasure to be a part of such an exciting community.

I would like to thank Dr. Xin Cao, Dr. Lijun Chang, Dr. Xiaoyang Wang, Dr. Shiyu Yang, Dr. Zengfeng Huang, and Dr. Yixiang Fang for sharing the ideas and experiences. Thanks to the members from database groups at UNSW and UTS, including Dr. Long Yuan, Dr. Longbin Lai, Dr. Xiang Wang, Dr. Xing Feng, Dr. Xubo Wang, Dr. Haida Zhang, Dr. Yang Yang, Mr. Xuefeng Chen, Dr. You Peng, Dr. Boge Liu, Ms. Xiaoshuang Chen, Mr. Yuren Mao, Dr. Fan Zhang, Dr. Dong Wen, Dr. Dian Ouyang, Ms. Wanqi Liu, Mr. Hanchen Wang, and Mr. Yuanhang Yu, for creating a dynamic and vibrant atmosphere in the labs and in life. I would also like to thank Mr. Daniel Ouyang, Mr. Peng Zhang, and Mr. Xunxiang Yao for their selfless help and care during my PhD life.

Last but not least, I would like to thank my father Mr. Zheng Li, my mother Mrs. Fengying Zhu, my brother Mr. Mingchang Li and my sister Mrs. Jenly Li for their continuous support, encouragement and love during my entire PhD journey and in my life. I am greatly indebted to them.

PUBLICATIONS

- **Mingjie Li**, Ying Zhang, Yifang Sun, Wei Wang, Ivor W. Tsang, Xuemin Lin. An Efficient Exact Nearest Neighbor Search by Compounded Embedding. DASFAA 2018. (Chapter 4)
- **Mingjie Li**, Ying Zhang, Yifang Sun, Wei Wang, Ivor W. Tsang, Xuemin Lin. I/O Efficient Approximate Nearest Neighbour Search based on Learned Functions. ICDE 2020. (Chapter 5)
- Wen Li, Ying Zhang, Yifang Sun, Wei Wang, **Mingjie Li***, Wenjie Zhang, Xuemin Lin. Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement. TKDE 2019. (*Corresponding Author)(Chapter 6)

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Nearest neighbor search (NNS) in high dimensional space is a fundamental and essential operation in applications from many domains, such as machine learning, databases, multimedia and computer vision, to name a few. In this thesis, we investigate both exact and approximate NNS in high dimensional space.

For the exact NNS, we propose an efficient technique which can have a significant speedup over the state-of-the-art exact solutions. Specifically, we first propose a novel compounded embedding technique, by which we achieve a tight distance lower bound for Euclidean distance. Then each point in a high dimensional space can be embedded into a low dimensional space such that the distance between two embedded points lower bounds their distance in the original space. Following the *filter-and-verify* paradigm, we develop an efficient exact NNS algorithm by pruning a large number of candidates using the new lower bounding technique. Comprehensive experiments on many real-world data demonstrate the effectiveness and efficiency of our new algorithm.

In terms of the approximate NNS, we propose an external memory-based approximate NNS algorithm by learning to hash. Specifically, we introduce a novel data-sensitive indexing and query processing framework for approximate NNS with an emphasis on optimizing the I/O efficiency, especially, the sequential

I/Os. The proposed index consists of several lists of point IDs, ordered by values that are obtained by learned hashing functions on each corresponding data point. The functions are learned from the data and approximately preserve the order in the high-dimensional space. We consider two instantiations of the functions (linear and non-linear), both learned from the data with novel objective functions. Comprehensive experiments on six large scale high dimensional datasets show that our proposed methods with learned index structure perform much better than the state-of-the-art external memory-based approximate NNS methods in terms of I/O efficiency and search accuracy.

Although lots of approximate NNS algorithms have been continuously proposed in the literature each year, there is no comprehensive evaluation and analysis of their performance. Therefore, we conduct a comprehensive and systematic experimental evaluation for the state-of-the-art approximate methods. Our study (1) is cross-disciplinary (i.e., including 19 algorithms in different domains, and from practitioners) and (2) has evaluated a diverse range of settings, including 20 datasets, several evaluation metrics, and different query workloads. The experimental results are carefully reported and analyzed to understand the performance results. Furthermore, we propose a new method that achieves both high query efficiency and high recall empirically on majority of the datasets under a wide range of settings.

Chapter 1

Introduction

Nearest neighbor search (NNS) in high dimensional space aims to find a point in a reference database which has the smallest distance to a given query point. It is a fundamental and significant operation in many applications from many domains, such as machine learning, multimedia databases and computer vision. In these applications, each object is usually represented by a point in a high dimensional space. For instance, by utilizing the deep learning technique [71], an image can be embedded into a point in a 4096-dimensional space. Then, for a given image (i.e., a high dimensional point), NNS can be used to identify the most similar one within an image database.

There are two kinds of NNS problems, exact NNS and approximate NNS. Exact NNS aims to find out the truth nearest neighbors for given queries while the approximate one focuses on retrieving the approximate nearest neighbours for better efficiency. For the users, choosing the exact solution or the approximate one depends on the practical applications and demands. In the application of protein analysis, each protein is represented by a mass spectrum where each spectrum is basically a high dimensional vector. When a new protein is discovered, the scientists need to fully analyze its chemical structure based on the

information from the known proteins. At this point, they prefer to choose exact NNS to find out the most similar spectrums within a protein database. However, in some applications, the approximate NNS is the first choice as it achieves satisfactory results while having a better efficiency. For example, in image retrieval, Google applies approximate NNS techniques alongside the PageRank for large-scale image search [62]. In natural language processing, researchers utilize the approximate NNS method to accelerate the training of large transformer models [68].

In this thesis, both exact and approximate NNS in high dimensional space are investigated, and the Euclidean distance, one of the most popular distance metrics widely used in NNS applications, is used as the similarity measure. We propose an exact NNS algorithm by using the embedding techniques, and an approximate NNS algorithm by learning to hash. Finally, we conduct a comprehensive experimental evaluation for the state-of-the-art approximate NNS methods.

1.1 Exact Nearest Neighbor Search

It is commonly believed that the computation of the exact NNS in high dimensional space *in the worst case* is very expensive due to the *curse of dimensionality* [55]. Despite of the hardness of exact NNS, thanks to the fact that the *intrinsic dimensionality* of the real-life high dimensional data is usually much lower [2, 75], it is still feasible to develop efficient and practical *exact* NNS algorithms for high dimensional real-life data. A variety of exact NNS algorithms have been proposed in the literature. Some of them are based on tree structures such as KD-tree [12], iDistance [56], and Cover-Tree [16]. As reported in [54], they cannot scale to high dimensional data due to the poor performance of tree

structure in high dimensional space.

Existing Methods. OST [76], FNN [54] and HB+ [31] are three most recent exact NNS algorithms in high dimensional Euclidean space. OST proposes an orthogonal search tree using the PCA basis obtained from the data set and then exact NN search is performed on this search tree by following the *filter-and-verify* paradigm. However, OST individually use the PCA without the consideration of the coherence between PCA components. FNN obtains the distance lower bounds between query and data points by nonlinearly embedding the dataset into a low dimensional space according to two important statistics (mean and variance) of the coordinate values of all dimensions. Then NNS can be conducted following the filter-and-verify paradigm by using the distance lower bounds between query and data points in the embedded space. FNN does not need to construct any index, so its preprocessing procedure is very computationally cheap. However, the distance lower bound obtained by FNN is not very tight due to the use of fixed embedded dimensions. HB+ is a newly proposed method which is an extension of HB [100]. The key idea of HB is to devise distance lower bound between a query and a data point by exploiting separating hyperplanes between the query point and the corresponding cluster of the data point. The complexity of the lower bound computation is $O(k^2d)$ where k and d denote the number of clusters and the dimensionality, respectively. This seriously limits the search performance of HB in high dimensional space. HB+ alleviates this issue by accelerating the lower bound computation.

Our Solution. Motivated by the above analysis, in this thesis, we propose a new embedding technique for Euclidean space to devise distance lower bound, which is essential for exact NNS. In a nutshell, we embed data points in the d dimensional space $\mathcal{R}^d$ into a $m+n$ dimensional space, denoted by $\mathcal{E}^{m+n}$ ($m+n \ll d$). The first m dimensions of $\mathcal{E}^{m+n}$ is a *linear* embedding of data points in $\mathcal{R}^d$,

and the last n dimensions are obtained by the *non-linear* embedding method. Specifically, we choose a subspace $\mathcal{S}^m$ of the d-dimensional Euclidean space $\mathcal{R}^d$ with dimensionality m, and the remaining dimensions form the other subspace $\mathcal{S}^{d-m}$. Then each point p in $\mathcal{R}^d$ can be embedded into a $m+n$ dimensional Euclidean space where the coordinate values of the first m dimensions come from $\mathcal{S}^m$ and the coordinate values of the last n dimensions are from n times space partitioning on $\mathcal{S}^{d-m}$. We show that the distance between two points in the embedded space is a lower bound of their distance in the original space.

Following the *filter-and-verify* paradigm, we develop an efficient exact NNS algorithm for high dimensional data by pruning a large number of candidates using the distance lower bound obtained by our embedding technique and hence reducing the expensive cost of distance verification in high dimensional space.

Contributions. The principle contributions in response to handle the exact NNS are summarized as follows:

- We propose a new embedding technique, combining the linear and non-linear methods, for devising the Euclidean distance lower bound. We also show that the linear embedding part of the distance lower bound can be optimized by the PCA technique. Specifically, the linear embedding is directly from the PCA subspace while the non-linear embedding is achieved by space partitioning.

- We develop an efficient *exact* NN search algorithm following the *filter-and-verify* paradigm, by leveraging the new distance lower-bounding technique in low-dimensional space. Specifically, We construct a Cover-Tree [16] index under the embedded space to perform a range query and then the survived candidates will experience the distance verification, among which the distance lower bound still keeps being refined and leveraged for pruning.

- Our comprehensive experiments on 10 real-life high dimensional data demonstrate the effectiveness and efficiency of our proposed techniques. Our algorithm can significantly outperform the state-of-the-art exact NN search techniques in terms of the pruning power and the search time in high dimensional space.

The details of this work are presented in Chapter 4.

1.2 Approximate Nearest Neighbor Search by Learning to hash

Instead of finding the exact nearest neighbors, an enormous amount of research effort has been attracted to the problem of approximate nearest neighbor search (ANNS), which circumvents the curse of dimensionality by the trade-off between the search time and search accuracy. However, most of the existing approximate approaches proposed in the literature are main-memory algorithms which focus on engineering the best trade-off between the CPU cost and the accuracy. In the era of big data, increasing amount of applications are being produced and operated on huge volume of the high dimensional data, where external I/O storage and I/O-centric query processing are needed. For instance, billions of objects (users and items) are mapped to 160-dimensional points via a deep learning model at Alibaba for user recommendation [114]. To handle the large scale data, it is desirable to develop highly efficient external-memory algorithms for better scalability.

Motivation. Existing external memory-based ANNS algorithms are mainly hash-based approaches, which can be further classified into two categories.

1. Random hash based approaches. A number of I/O efficient LSH-based ANNS methods were proposed, such as LSH-forest [109], C2LSH [34],

QALSH [53], which aim to obtain a good trade-off between search accuracy and I/O efficiency with theoretical guarantees. I-LSH [81] is the state-of-the-art I/O efficient random hash based method, which has a small I/O cost by using an incremental, rather than exponentially expanding, search strategy. All these approaches rely on the sorted-lists where each list corresponds to the hashed values of the objects; that is, the objects in the high dimensional space are mapped into multiple sorted lists (i.e., a low-dimensional embedding space) and each list is further divided into consecutive buckets. The query processing consists of a set of sequential scans on these lists/buckets. These approaches are I/O efficient in the sense that sequential I/Os are invoked for the search of these lists/buckets. However, the search quality of these methods is far from satisfactory because they employ random projections, which are independent of the actual data distribution.

2. Learning to hash (L2H) based approaches. These algorithms significantly outperform the random hash based methods [75], as they can learn data-sensitive hash functions to generate high-quality low dimensional embeddings that preserve the locality in the expected sense rather than in the worst-case sense. However, most of these L2H methods are main-memory based, and do not consider the query processing on external memory. To the best of our knowledge, PQBF [83] and AOSKNN [43] are the only two existing L2H methods that aim to optimize the I/O performance where the objects and their embeddings are located in external memory. However, they still require excessive amount of random I/Os, which are much more expensive than sequential I/Os. Moreover, the learning of hash functions in [83, 43], which is the key to the performance of the methods, is independent to the index structure, and hence there is no optimization for the I/O efficiency in the learning of their hash functions.

Our Solution. Motivated by the above analysis, in this thesis, we aim at

designing external-memory based indexing and query processing techniques for ANNS such that we can (i) maximally use the inexpensive sequential I/Os during the search; and (ii) conduct end-to-end learning of the hashing functions, which are also aware of the I/O characteristics.

To address the first goal, after the learned hashing (i.e., mapping) function maps every point to its corresponding low dimensional embedding, we build indexes as a set of M sorted lists. Each entry of a list is of the form (ID, value) which records the object ID and its embedding value on a particular dimension; the list is sorted in ascending order of values. By doing this, the objects in high dimensional space are mapped into M sorted lists by learned mapping functions. Our query processing procedure only makes bi-directional sequential access to each list and hence fully exploits the sequential I/Os (See Fig. 1.1 for an illustration).

To address the second goal, our idea is to leverage machine learning methods to preserve locality of point objects in the embedding space. This is done by designing novel loss function that is aware of the block-based I/O assess characteristics, novel relaxation and optimization techniques. In addition, we consider different models that learn linear hashing functions and non-linear ones, respectively.

Contributions. The main contributions for this work are summarized as follows.

- We develop an I/O efficient external-memory ANNS framework, which consists of a set of sorted lists and a querying processing algorithm.
- We propose two hash learning methods for linear and non-linear hash functions. Particularly, linear hashing method learns the sorted lists separately by penalizing the order mismatch. A fast training method based on stochastic gradient descent (SGD) is developed for optimization. Non-

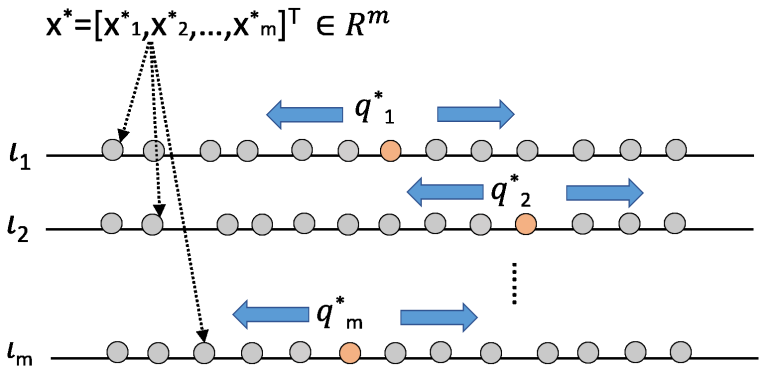


Figure 1.1: Illustration of our idea of index and query processing. Grey points are embedding values of data points, orange points are embedding values of the query point (i.e., $q_1^*, \ldots, q_m^*$).

linear hashing method learns a neural network for our framework. A training architecture based on a fully-connected neural network is constructed for the optimization of the loss function.

- Comprehensive experiments on six large scale high dimensional datasets demonstrate that our proposed methods outperform the state-of-the-art ANNS techniques in terms of I/O efficiency and search accuracy.

The details of this work are presented in Chapter 5.

1.3 Approximate Nearest Neighbor Search: An Experimental Study

Motivation. There are hundreds of papers published on algorithms for approximate nearest neighbor search (ANNS), but there have been few systematic and comprehensive comparisons among these algorithms. In this thesis, we conduct a comprehensive experimental study on the state-of-the-art ANNS algorithms in the literature, due to the following needs:

1. Coverage of Competitor Algorithms and Datasets from Different Areas. As the need for performing ANNS arises naturally in so many diverse domains, researchers have come up with many methods while unaware of alternative methods proposed in another area. In addition, there are practical methods proposed by practitioners and deployed in large-scale projects such as the music recommendation system at `spotify.com` [14]. As a result, it is not uncommon that important algorithms from different areas are overlooked and not compared with. For example, there is no evaluation among Rank Cover Tree [51] (from Machine Learning), Product Quantization [57, 38] (from Multimedia), SRS [106] (from Databases), and KGraph [27] (from practitioners). Moreover, each domain typically has a *small* set of commonly used datasets to evaluate ANNS algorithms; there are very few datasets used by all these domains. In contrast, we conduct comprehensive experiments using carefully selected representative or latest algorithms from different domains, and test *all of* them on 20 datasets including those frequently used in prior studies in different domains. Our study confirms that there are substantial variability of the performance of all the algorithms across these datasets.

2. Overlooked Evaluation Measures/Settings. An ANNS algorithm can be measured from various aspects, including (i) search time complexity, (ii) search quality, (iii) index size, (iv) scalability with respect to the number of objects and the number of dimensions, (v) robustness against datasets, query workloads, and parameter settings, (vi) updatability, and (vii) efforts required in tuning its parameters. Unfortunately, none of the prior studies evaluates these measures completely and thoroughly. For example, most existing studies use a query workload that is essentially the same as the distribution of the data. Measuring algorithms under different query workloads is an important issue, but little result is known. In this study, we evaluate the performance of the algorithms under a

wide variety of settings and measures, to gain a complete understanding of each algorithm (c.f., Table 6.3).

3. Discrepancies in Existing Results. There are discrepancies in the experimental results reported in some of the notable papers on this topic. For example, AGH was shown to perform better than Spectral Hashing in the literature [82], while the study in [98] indicates otherwise. This situation also exists between Spectral Hashing and DSH [98, 36]. While much of the discrepancies can be explained by the different settings, datasets and tuning methods used, as well as implementation differences, it is always desirable to have a maximally consistent result to reach an up-to-date rule-of-thumb recommendation in different scenarios for researchers and practitioners. In this study, we try our best to make a fair comparison of several algorithms, and test them on all 20 datasets. Finally, we will also publish the source code, datasets, and other documents so that the results can be easily reproduced.

Contributions. Our principle contributions are summarized as follows.

- Comprehensive experimental study of the state-of-the-art ANNS methods across several different research areas. Our comprehensive experimental study extends beyond past studies by: (i) comparing all the methods without adding any implementation tricks, which makes the comparison more fair; (ii) evaluating all the methods using multiple measures; and (iii) we provide rule-of-thumb recommendations about how to select the method under different settings. We believe such a comprehensive experimental evaluation will be beneficial to both the scientific community and practitioners, and similar studies have been performed in other areas (e.g., classification algorithms [21]).

- We group algorithms into several categories, and then perform detailed analysis on both intra- and inter-category evaluations. Our *data-based*

analyses provide confirmation of useful principles to solve the problem, the strength and weakness of some of the best methods, and some initial explanation and understanding of why some datasets are harder than others. The experience and insights we gained throughout the study enable us to engineer a new empirical algorithm, DPG, that achieves both high query efficiency and high recall empirically on majority of the datasets under a wide range of settings.

The details of this work are presented in Chapter 6.

Chapter 2

Literature Review

A large amount of methods regarding the exact and approximate nearest neighbour search in high dimensional space have been proposed in the past years and achieved prominent success in terms of the academic research and the industrial applications. This chapter provides a comprehensive literature review for the existing NNS algorithms in community.

2.1 Exact Nearest Neighbor Search

Exact nearest neighbour search (NNS) aims to find out the true nearest neighbors for the given query within a database. In the past, a variety of exact NNS algorithms have been proposed in the literature. Some of them are based on tree structures, such as KD-tree [12], iDistance [56] and Cover Tree [16]. KD-tree, also called k-dimensional tree, is a binary search tree structure for organizing data points in a space with k dimensions. It works by recursively partitioning the data points based on a median value of a chosen attribute. When given a query point, we find the matching leaf of the KD-tree, and compare the query point to all points in the leaf. KD-tree supports approximate and exact NNS,

and have a good performance when the dimensionality of data is low (e.g., 5-10 dimensions). iDistance [56] is a B^+-tree variant, which is an indexing and query processing technique for exact nearest neighbor search on point data in multi-dimensional metric spaces. In the indexing stage, data points in multiple dimensional space are mapped to one-dimensional values (iDistance value), and then B^+-tree can be adopted to index the points using the iDistance as the key. To process a kNN query, the query is mapped to a number of one-dimensional range queries, which are then processed efficiently on the iDistance index. Cover Tree [16] can be thought of as a hierarchy of levels with the top level containing the root point and the bottom level containing every point in the metric space. Cover Tree is efficient for range queries and answering exact k-nearest neighbor search. However, as reported in [54], these algorithms have a poor performance on high dimensional data.

Following the *filter-and-verify* paradigm, another kind of exact NNS algorithms have been proposed and show their advantages in terms of addressing data in high dimensional space, such as OST [76], FNN [54] and HB+ [31]. OST proposes an orthogonal search tree using the PCA basis evaluated from the data set, where the tree depth corresponds to the total number of PCA dimensions used for projection. Particularly, the search tree is constructed in a similar way to KD-tree [12], where the data points are recursively partitioned based on their sorted projection values in each individual PCA dimension. Exact NN search can be conducted on this search tree, where the difference between the query projection and each node is used as distance lower bound for NN candidate pruning, and then the distance verification is executed if the pruning fails. FNN [54] obtains the distance lower bounds between query and data points by nonlinearly embedding the dataset into a 2-dimensional space according to two important statistics (mean and variance) of the coordinate values of all dimen-

Methods	Indexing/Pruning approach	Searching paradigm
KD-tree [12]	Binary search tree based on dimension partitioning	Tree search and verification
iDistance [56]	B^+-tree variant based on dimension reduction	Range query and verification
Cover tree [16]	Hierarchy search tree based on distance bounding	Multi-layer tree search
OST [76]	Orthogonal search tree based on PCA	Filtering and verification
FNN [54]	Distance lower bounding based on non-linear embedding	Filtering and verification
HB+ [31]	Distance lower bounding based on clustering	Filtering and verification

Table 2.1: Overview of the Indexing and Searching of Existing Exact NNS Algorithms

sions. The performance can be enhanced by partitioning the dimensions into t disjoint groups and computing the statistics for each group, where each point is embedded to a $2t$-dimensional space. Then NNS can be conducted following the filter-and-verify paradigm by using the distance lower bounds between query and data points in the embedded space. Table 2.1 summarizes the existing exact methods in terms of their indexing (pruning) techniques and searching paradigms.

Overall, exact NNS methods suffer from the *curse of dimensionality* [55], and currently cannot scale to deal with very large scale data with up to one billion data points. Therefore, in most cases, the problem of approximate NNS enjoys more interests of researchers.

2.2 Approximate Nearest Neighbor Search

The approximate nearest neighbour search (ANNS) focuses on how close the retrieved points are to the query rather than whether the returned points are the true nearest neighbours. It aims to get a trade-off between the search accuracy

and search efficiency by exploring the approximate results. This section classifies the existing ANNS algorithms into three main categories: *Hashing-based*, *Partition-based* and *Graph-based*. Table 2.2 summarizes the existing representative approximate algorithms in terms of their indexing (hashing) techniques and searching approaches.

2.2.1 Hashing-based Methods

The algorithms belonging to this class transform data point to a low-dimensional representation, so each point could be represented by a short code (called hash code). There are two main sub-categories in this class: Locality Sensitive Hashing (LSH) and Learning to Hash (L2H).

Locality Sensitive Hashing

Locality sensitive hashing (LSH) is data-independent hashing approach. The LSH methods rely on a family of locality sensitive hash functions that map similar input data points (distance$< r$) to the same hash codes with higher probability than dissimilar points (distance$> cr$), so LSH methods are initial designed to solve (r,c)-ANN problem. The designing of good locality sensitive hash functions is vital for LSH-related methods. For Euclidian distance measure, a great number of hash functions are proposed [24, 3, 5, 110, 6]. Random linear projections [24, 39, 96, 108] are the most commonly used hash function to generate hash code, in which the random projection parameters are chosen from a 2-stable distribution (e.g. Gaussian distribution).

In order to achieve good search precision, several hash functions are concatenated to form a hash table, thus decreasing the collision probability for dissimilar points. While it also reduces the collision probability of nearby points, so one usually requires to construct multiple hash tables, leading to large memory cost

Type	Method	Indexing/hashing	Searching
LSH-based	SRS [106]	R-tree based on random projection	Incremental k-NN search with early termination examination
	QALSH [53]	B^+-tree based on random projection	Range search with query-aware exponential bucket expansion
	I-LSH [81]	B^+-tree based on random projection	Range search with incremental bucket expansion
L2H-based	PQ [57]	Inverted file based on cartesian product k-means	k-NN search on inverted file with asymmetric distance computation
	PQBF [83]	B^+-tree based on Z-order [108] on PQ codes	k-NN search on a set of B^+-trees
	ITQ [41]	Iterative quantization based PCA strategy	k-NN search based on hamming ranking
	AGH [82]	Graph-based hashing	k-NN search based on hamming ranking
	DeepBit [77]	DCNs with quantization loss minimization	k-NN search based on hamming ranking
	SADH [102]	DCNs with similarity-adaptive optimization	k-NN search based on hamming ranking
Partition-based	Randomized KD-tree [103]	Binary search tree based on hyperplane partitioning	Depth-first search
	FLANN [91]	A algorithm configuration method based on the randomized KD-tree, hierarchical k-means tree and linear scan	k-NN search on the selected algorithm
	VP-tree [18]	Binary search tree based on pivoting partitioning	Range search with decreasing radius
Graph-based	Kgraph [28, 27]	Approximate directed k-NN graph	Greedy search
	DPG [75]	Approximate directed k-NN graph considering the diversification	Greedy search
	SW [87]	Navigable small world graph	Greedy search with multi-restarts
	HNSW [88]	Hierarchical Navigable small world graph	Multi-layer greedy search

Table 2.2: Overview of the Indexing and Searching of Representative ANNS Algorithms

and long query time. Hence, some heuristic methods [85, 65, 86] are presented to check more hash buckets which may contain the nearest neighbor or the candidates near the query point, so as to increase the search quality without increasing the number of hash tables.

Because the hash tables are constructed before searching, the points collided with the query in a part of hash functions in one hash table are neglected although they are likely near. Hence, instead of using "static" compound hash functions to construct hash table before searching, some recent LSH-based methods (e.g. C2LSH [34], LazyLSH [130], QALSH [86]) employ dynamic collision counting scheme for more efficient searching. Very recently, Liu et al. [81] proposes I-LSH to dramatically reduce the I/O cost of approximate NNS with theoretical guarantee. Unlike the previous LSH methods [34, 53], which expand the bucket width in an exponential way, I-LSH adopts a more natural search strategy to incrementally access the hash values of the objects, thus can greatly reduce I/O cost under the same theoretical guarantee.

LSH-based methods are widely studied by the theory community and enjoy the sound probabilistic theoretical guarantees on query result quality (based on distance ratios), efficiency, and index size even in the worst case. Note that the soundness of theoretical guarantees of LSH algorithms relies on the assumption that: given two data points, the hash functions are selected randomly and independently [113].

Learning to Hash

Learning to hash (L2H) methods fully make use of the data distribution to generate specific hash functions, leading to higher efficiency at the cost of relinquishing the theoretical guarantees. The main methodology of Learning to hash methods is similarity-preserving, so that the ANN relationships between the data points

in the original space could be maximally preserved in the hash coding space.

According to the difference of the optimization objective design to preserve similarity, the learning to hash algorithms could be grouped into the following classes: pairwise-similarity persevering class [121, 82, 80, 47, 78], multiwise-similarity persevering class [118, 116, 120], implicitly-similarity persevering class [60, 64] and quantization class [57, 41, 70]. More related references could be found in [117, 115, 119]. Besides similarity persevering criterion, most of the hashing methods require the codes to be balance and uncorrelated.

Many literatures indicate that the quantization algorithms are more efficient than other learning to hash methods. The quantization-based methods seek to minimize the quantization distortion (equal to the sum difference of each data point and its approximation). Product Quantization (PQ) [57] is a popular methods for ANNS, which decomposes the original vector space into the Cartesian product of M lower dimensional subspaces, and performs vector quantization [42] in each subspace separately. A vector is then represented by a short code composed of its subspace quantization indices (i.e., PQ codes). Recently, there are many extensions are proposed to improve the performance of PQ for indexing step [38, 95, 10, 123, 66] and searching step [11, 58, 66, 48]. For example, Optimized Product Quantization (OPQ) [38] use pre-rotation to further minimize the quantization distortion. Additive Quantization (AQ) [9] and Composite Quantization (CQ) [127] are the generalization of PQ and represent a vector as the sum of M D-dimensional vectors where D is equal to the dimension of input data.

Benefiting from the development of deep neural network, deep hashing methods that employ deep learning are widely studied in recent years. As we does not use label information, we only introduce unsupervised deep hashing methods in this paper. More evaluation of supervised deep hashing algorithms could be found in [19]. Semantic hashing [101] is the first work on using deep learning

techniques for hashing, which builds a multi-layers Restricted Boltzmann Machines (RBM) to learn compact binary codes for text and documents. In order to learn the binary codes, most of deep hashing methods design a sign activation layer to produce binary codes and minimize the loss between the compact real-valued code and the learned binary vector [77, 79, 122, 84]. Another solution is to reconstruct the original data. For example, [20, 124] use autoencoder as hidden layers. Thanh-Toan et al. [26] propose to constrain the penultimate layer to directly output the binary.

Because hashing methods must obtain the binary code from the output of hash functions, the binary constraint optimization problem is an NP-hard problem. To ease the optimization, most of the hashing methods adopt the following "relaxation + rounding" approach, which makes the binary codes are suboptimal. For handling this problem, some discrete optimization methods are developed [80, 25, 102].

Most of learning to hash methods are main memory based, that is, all operations including the indexing (hashing) and querying are conducted in main memory for better search efficiency. And also, they cannot trivially be extended to support I/O efficient nearest neighbor search in external memory. Recently, two external memory based approximate algorithms are proposed, which leverage existing learning to hash methods for I/O efficient index construction.

Liu et al. [83] proposed the first I/O-efficient PQ-based solution for ANNS, called PQBF. They design a linear order on the PQ codes by employing the Z-order [108], where a lower bound for the AQD distance (i.e., an approximation of original Euclidean distance) can be achieved. Then they design an index called PQB$^+$-forest to support efficient similarity search on AQD. Specifically, PQB$^+$-forest first creates a number of partitions of the PQ codes by a coarse quantizer and then builds a B$^+$-tree, called PQB$^+$-tree, for each partition. The search

process is expedited by focusing on a few selected partitions that are closest to the query, as well as by the pruning power of PQB$^+$-trees. Note that although the objects in PQB$^+$-tree are indexed by B$^+$-tree, the random I/Os are invoked because it is unlikely to ensure the nearby objects accessed during the search are allocated at the adjacent pages in one order alone.

Gu et al. proposed an external memory-based ANNS algorithm, namely AOSKNN, in [43] based on the "projection-filter-refinement" framework. Specifically, they adopt PCA to embed the high-dimensional point objects into a low-dimensional space. Then, a filter condition is inferred to execute pruning over the projected data. As an R-tree-based index is employed to organize the embedded objects in low dimensional space, random I/Os are invoked during the search.

2.2.2 Partition-based Methods

Methods in this category can be deemed as dividing the entire high dimensional space into multiple disjoint regions. Let the query q be located in a region r_q, then its nearest neighbors should reside in r_q or regions *near* r_q.

The partition process often carry out in a recursive way, so partition-based methods are best represented by a tree or a forest. Tree-based space partition has been widely used for exact and approximate nearest neighbor search in low dimensional space. It can be modified to work for k-ANNS in high-dimensional space. For example, KD-tree [12] is one of the most popular exact NN search method in low dimensional space. While randomized KD-trees are developed from KD-tree to speed up ANN search by building multiple randomized kd-trees as index.

According to the references used in the division, there are three main partitioning approaches: pivoting, hyperplane and compact partitioning schemes.

Pivoting methods partition the points relying on the distances from the point to some pivots (usually randomly chosen). Algorithms in this class contain VP-Tree [126], Ball Tree [22] etc. Hyperplane partitioning methods recursively divide the space by the hyperplanes with random directions (e.g. Annoy [14], Random-Projection Tree [23]) or axis-aligned separating hyperplanes [103, 105]. Compact partitioning algorithms either divide the data into clusters [33] or create possibly approximate Voronoi partitions [93, 16] to exploit locality.

2.2.3 Graph-based Methods

Graph-based methods construct a proximity graph where each data point corresponds to a node and edges connecting some nodes define the neighbor-relationship. The main idea of these methods is *a neighbor's neighbor is also likely to be a neighbor*. The search could be efficiently performed by iteratively expanding neighbors' neighbors in a best-first search strategy following the edges.

According to the difference of the graph structures, graph-based methods are divided into several classes. The first one tries to build a Delaunay Graph in which each node connects with all its "Voronoi neighbors" (the points who share a Voronoi border with the node). However, the Delaunay Graph is proved to be complete graph as the increase of the dimensionality, which is not possible to compute efficiently. A practical approach is to build an exact or approximate k-nearest neighbor graph that records the top-k nearest neighbors for each node. Especially for high dimensional space, the approximate k-nearest neighbor graph construction methods were widely studied recently [44, 35, 129, 28, 128, 125]. With the support of kNN graph, nearest neighbor search is conducted by hill-climbing strategy [13] and usually assigns some random data points as initial enter points, which is easy to get trapped in local optimal. In order to obtain better starting points, some schemes are proposed to locate some initial entries

quickly. For example, IEH [61] and Efanna [32] use hashing and randomized kd-tree to generate the initial candidate neighbors.

The second class is a proximity graph called navigable Small World graph (SW-graph) [17, 69]. Yury et al. [87] proposed NSW method to build SW-graph by iteratively inserted the points where each point is linked to some nodes selected by a greedy search algorithm from the graph in building. The SW-graph contains an approximation of the Delaunay graph and has long-range links together with the small-world navigation property, so it is more efficient for the nearest neighbor search. But the degree of NSW is too high to be efficient and there also exist connectivity problems in it. HNSW [88] is an Extension of SW-graph, which generates a multi-layer proximity graph. HNSW is one of the most efficient ANNS algorithms so far.

Chapter 3

Problem Statement

In this chapter, we formally present the problem definition and then introduce some notations widely used through this thesis.

3.1 Problem Definition

In this thesis, we consider a dataset D which contains N points in a d dimensional Euclidean space $\mathcal{R}^d$ (i.e., $D \in \mathcal{R}^{d \times N}$). We are particularly interested in the high-dimensional case where d is a fairly large number (e.g., $d \geq 100$). Let p denote a point (column vector) in the space $\mathcal{R}^d$ with d dimensions, and we use p_i to represent its coordinate value in the i-th dimension. The Euclidean distance between two points, $\|qp\|$, is defined as $\sqrt{\sum_{i=1}^{d}(q_i - p_i)^2}$. Sometimes, we would use $\|q, p\|$ to denote the distance between q and p to make the formula (context) more readable. We consider the Euclidean distance as our distance metric and throughout the thesis, we use "space" to denote the "Euclidean space". The point vectors used in thesis are all column vectors.

Exact Nearest Neighbor Search. Given a query point q, the *exact nearest neighbor* of q (denoted as o^*) is the point in D that has the smallest distance to

Notation	Definition
$\mathcal{R}^d$	An Euclidean space (d dimensions)
$\mathcal{S}^m$	A subspace of $\mathcal{R}^d$ (m dimensions)
$\mathcal{P}^t$	A t dimensional truncated PCA ($t \ll d$)
$\mathcal{E}^{m+n}$	An embedded space ($m+n$ dimensions)
D	The dataset represented in a $d \times N$ design matrix; each column is a d-dimensional point (object)
p	A data point in original space $\mathcal{R}^d$
p_i	The i-th coordinate value of p
p'	The projection point of p in subspace $\mathcal{S}^m$
p^*	The embedded point of p in space $\mathcal{E}^{m+n}$
$\|pq\|$ / $\|p,q\|$	the Euclidean distance between point p and q
$\overrightarrow{pq}$	The vector from point p to point q
l	A sorted list with entries of form (ID, value)
$\mathcal{H}$	The (learned) mapping function to a lower dimensional embedding space
$\mathbf{W}$	The parameters of the linear mapping functions
$\boldsymbol{\theta}$	The parameters of the non-linear mapping functions

Table 3.1: Summary of Notations

q. We can generalize the concept to the i-th exact nearest neighbor (denoted as o^{*i}). A *k-exact nearest neighbor query* returns the set of $\{o^{*1}, o^{*2}, ..., o^{*k}\}$.

Approximate Nearest Neighbor Search. Instead of returning the exact nearest neighbour, the *approximate nearest neighbor search (ANNS)* focuses on how close the retrieved objects are to the query q. Let the o^* be exact nearest neighbor in D with respect to a query q, and let p be the ANNS result returned by an algorithm, we can measure the quality of p by the ratio of its distance to the query over the nearest neighbor distance, i.e., $\frac{\|pq\|}{\|o^*q\|}$. The ratio is lower, the result is better. It is easy to extend the definition to the top-k version of ANNS.

3.2 Notations

Given any two points p and q in $\mathcal{R}^d$, we use pq to represent the line segment between p and q, and the Euclidean distance between them is denoted by $\|pq\|$.

By $\overrightarrow{pq}$, we denote the vector from p to q, and $\measuredangle abc$ is used to denote the angle between ab and bc. Other important notations frequently used in this thesis are summarized in Table 3.1.

Chapter 4

Exact Nearest Neighbor Search by Compounded Embedding

4.1 Overview

In this chapter, we introduce our proposed embedding technique for efficient exact nearest neighbour search. This work is published in [73] and the rest of this chapter is organized as follows. Section 4.2 introduces our embedding method and the distance lower bound obtained from it. Section 4.3 presents our exact NNS solution based on the embedding techniques. The comprehensive experimental results for our proposed techniques are reported in Section 4.4. Finally, we summarize this chapter in Section 4.5.

4.2 Embedding and Distance Lower Bound

In this section, we first introduce the motivation of our distance lower bound technique, followed by our embedding method, *linear* and *non-linear* embedding. Then we formally show the correctness of our distance lower bound within the

embedded space, followed by its optimization. Finally, we remark that the above techniques can be applied to the PCA space of $\mathcal{R}^d$ to achieve better performance.

4.2.1 Motivation

Suppose the original space $\mathcal{R}^3 = \{w_1, w_2, w_3\}$ is a 3-dimensional space as shown in Fig. 4.1. Given a query point q and a data point p, we use q' and p' to denote their orthogonal projections on a one-dimensional subspace $\mathcal{S}^1 = \{w_1\}$. Now we will show how to devise a distance lower bound for $\|pq\|$ based on $\|p'q'\|$, $\|qq'\|$, and $\|pp'\|$.

Let o be the point in $\mathcal{R}^d$ generated by moving q along the direction of $\overrightarrow{q'p'}$ with distance $\|q'p'\|$. It is trivial to see that qo is parallel to $q'p'$, and $\|qo\| = \|q'p'\|$, thus $\|op'\| = \|qq'\|$. It is easy to know that qo is perpendicular to op, i.e., $\measuredangle qop$ is a right angle. According to the Pythagorean Theorem, we have $\|pq\|^2 = \|qo\|^2 + \|op\|^2$. By Triangle Inequality, we have $\|op\| \geq |\,\|op'\| - \|pp'\|\,|$. Then we have the following distance lower bound for $\|pq\|$:

$$\|pq\|^2 \geq \|p'q'\|^2 + (\|pp'\| - \|qq'\|)^2. \tag{4.1}$$

Let q^* (resp. p^*) be a 2-dimensional point with coordinate values $q_1^* = q_1$ (resp. $p_1^* = p_1$) and $q_2^* = \|qq'\|$ (resp. $p_2^* = \|pp'\|$). Recall that q_1 denotes the coordinate value of point q on the 1st dimension. By doing this, we introduce another dimension to the subspace $\mathcal{S}^1$, resulting in a 2-dimensional embedded space, denoted by $\mathcal{E}^2$. The 1st dimension of $\mathcal{E}^2$ directly comes from $\mathcal{S}^1$, while the 2nd dimension is the distance between the point and its projection on the subspace $\mathcal{S}^1$. The points p and q from $\mathcal{R}^3$ are mapped to p^* and q^* in the embedded space $\mathcal{E}^2$. Then inequality (4.1) can be re-written as follows:

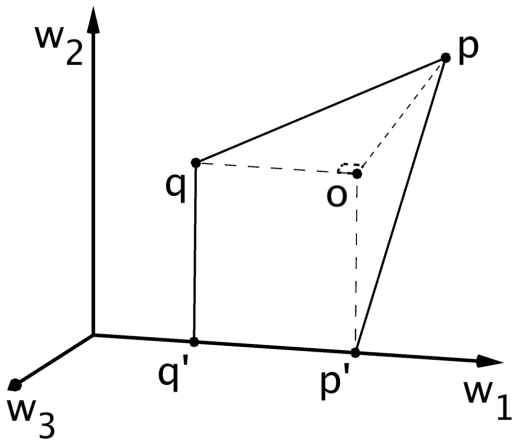


Figure 4.1: Motivation of Distance Lower Bound

$$\|pq\|^2 \geq (p_1 - q_1)^2 + (\|pp'\| - \|qq'\|)^2 = \|p^*q^*\|^2. \tag{4.2}$$

Inequality (4.2) implies that the distance between p and q in the embedded space (i.e., $\|p^*q^*\|$) is always no larger than their distance in the original space (i.e., $\|pq\|$).

In the following three subsections, we formally define our embedding method and then prove the correctness of the distance lower bound in the embedded space, followed by the optimization of the distance lower bound.

4.2.2 Embedding Method

Given the Euclidean space $\mathcal{R}^d = \{w_1, w_2, \ldots, w_d\}$, where $w_1, w_2, \ldots, w_d$ are the orthonormal basis of $\mathcal{R}^d$. We use $\mathcal{S}^m$ to denote a subspace of $\mathcal{R}^d$ with m dimensions ($m \ll d$), and the remaining dimensions form another subspace $\mathcal{S}^{d-m}$. Then, we partition the $\mathcal{S}^{d-m}$ into n ($n \ll d$) disjoint subspaces, denoted by $\mathcal{S}^{e_1}$, $\mathcal{S}^{e_2}$,..., and $\mathcal{S}^{e_n}$, respectively, where $e_1 + e_2 + ... + e_n = d - m$. Then we combine the subspace $\mathcal{S}^m$ and $\mathcal{S}^{e_1}$ together to form a new subspace, denoted by $\mathcal{S}^{m+e_1}$. Accordingly, $\mathcal{S}^m$ and $\mathcal{S}^{e_2}$ are combined together to form the second subspace

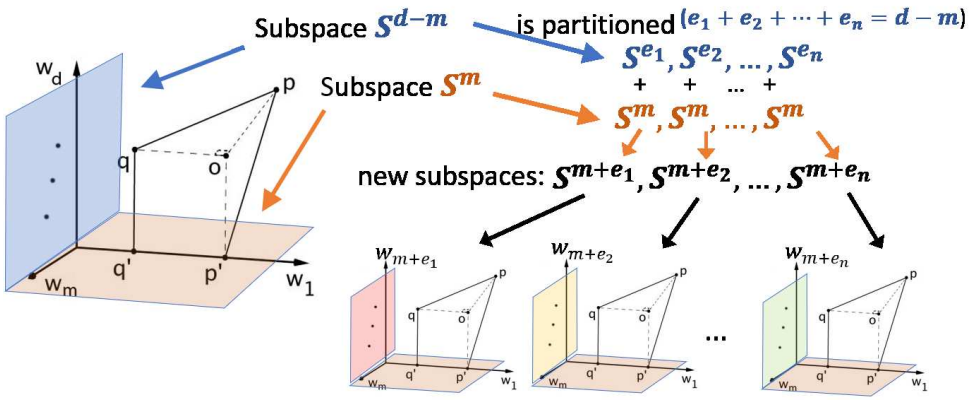


Figure 4.2: Illustration of Our Embedding Method

$\mathcal{S}^{m+e_2}$. Consequently, we generate n new subspaces, denoted by $\mathcal{S}^{m+e_1}$, $\mathcal{S}^{m+e_2}$,..., and $\mathcal{S}^{m+e_n}$, respectively. They share a common subspace $\mathcal{S}^m$ (See Fig. 4.2 for an illustration).

The embedded space regarding these $1+n$ subspaces is a $(m+n)$-dimensional space, denoted by $\mathcal{E}^{m+n}$. Then for each point $p \in \mathcal{R}^d$, we use $\hat{p}^1$, $\hat{p}^2$,..., and $\hat{p}^n$ to denote its corresponding projections on the subspaces $\mathcal{S}^{m+e_1}$, $\mathcal{S}^{m+e_2}$,..., and $\mathcal{S}^{m+e_n}$, respectively. Then for each $\hat{p}^i$, where $i = 1, ..., n$, we use p'^i to denote its projection on subspace $\mathcal{S}^m$. We directly denote p' as the projection of p on $\mathcal{S}^m$, it is trivial that $p' = p'^1 = p'^2 = ... = p'^n$. By p^*, we denote the corresponding embedding of p in the embedded space $\mathcal{E}^{m+n}$, where $p^*_j = p'_j$ for $1 \leq j \leq m$ and $p^*_{m+k} = \|\hat{p}^k p'^k\|$ for $1 \leq k \leq n$. Note that $\|\hat{p}^k p'^k\|$ is the distance between $\hat{p}^k$ and its projection on the subspace S^m. In other words, the embedded space $\mathcal{E}^{m+n}$ contains two parts: linear embedding (the first m dimensions) and non-linear embedding (the last n dimensions).

4.2.3 Correctness of Distance Lower Bound

In this subsection, we formally show that the distance between two points within the embedded space $\mathcal{E}^{m+n}$ can serve as the lower bound of their distance in

original $\mathcal{R}^d$.

Theorem 1. *Given an Euclidean space $\mathcal{R}^d$ with an orthonormal basis $\{w_1, w_2, \ldots, w_d\}$, the subspace $\mathcal{S}^m = \{w_1, w_2, \ldots, w_m\}$ and the corresponding embedded space $\mathcal{E}^{m+n}$, we have $\|p^*q^*\| \leq \|pq\|$ where p and q are two points in $\mathcal{R}^d$ while p^* and q^* are their embedded points in $\mathcal{E}^{m+n}$.*

Proof. (1) When $n = 1$, there is no space partitioning on $\mathcal{S}^{d-m}$. Taking Fig. 4.1 for example, we let q' and p' be the projections of q and p on $\mathcal{S}^m$, respectively. We define o as the point in $\mathcal{R}^d$ such that $\overrightarrow{q'q} = \overrightarrow{p'o}$. Then we can have $\overrightarrow{qo} = \overrightarrow{q'p'}$ and $\overrightarrow{op} = \overrightarrow{p'p} - \overrightarrow{p'o}$. It is trivial that $\overrightarrow{q'p'}$ is within subspace $\mathcal{S}^m$, and $\overrightarrow{p'o}$ and $\overrightarrow{p'p}$ are both within the subspace $\mathcal{S}^{d-m}$. Since $\mathcal{S}^m$ is orthogonal to $\mathcal{S}^{d-m}$, $\overrightarrow{qo}$ and $\overrightarrow{op}$ are perpendicular to each other. According to the Pythagorean Theorem and Triangle Inequality, we can have $\|pq\|^2 \geq \|q'p'\|^2 + (\|q'q\| - \|p'p\|)^2$.

Now consider p^* and q^*, which are the embedded points of p and q in the embedded space $\mathcal{E}^{m+1}$. By definition, we have $p^* = (p', \|pp'\|)$ and $q^* = (q', \|qq'\|)$, then we have:

$$\|p^*q^*\|^2 = \sum_{i=1}^{m}(p'_i - q'_i)^2 + (\|pp'\| - \|qq'\|)^2 = \|q'p'\|^2 + (\|q'q\| - \|p'p\|)^2 \leq \|pq\|^2 .$$

Hence, when $n = 1$, the Theorem 1 holds. This case is equivalent to the inequality Lemma in [76].

(2) When $n = 2$. For $n = 1$, we have $\|q'p'\|^2 + (\|q'q\| - \|p'p\|)^2 \leq \|pq\|^2$. Since $\|q'p'\|^2$ is directly from the subspace $\mathcal{S}^m$ of $\mathcal{R}^d$, we can have:

$$\sum_{i=m+1}^{d}(p_i - q_i)^2 \geq (\|q'q\| - \|p'p\|)^2. \tag{4.3}$$

As $n = 2$, we generate 2 new subspaces $\mathcal{S}^{m+e_1}$ and $\mathcal{S}^{m+e_2}$, respectively, where $e_1 + e_2 = d - m$. For each new subspace, we can have that:

$$\left\|\hat{q}^1\hat{p}^1\right\|^2 \geq \|q'p'\|^2 + \left(\left\|\hat{q}^1q'\right\| - \left\|\hat{p}^1p'\right\|\right)^2, \tag{4.4}$$

$$\left\|\hat{q}^2\hat{p}^2\right\|^2 \geq \|q'p'\|^2 + (\left\|\hat{q}^2q'\right\| - \left\|\hat{p}^2p'\right\|)^2. \tag{4.5}$$

The Eqn. (4.4) and Eqn. (4.5) can be further re-written as:

$$\sum_{i=m+1}^{m+e_1}(p_i - q_i)^2 \geq (\left\|\hat{q}^1q'\right\| - \left\|\hat{p}^1p'\right\|)^2, \tag{4.6}$$

$$\sum_{i=m+e_1+1}^{d}(p_i - q_i)^2 \geq (\left\|\hat{q}^2q'\right\| - \left\|\hat{p}^2p'\right\|)^2. \tag{4.7}$$

After combining the Eqn. (4.6) and Eqn. (4.7), we can have:

$$\sum_{i=m+1}^{d}(p_i - q_i)^2 \geq (\left\|\hat{q}^1q'\right\| - \left\|\hat{p}^1p'\right\|)^2 + (\left\|\hat{q}^2q'\right\| - \left\|\hat{p}^2p'\right\|)^2. \tag{4.8}$$

Finally, we can have:

$$\|pq\|^2 \geq \|q'p'\|^2 + (\left\|\hat{q}^1q'\right\| - \left\|\hat{p}^1p'\right\|)^2 + (\left\|\hat{q}^2q'\right\| - \left\|\hat{p}^2p'\right\|)^2 = \|p^*q^*\|^2. \tag{4.9}$$

Therefore, when $n = 2$, the Theorem 1 holds.

(3) When $n \geq 3$. Based on the cases of $n = 1$ and $n = 2$, it is easy to prove that the Theorem 1 holds for $n \geq 3$, so we omit it here. □

4.2.4 Optimization of Distance Lower Bound

In this subsection, we discuss the optimization of our distance lower bound within the embedded space $\mathcal{E}^{m+n}$. Our distance lower bound is from the combination of *linear* and *non-linear* embeddings, so the optimization consists of two parts.

For the linear case, given a dataset with N points in space R^d, we aim to find a subspace $\mathcal{S}^m$ of R^d that is able to maximize the average square distance of all pairwise projected data points on $\mathcal{S}^m$. This optimization objective is given as follows:

$$Maximize \quad \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}d_{ij}^2. \tag{4.10}$$

where the d_{ij} denotes the distance between the projected point i and j on $\mathcal{S}^m$. The optimization of the above objective leads to our another important theorem, which is given as follows. The proof is similar to the Multidimensional scaling [112].

Theorem 2. *Given a dataset D with N points in space R^d, the subspace $\mathcal{S}^m = \{w_1, ..., w_m\}$ ($m \ll d$) which can maximize the average square distance of all pairwise projected data points on $\mathcal{S}^m$, is the m-dimensional PCA of dataset D.*

Proof. We assume that the dataset D is already centered, i.e., $\sum_{i=1}^{N} x_l^i = 0, for\ all\ l = 1, ..., d$, where the x_l^i denotes the l-th coordinate value of the i-th point. Then we can have that the dataset $\tilde{D}$, which is the orthogonal projection of dataset D on $\mathcal{S}^m$, is also centered, as $\sum_{i=1}^{N} w_l^T x^i = w_l^T \sum_{i=1}^{N} x^i = 0, for\ all\ l = 1, ..., m$. Let $\tilde{x}^i$ denote the projection of data point i on $\mathcal{S}^m$, then

$$\sum_{i=1}^{N} \tilde{x}_l^i = 0, for\ all\ l = 1, ..., m. \tag{4.11}$$

Let b_{ij} denote the inner product between $\tilde{x}^i$ and $\tilde{x}^j$. Since $||\tilde{x}^i - \tilde{x}^j||^2 = (\tilde{x}^i)^T \tilde{x}^i + (\tilde{x}^j)^T \tilde{x}^j - 2(\tilde{x}^i)^T \tilde{x}^j$, we can have:

$$d_{ij}^2 = b_{ii} + b_{jj} - 2b_{ij}. \tag{4.12}$$

The Eqn. (4.11) leads to

$$\sum_{i=1}^{N} b_{ij} = \sum_{i=1}^{N} \sum_{l=1}^{m} \tilde{x}_l^i \tilde{x}_l^j = \sum_{l=1}^{m} \tilde{x}_l^j \sum_{i=1}^{N} \tilde{x}_l^i = 0, for\ all\ j = 1, ..., N. \tag{4.13}$$

With a notation $T = \sum_{i=1}^{N} b_{ii}$, by Eqns. (4.12) and (4.13), we have:

$$\sum_{i=1}^{N} d_{ij}^2 = \sum_{j=1}^{N} d_{ij}^2 = 2T. \tag{4.14}$$

Hence, our optimization objective can be rewritten as:

$$\frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} d_{ij}^2 = \frac{1}{N^2} \times 2NT = 2 \times \frac{1}{N}\sum_{i=1}^{N} b_{ii} = 2 \times \frac{1}{N}\sum_{i=1}^{N} (\tilde{x}^i)^T \tilde{x}^i. \tag{4.15}$$

PCA can be defined as the orthogonal projection of the data onto a lower dimensional linear space, such that the variance of the projected data is maximized [49]. In other words, a m-dimensional PCA of dataset D is a subspace that can maximize the average square distance of the projected data points to their central point. As the dataset D is centered, the projected data points on m-dimensional PCA are centered. Hence, the $\frac{1}{N}\sum_{i=1}^{N}(\tilde{x}^i)^T\tilde{x}^i$ is the optimization objective of PCA. Therefore, our optimization objective is equivalent to that of PCA, Theorem 2 holds. □

In terms of the non-linear case, after the $\mathcal{S}^m$ is decided, we aim to choose a space partitioning way for $\mathcal{S}^{d-m}$, which can tighten the partial distance lower bound contributed by the non-linear embedding as much as possible. However, as the subspace partitioning is quite computationally expensive, in this chapter, we set $n = 2$ and choose a simple partitioning way for $\mathcal{S}^{d-m}$. Specifically, we partition the $\mathcal{S}^{d-m}$ into 2 disjoint but consecutive subspaces with bases $\{w_{m+1}, ..., w_{m+\lfloor (d-m)/2 \rfloor}\}$ and $\{w_{m+\lfloor (d-m)/2 \rfloor+1}, ..., w_d\}$, respectively. Our empirical study shows that this partitioning way already gives us a good performance for all datasets.

4.2.5 Using PCA Technique

Theorem 1 holds for any Euclidean linear space with an orthonormal basis. According to Theorem 2, we choose the t-dimensional truncated PCA of dataset in $\mathcal{R}^d$ for embedding ($t < d$). In contrast to full PCA (i.e., the d-dimensional PCA), choosing a truncated PCA with smaller dimensions can save more preprocess-

ing time, and Theorem 1 still holds in a t-dimensional PCA space. Since the Euclidean distance is preserved under any orthogonal linear transformation [40], we first conduct PCA on the original dataset to achieve the t-dimensional PCA space, denoted by $\mathcal{P}^t$, and then first m ($m \ll t$) dimensions (components) are used as the subspace $\mathcal{S}^m$ for the linear embedding while the remaining $t - m$ dimensions are partitioned into n ($n \ll t$) subspaces for non-linear embedding. Such that our embedded space $\mathcal{E}^{m+n}$ is generated by the compound of the linear and non-linear embedding, on which the NN search will be conducted.

4.3 Efficient Exact NNS Algorithm

This section presents our exact NNS algorithm. We first show the key idea of the algorithm, followed by the detailed implementation of the algorithm. Finally, we present the performance analysis.

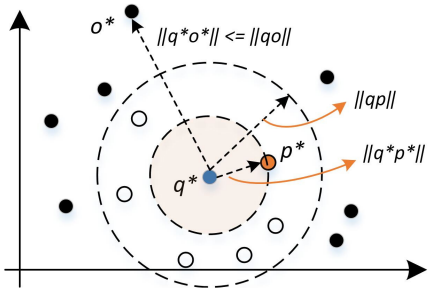


Figure 4.3: Motivation of Exact NNS using Embedded Space

4.3.1 Motivation

Given the embedded space $\mathcal{E}^{m+n}$, a straightforward implementation is to randomly choose a point and compute its distance to the query q as the distance threshold λ. Then for each remaining point, we compute its distance lower bound w.r.t q (i.e., the distance under $\mathcal{E}^{m+n}$), and the point will be safely excluded if

its lower bound is larger than λ. Otherwise, we can compute its true distance to q and update λ. Finally, the point contributing to λ is the exact NN of query q. There are two shortcomings of this implementation: (i) it may take many steps to find a good distance threshold λ, and (ii) we need to compute the distance lower bounds for all data points, which is still expensive even the dimensionality of the embedded space is low.

To address the above issues, we seek help from the multi-dimensional index techniques which can efficiently support exact NNS and range search on relatively low dimensional space (e.g., 8-20 dimensions). Regarding the example shown in Fig. 4.3, we suppose the data points in the embedded space $\mathcal{E}^{m+n}$ are organized by a multi-dimensional index. Then, we can identify the true NN of the query q by issuing one NN search and one range search in the embedded space, together with the verification of the survived candidates. Specifically, we first find the nearest neighbor of q^* (i.e., embedding of q in $\mathcal{E}^{m+n}$) in the embedded space, which is p^* in our example. Then we take $\|pq\|$ as the initial distance threshold λ, and issue a range query from q^* with radius $\|pq\|$. Clearly, we only need to access points within the range because the distance lower bounds of other points are already larger than the current distance threshold. For instance, we do not need to access the point o^* in the example of Fig. 4.3. By doing this, we can find a good distance threshold for better pruning performance and avoid computing distance lower bounds for many points outside of the range search. Detailed implementations will be introduced in the following subsection.

Remark 1. *In this paper, we choose cover tree [16] as the multi-dimensional index structure for the embedded data points due to its good performance under our settings. Note that we use the cover tree to organize the low dimensional embedded points in $\mathcal{E}^{m+n}$, not the high dimensional data points in $\mathcal{R}^d$. It is already reported in [54] that the performance of cover tree in high dimensional*

space is not competitive compared with FNN [54].

Algorithm 1: ***Exact NN Search*** (D, $\tilde{D}$, D^*, q)

Input : D: data points in original space $\mathcal{R}^d$,
$\tilde{D}$: data points in t-dimensional PCA space $\mathcal{P}^t$,
D^*: data points in embedded space $\mathcal{E}^{m+n}$,
q: query point in space $\mathcal{R}^d$,

Output: r : the nearest neighbor of q in D

```
1  q̃ ← transfer q from space R^d to PCA space P^t;
2  q* ← the embedded point of q in E^{m+n};
3  p* ← the nearest neighbor of q* in D*;
4  p ← the corresponding point of p* in D;
5  min_dist := ‖pq‖^2; r ← p;
6  C ← data points {o*} in D* with ‖q*o*‖ ≤ ‖pq‖;
7  for each data point o* ∈ C do
8      if ‖q*o*‖^2 ≥ min_dist then
9          continue;
10     dist := ‖q*o*‖^2 − Δ;
11     õ ← the corresponding points of o* in D̃;
12     is_rejected ← false;
13     for j := (m + 1) → t do
14         dist := dist + (q̃_j − õ_j)^2;
15         if dist ≥ min_dist then
16             is_rejected ← true;
17             break;
18     if is_rejected = false then
19         distance verification in D, that is, dist := ‖oq‖^2;
20     if dist < min_dist then
21         min_dist := dist; r ← o;
22 return r;
```

4.3.2 Exact NNS Algorithm

In Algorithm 1, we illustrate the details of the exact nearest neighbor search (NNS) algorithm by using our embedding technique. In our implementation,

the dataset in original space $\mathcal{R}^d$ is represented by D, and the data points in t-dimensional PCA space $\mathcal{P}^t$, denoted by $\tilde{D}$, are computed off-line on the D. We set the first m dimensions of the embedded space $\mathcal{E}^{m+n}$ as the first m dimensions of the PCA space $\mathcal{P}^t$, and the last n dimensions of $\mathcal{E}^{m+n}$ are from n times space partitioning on $\mathcal{P}^{t-m}$. The corresponding points in the embedded space can also be pre-computed, denoted by D^*. We use a Cover Tree [16] to organize the embedded points, which can efficiently support NN search and range search when $m+n$ is small.

At Lines $1-2$, the query point q is mapped to the PCA space and q^* is its corresponding point in the embedded space. Line 3 retrieves the nearest neighbor of q^* in $\mathcal{E}^{m+n}$, denoted by p^*, by issuing NN search on the cover tree. Note that p^* is the embedded point of p from D. In the algorithm, we use r and min_dist to denote the current closest point of q and its threshold (i.e., squared Euclidean distance), which are initialized at Line 5. Then we issue the range search on the cover tree with centre at q^* and radius $\|pq\|$. The data points within the search range are kept in the set C for further processing (Line 6).

Lines $7-21$ incrementally verify the candidate data points in C. If a data point o cannot be pruned based on its distance lower bound (i.e., $\|q^*o^*\|$) and the current distance threshold min_dist, we need to compute its distance to q in the original space R^d. To reduce the verification cost, we do not directly compute the distance between o and q in $\mathcal{R}^d$. We have $\tilde{p}_i = p_i^*$ for $1 \le i \le m$ because the first m-dimensions of embedded space are from the first m-dimensions of PCA space $\mathcal{P}^t$. Thus, we can reuse the computation of the distance lower bound $\|q^*o^*\|$ by ignoring the contribution of the last n dimensions, denoted as Δ, at Line 10. As such, we can accumulatively compute the distance in PCA space $\mathcal{P}^t$ and immediately terminate the distance computation when the distance computed so far already exceeds the distance threshold (Lines $13-17$). If the verification

in PCA space $\mathcal{P}^t$ fails, we conduct the distance verification in original space $\mathcal{R}^d$ (Lines 18 – 19). Meanwhile, we update the distance threshold as well as the corresponding data point (Lines 20 – 21). The nearest neighbor will be returned after all candidate points are explored. Our empirical study shows that only a small part of candidate points in C need to experience the verification in $\mathcal{R}^d$.

Remark 2. *Algorithm 1 can be easily extended to support k nearest neighbor search (kNNS). Instead of NN search, we issue a kNN search at Line* 3 *and* p^* *is the k-th nearest neighbor of* q^* *in the embedded space. Then the distance threshold is the distance of the k-th closest data point to q seen so far.*

4.3.3 Performance Analysis

The dominant cost of the pre-processing phase is the PCA computation. In the literature, many research efforts have been devoted to develop efficient PCA computation, and various exact and approximate algorithms have been proposed [63, 45, 90]. In our implementation, we use the popular randomized PCA computing algorithm [45], which takes $O(Nd\log t)$ time to compute the PCA space $\mathcal{P}^t$ ($t < d$). In this chapter, the t is set to be 60 for all datasets. It takes $O(Ndt)$ time to transform data points from space $\mathcal{R}^d$ to the PCA space $\mathcal{P}^t$. Regarding the embedded data points in $\mathcal{E}^{m+n}$ ($m + n \ll t$), we simply take the first m dimensions of the data points in $\mathcal{P}^t$ (i.e., their projections in the subspace $\mathcal{P}^m$). The computation of the last n dimensions (i.e., the distances from partitioned subspaces to the subspace $\mathcal{P}^m$) takes time $O(Ndn)$. The embedded data points are organized by cover tree with construction time $O(c^6 N \log N)$, where c is related to the intrinsic dimensionality of the $m + n$ dimensional embedded data [16].

The dominant cost of NN search comes from the pruning (Lines 3 – 6) and verification (Lines 7 – 21). The pruning phase consists of NN search and range

search on the embedded data. The NN search and range search costs on the cover tree are bounded by $O(c^{12} \log N)$ and $O(c^{12} e \log N)$, respectively, where e is the number of points within the range search [16]. Note that, in the worst case, it takes $O(N(m+n))$ time to compute distance lower bounds for all data points. Regarding the verification phase, it contains the verifications in PCA space $\mathcal{P}^t$ and original space $\mathcal{R}^d$, respectively, and it takes $O(t+d)$ time in the worse case to compute the distance for each survived candidate point.

The analysis for the space complexity of our proposed algorithm is quite straightforward. As the whole dataset is loaded into main memory for pre-processing and query processing, the space cost of our proposed algorithm is dominated by $O(Nd)$.

4.3.4 Discussion

Here, we discuss some important issues related to our exact NNS algorithm.

Extension to approximate NN search. Algorithm 1 can also be easily extended to approximate k nearest neighbor search (k-ANNS). In stead of verifying all candidates, we can limit the size of the candidate set by only keep T data points with smallest lower bound distances. Our empirical study shows that its performance is competitive to the well-known ANNS algorithm FLANN [91] for high recall. This indicates that distance lower bound obtained from the embedded space well preserve the distance in high dimensional space and can provide a good access order for NNS.

Use of advanced PCA techniques. A natural question is that if we can apply advanced PCA techniques (e.g., PCA-tree) to enhance the performance of NN search. Same as [1], instead of one PCA space, we iteratively find multiple PCA spaces for data points to ensure that each point can better fit its corresponding PCA space. However, our initial empirical study indicates that, in addition to the

high demanding pre-computational cost, the existence of multiple PCA spaces cannot improve the search performance due to the overhead incurred. It will be an interesting direction to investigate how to effectively incorporate advanced orthogonal transformation techniques to further enhance our algorithm.

4.4 Experiments

In this section, we report and analyze the experimental results for our proposed exact NNS algorithm.

4.4.1 Experimental Settings

Algorithms. We choose the following exact algorithms for conducting Euclidean distance based exact NN search on high dimensional data for comparison.

- LNL is the proposed algorithm. The abbreviation stands for **L**inear and **N**on-**L**inear embedding. The cover tree used in our algorithm is from the cover tree source code[1].
- OST is a PCA-based exact NNS method proposed in [76]. We implement OST and make its performance as good as possible.
- FNN [54] is the state-of-the-art method for exact NN search on high dimensional data. The source code of FNN is public available[2].
- HB+ is a cluster-based exact NNS algorithm proposed in [31]. The source code is obtained from authors.

[1] `http://hunch.net/~jl/projects/cover_tree/cover_tree.html`
[2] `http://research.yoonho.info/fnnne`

We do not show the comparison with cover tree [16], ANN [7] and optimized brute force, since [54] has shown that FNN outperforms these algorithms.

Datasets	N	d	**Data Type**
Audio	53,387	192	Audio
Cifar	50,000	512	Image
Deep	1,000,000	256	Image
GoogleNews	2,999,800	300	Text
Sun	79,106	512	Image
Gist	982,677	960	Image
MNIST	69,000	784	Image
Trevi	99,900	4096	Image
Nusw	268,643	500	Image
Youtube	346,194	1,770	Video

Table 4.1: Dataset Summary

Datasets. We use 10 real-life datasets with different types, including image data (**Cifar**[3], **Deep**[4], **Sun**[5], **Gist**[6], **MNIST**[7], **Nusw**[8], and **Trevi**[9]), audio data (**Audio**[10]), text data (**GoogleNews**[11]), and video data (**Youtube**[12]); they are also widely used in prior research literature to evaluate nearest neighbor query performance. Table 4.1 summarizes these datasets. For each dataset, after deduplication, we randomly select 200 data points and reserve them as the query points.

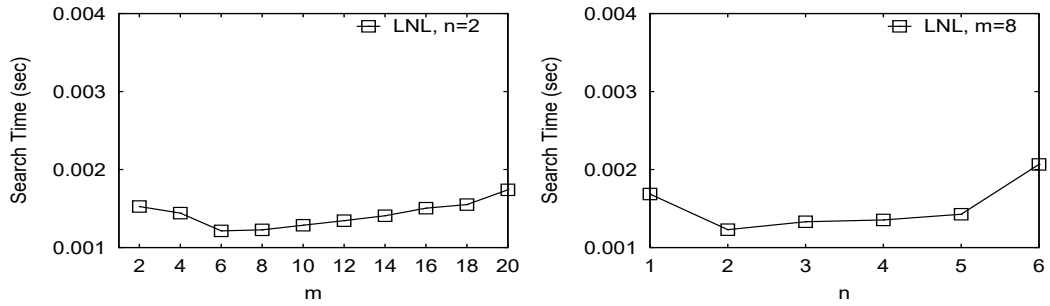
Implementation details. All algorithms are implemented in standard C++ and compiled with G++ with -O3 in Linux. All experiments are performed on a machine with Intel Xeon 3.33GHz CPU and Redhat Linux System, with 32G

[3] http://www.cs.toronto.edu/~kriz/cifar.html
[4] https://yadi.sk/d/I_yaFVqchJmoc
[5] http://groups.csail.mit.edu/vision/SUN/
[6] http://corpus-texmex.irisa.fr
[7] http://yann.lecun.com/exdb/mnist/
[8] http://lms.comp.nus.edu.sg/research/NUS-WIDE.htm
[9] http://phototour.cs.washington.edu/patches/default.htm
[10] http://www.cs.princeton.edu/cass/audio.tar.gz
[11] https://code.google.com/archive/p/word2vec/
[12] http://www.cs.tau.ac.il/~wolf/ytfaces/index.html

main memory. In the experiments, all datasets and indices are fit in the main memory and we report the average NN search time, average kNN search time and preprocessing time.

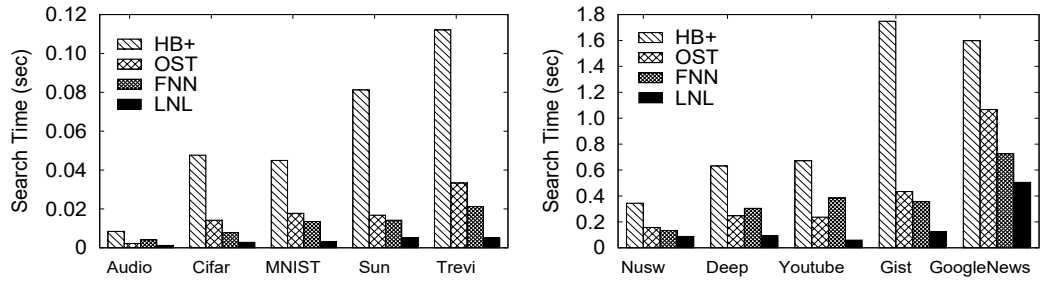
4.4.2 Performance Evaluation

In this subsection, we compare the performance of OST [76], FNN [54], HB+ [31] and the proposed LNL on the exact NNS task. We also conduct some experiments to explore the potential of approximate version of LNL.



(a) Audio (b) Audio

Figure 4.4: Search time with respect to m and n



(a) medium size datasets (b) large size datasets

Figure 4.5: Comparison of search time on all datasets

Parameter settings. For OST, FNN and HB+ algorithms, we set their experimental parameters according to the suggestions by authors.

Next, we determine the value of m and n in LNL by conducting experiments on **Audio** dataset. The other datasets follow the similar trends. Note that the parameter t of the truncated PCA space $\mathcal{P}^t$ is set to be 60 for all datasets. We firstly investigate the impact of m on NN search time with n set to be 2. Fig. 4.4 (a) shows the trade-off between m and the average search time of LNL. Then, we study the influence of n on NN search time with $m = 8$, which can be seen in Fig. 4.4 (b). We can observe that increasing the m or n can tighten the distance lower bound, but it will lead to more other computation costs from lower bound computation and space partitioning. In our experiments, we choose $m = 8$ and $n = 2$ for all datasets to avoid manually tuning, as this setting does achieve a relatively good performance over all datasets.

Efficiency of NN search. Fig. 4.5 reports the searching performance of the four algorithms on all datasets. It is clear that LNL outperforms the other three methods on all the 10 datasets, especially on the high dimensional datasets, such as **Gist**, **Trevi** and **Youtube**. On most datasets, LNL can achieve 2 to 6 times faster than OST and FNN, and 7 to 20 times faster than HB+. For example, LNL needs on average 0.0032s to process a query on **MNIST** dataset while OST, FNN and HB+ need 0.0178s, 0.0135s and 0.04497s, respectively. Further more, the range query allows LNL to avoid computing distance lower bound for all data points, which can reduce a certain amount of searching cost. Note that both LNL and OST use PCA technique, however, OST individually use the PCA on a search tree without the consideration of the coherence between PCA components. Hence, OST is not competitive to the proposed LNL.

Pruning power. We show the pruning power of four algorithms in Fig. 4.6. Each kind of bar represents the verification ratio of each algorithm, which is the percentage of the verified points after the filtering. It can be seen that LNL has a much stronger pruning power than OST, FNN and HB+ on most datasets. Even

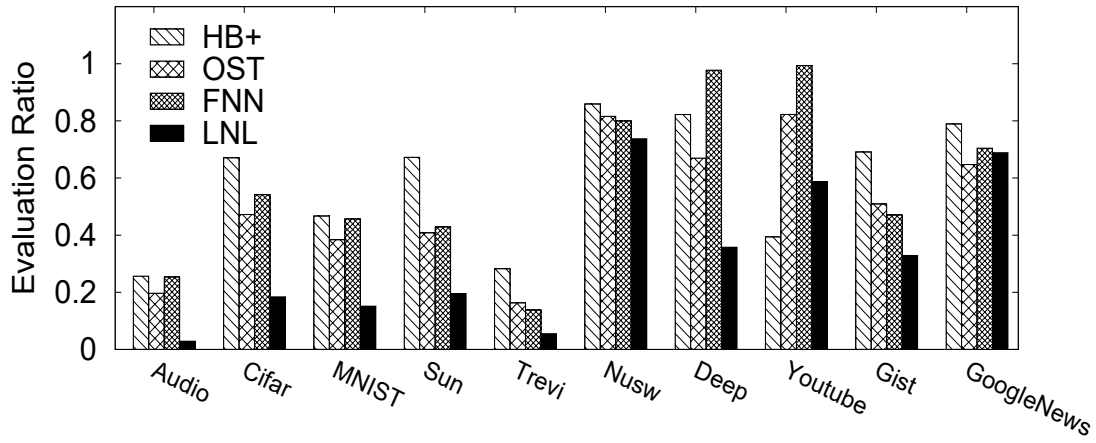


Figure 4.6: Pruning performance (lower the better)

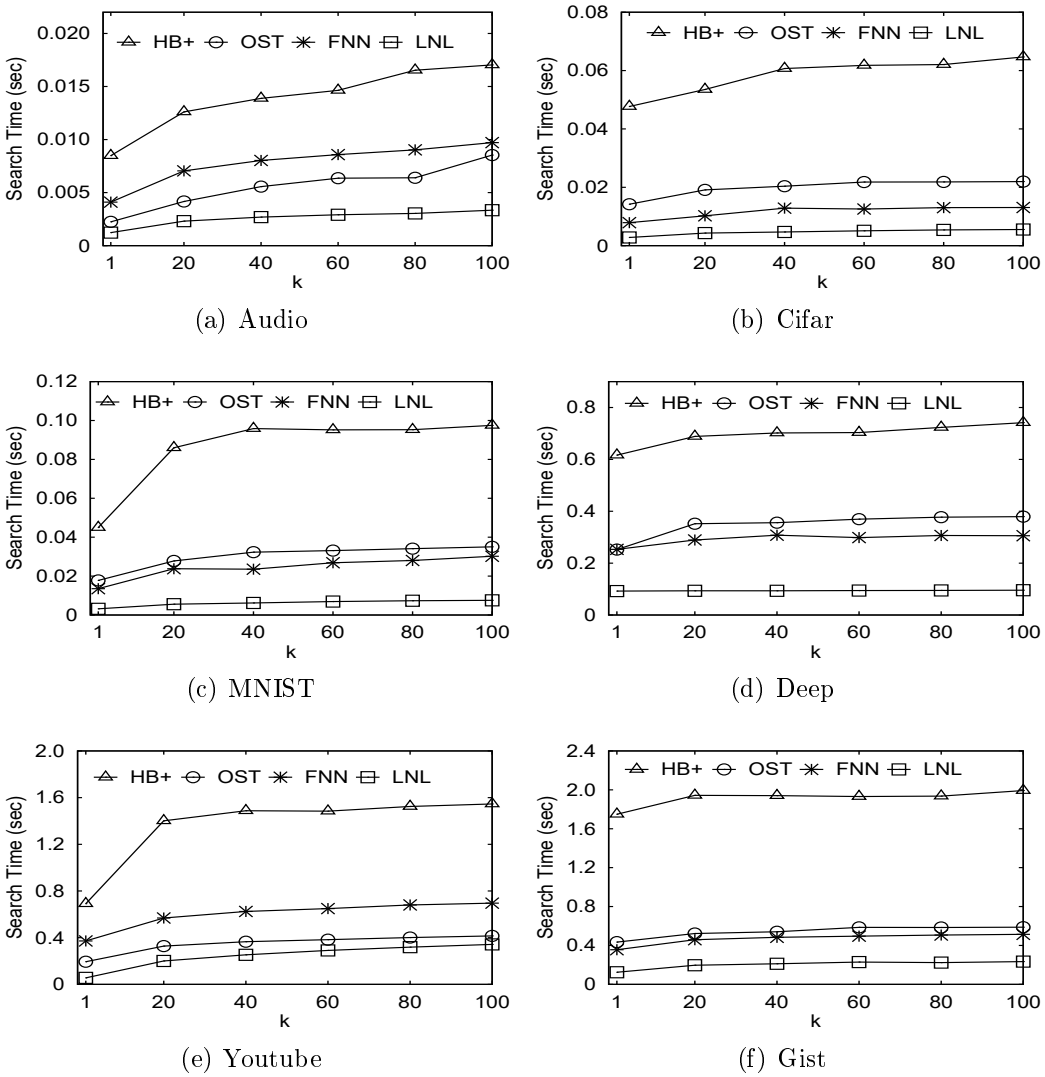
on some datasets where the pruning power of the other three algorithms is very poor, LNL can still filter more than 50% points, such as **Deep** dataset. This is because of two reasons. On one hand, LNL tends to select the point that is closer to the query as the start point in NNS. On the other hand, for the same points, LNL usually has a much tighter distance lower bound than the other three algorithms.

We can see that the verification ratio of LNL is a little higher than HB+ and OST on **Youtube** and **GoogleNews**, respectively. However, the searching time of LNL is much less than these two algorithms, which is due to the efficiency of our verification.

Comparison with respect to k. Fig. 4.7 shows the trends of the performance of four algorithms for kNN search with increasing k on six datasets. Although they are all not sensitive to k, especially when k is large (e.g., ≥ 20), LNL has the most stable performance when k changes. This is because the pruning power of LNL decreases slower than that of other three methods when k increases. We can observe that the proposed LNL still performs the best among the other tree algorithms on the kNN search task.

Pre-processing time. We show the pre-processing time for all four algorithms

in Fig. 4.8. FNN has the smallest preprocessing time, as it does not need to construct any index but only need to compute the mean and variance of the elements for each partition in the vector space. HB+ takes the highest pre-processing time because it needs to construct the clusters which is quite time consuming. The preprocessing time of LNL and OST are similar, since both



(a) Audio (b) Cifar

(c) MNIST (d) Deep

(e) Youtube (f) Gist

Figure 4.7: Comparison with respect to k

of them need the PCA computation and index structure. Note that although LNL requires more pre-processing time than FNN, it is still faster than HB+

and is practical for this *off-line* procedure. For example, LNL can process one million 960-dimensional data points (e.g., **Gist**) in 3 minutes. Considering of the excellent search performance of LNL, it offers an attractive tradeoff between *off-line* and *online* processing times.

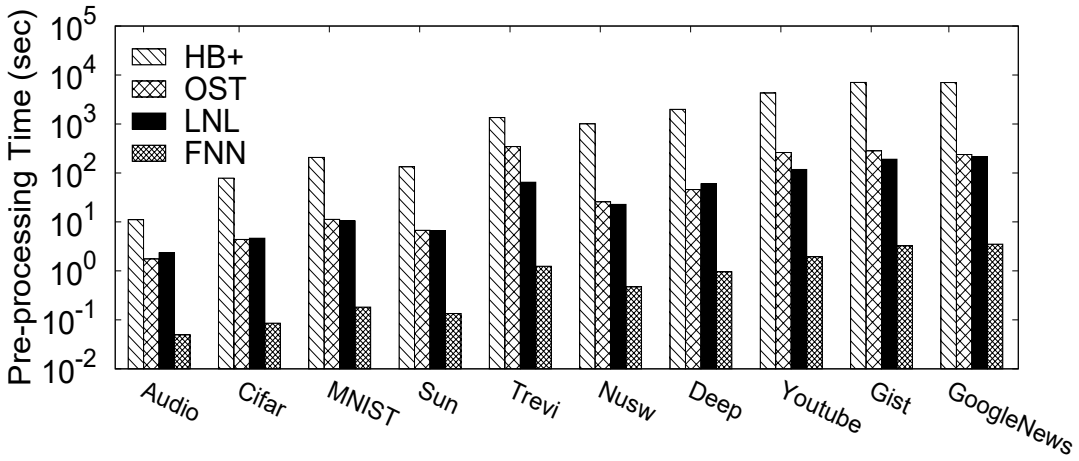
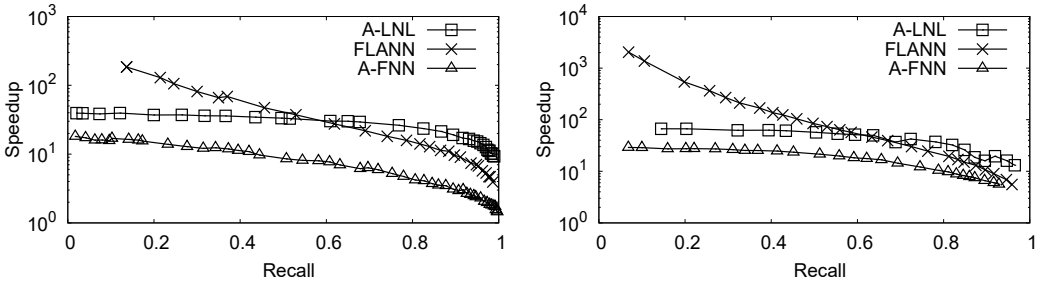


Figure 4.8: Pre-processing time



(a) Cifar (b) Gist

Figure 4.9: Speedup with respect to recall

Approximate Nearest Neighbor Search. We also explore the potential of LNL by evaluating its approximate version, denoted by A-LNL, which is described in the discussion (Subsection 4.3.4). The approximate version of FNN, namely A-FNN, is also modified in the similar way. We also include FLANN [91], which is a widely-used and efficiently implemented library for fast *approximate* nearest neighbor search (e.g., FLANN is part of the OpenCV library).

We verify the approximate NNS performance by speedup with respect to recall with k = 20. Let the results returned by an approximate and an exact algorithm be $X = \{x_i | 1 \leq i \leq k\}$ and kNN = $\{x_i^* | 1 \leq i \leq k\}$, respectively. The recall is defined as $\frac{|X \cap \text{kNN}|}{k}$. We let the speedup be $\frac{t_{BF}}{t_A}$, which is the search time of algorithm A over that of the linear scan algorithm. For A-LNL and A-FNN, we vary T to obtain the speedup versus recall curves. For FLANN, we achieve different recall by tuning the number of verified points.

Fig. 4.9 demonstrates the speedup with respect to the recall of three algorithms on **Cifar** and **Gist**. We can observe that our algorithm is much better than A-FNN, and also outperforms FLANN when the recall is over 0.6. This is mainly due to the high quality of the candidate points of A-LNL. For example, in order to achieve 95% recall on the **Cifar** dataset, A-LNL only needs to verify less than $1,000$ points while the number of verified points for FLANN and A-FNN are $1,800$ and $11,000$ respectively.

4.5 Conclusion

In this chapter, we investigate the problem of exact nearest neighbour search (NNS) in high dimensional space. We design a new embedding method which can embed the high dimensional points into a low dimensional space. Then an efficient exact NNS algorithm is developed following the *filter-and-verify* paradigm based on the embedded data. Extensive experiments on 10 real-life datasets with various types demonstrate that our method significantly outperforms the state-of-the-art exact NNS techniques in high dimensional space. Moreover, we empirically demonstrate that the distance lower bound achieved by our embedding method suggests a good access order of data points during NNS.

Chapter 5

Approximate Nearest Neighbor Search By Learned Functions

5.1 Overview

In this chapter, we introduce our proposed learning to hash methods for I/O efficient approximate nearest neighbour search (ANNS). This work is published in [74] and the rest of this chapter is organized as follows. Section 5.2 introduces our ANNS framework, including the indexing and query processing methods. Section 5.3 and Section 5.4 present the technical details of our linear and non-linear hashing methods that are designed for the proposed ANNS framework. Some discussions related to our models are presented in Section 5.5. The results of experimental evaluation for our proposed techniques are reported and analyzed in Section 5.6. Finally, we summarize this chapter in Section 5.7.

5.2 Our ANNS Framework

This section introduces the details of our proposed indexing and query processing methods for ANNS, followed by the performance analysis.

5.2.1 Our ANNS Solution

We also use the idea of dimensionality reduction which is similar to the action in previous chapter, that is, using a mapping (i.e., hashing) function[1] to map d-dimensional points to m-dimensional points, where $m \ll d$. This is also known as *embedding* in Machine Learning. Specifically, if we denote the mapping function $\mathcal{H} : \mathcal{R}^d \to \mathcal{R}^m$, then $x^* = \mathcal{H}(x)$ is the corresponding embedding for point x. We also slightly abuse the notation $D^* = \mathcal{H}(D)$ to obtain the embeddings as a matrix in $\mathcal{R}^{m \times N}$, where $\mathcal{H}$ is applied to each column of the input design matrix D.

We then index the values of the embedded vectors on each dimension individually as a sorted list. We perform the same embedding process for the query q and then use sequential I/Os to get T candidates. Finally, we perform re-ranking on the T candidates followed by verification to return the top-k nearest points to the query in a progressive manner.

Obviously, our method has several advantages such as leveraging sequential I/Os, and simple and flexible enough to be implemented within a database system where the sorted lists can be easily organized using B^+-trees. The key to make such simple method effective is the quality of the mapping function. We will provide details on how to learn such data-sensitive mapping functions, both linear and non-linear ones, in Sections 5.3–5.4.

Indexing. After applying the mapping function $\mathcal{H}$, we obtain the collection of embedded vectors D^*. We then index each dimension values in a sorted list,

[1] In this chapter, we use *mapping function* and *hashing function* exchangeably when the context is clear.

Algorithm 2: ***Indexing***(D, $\mathcal{H}$, m)

Input : D : The dataset as a design matrix in $\mathcal{R}^{d \times N}$,
$\mathcal{H}$: The learned mapping function,
m : The dimensionality of the embeddings (also the number of sorted lists)
Output: $\mathcal{L}$: The m sorted lists

```
1 D* ← H(D);
2 for i ← 1 to m do
3 |   Y, I ← sort (D*, i) according to the i-th dimension;   /* Y, I ∈ R^{m×N},
  |   where Y contains the dimension values and I contains the
  |   associated IDs */;
4 |   l_i ← an empty list;
5 |   for j ← 1 to N do
6 |   |   l_i[j] ← (Y[i, j], I[i, j]);
7 return L = { l_1, l_2, ..., l_m };
```

which results in m sorted lists. The details of our indexing method can be seen in Algorithm 2. Each entry in the list consists of only a point ID and a dimension value, which is typically 8 bytes.

Nonetheless, we have the option to further reduce the index size by almost 50% by exploiting the external memory access characteristics. Since we consider external memory scenario, the basic unit of access to the index is one page with page size as b bytes. We only need to include the dimension value of the *first* entry on each page, and omit the dimension values of the rest of the entries on the same page. This is similar to the optimization in a clustered index in database systems. In this way, each page consists of a dimension value and $\lfloor \frac{b}{4} - 1 \rfloor$ point IDs. We denote the page as (*IDs*, *value*).

Querying Processing. We show the query processing algorithm in Algorithm 3. The searching operation starts by applying the function $\mathcal{H}$ to the query q to obtain its embedding q^*. Then, it locates the positions where each of the dimension values of q^* will be on the corresponding sorted lists. After that we

insert the pages closest to q^* on each list into a priority queue. In the *while* loop, we obtain the pages according to the priority order and access the corresponding entries. Specifically, we firstly obtain the page with the highest priority, and remove it from the queue. Then we insert the next page closest to q^* on the same list into the priority queue. We access and bookkeep the entries of the current page in the ascending order of their *rank positions* on the list. We define the *rank position* of a point x with respect to the query q on list l_i as:

$$\mathbf{r}_i(q,x) = \sum_{j=1}^{N} \mathbf{1}_{|\mathcal{H}(q)_i - \mathcal{H}(x)_i| > |\mathcal{H}(q)_i - \mathcal{H}(x^j)_i|} + 1, \tag{5.1}$$

where the $\mathbf{1}_p$ is the indicator function which returns 1 if the predicate p is true, and returns 0 otherwise. Intuitively, this is 1 plus how many other points (i.e., x^j) has a smaller distance to query on list l_i than that of x. The closest point will have a rank position of 1. For simplicity, we abbreviate $\mathbf{r}_i(q,x)$ as $\mathbf{r}(x)$ henceforth.

If a point has been seen on all the m list, we add it to the candidate set $\mathcal{C}$. When the size of the candidate set exceeds a preset value T, the searching stops.

The last step is the re-ranking followed by the verification. Although all the T candidates have all been seen on the m lists, this only indicates that they are not too "faraway" from the query in the original space. We would like to reorder them according to some criteria that favor those who are actually close to the query. Here, we adopt a simple re-ranking method: we sum up the rank position of each candidates on all the lists, and reorder all the candidates in ascending order of this sum. Finally, we verify each candidate on the re-ranked candidate list by calculating its distance to the query. We also keep the candidate that has the minimum distance during the verification. Note that this can be easily extended to approximate k-nearest neighbour search (k-ANNS) with the help of

an extra priority queue.

Algorithm 3: ***Querying***($\mathcal{L}$, $\mathcal{H}$, q, T)

Input : q : The query point in $\mathcal{R}^d$,
$\mathcal{L}$: The sorted lists $\{ l_1, l_2, \ldots, l_m \}$,
$\mathcal{H}$: The mapping function,
T : A parameter to control the size of the candidate set (to be re-ranked and verified)
Output: The approximate nearest neighbour of q

```
1  C ← ∅;
2  hits ← an empty hash table;
3  queue ← an empty priority queue;
4  isTerminated ← false;
5  q* ← H(q);
6  for i ← 1 to m do
7      o ← the page closest to q_i^* in list l_i;
8      Insert (o, i) into queue with priority as -|o.value − q_i^*|;
9  while isTerminated = false and queue ≠ ∅ do
10     (o, i) ← the node with highest priority in queue;
11     o* ← the next page closest to q_i^* in list l_i;
12     Remove (o, i) from queue and insert (o*, i) into queue with priority as
       -|o*.value − q_i^*|;
13     for each id in o.IDs do
14         hits[id] ← hits[id] + 1;
15         if hits[id] = m then
16             C ← C ∪ { id };
17             if |C| >= T then
18                 isTerminated ← true;
19                 break;
20 Re-rank the candidates in C and verify their distances to the query;
21 return The point that has the smallest distance to the query;
```

5.2.2 Performance Analysis

In this subsection, we analyze the I/O complexity of the indexing and querying processing of our ANNS framework.

Let b be the page size. The I/O complexity for the indexing is $\mathcal{O}(\frac{Nd}{b} + \frac{Nm}{b})$, whereas the two terms correspond to the cost of learning the data-sensitive mapping function $\mathcal{H}$, and the cost of creating the m sorted list. We estimate the mapping function learning cost as $\mathcal{O}(\frac{Nd}{b})$ as most model learning (including the two we will present in the paper) requires fixed number of iterations over the data. For the query processing algorithm, assume that we access p pages on lists before we collected T candidates, then this phase costs $\mathcal{O}(p)$ *sequential* I/Os, which is equivalent to $\mathcal{O}(\epsilon \cdot p)$ (random) I/Os, where ϵ is typically in the range of $[0.01, 0.1]$. Finally, the verification requires another $\mathcal{O}(\frac{rTd}{b})$ I/Os, where $r \in (0, 1)$.

5.3 Learning to Index by Linear Hashing

In this section, we present the linear method to learn the mapping function from the data. The idea is to preserve the ordering information in the resulting embedding space. We will formally present the objective function based on the order preservation idea, followed by its optimization, involving the relaxation and stochastic gradient descent (SGD) algorithm.

5.3.1 Linear Model and Its Objective Function

We first consider linear mapping functions, which encompass linear projection function considered by learning-to-hash methods [104, 120] and locality sensitive hashing functions [24]:

$$h(x) = \mathbf{w}^\top x \tag{5.2}$$

where $\mathbf{w} \in \mathcal{R}^d$ is the parameter of the hash function and x is a point (column vector). Using m such hash functions with different parameters, we can obtain the mapping function $\mathcal{H}$ as:

$$\mathcal{H}(x) = [h_1(x), h_2(x), \ldots, h_m(x)]^\intercal \tag{5.3}$$

In order to learn these parameters $\mathbf{w}_i$ ($i \in [1, m]$) from the data, we will prepare a set of training data, which consists of uniform samples from the real query workload, or samples from the data itself if the query workload is not available. Next, we need to define the loss function such that we can learn the $\mathbf{w}_i$ values that achieve the minimum loss value.

Consider a given query, for any $\mathbf{w}_i$ value, we can obtain the order induced by rank position of all N data points based on the function h_i (denoted as l_i^s or simply l^s if there is no ambiguity). We can also easily compute the ground truth ordering of all data points based on the distance to query in the original space (denoted as l^o). We would like to define how much penalty to apply if the rank position ordering does not completely agree with the ground truth ordering. Although there exists measures, such as Kendall's tau coefficient, that defines the distance between two orderings, they are not considering the page-based access characteristics during the query processing, not differentiable and are costly to compute. Instead, we design our own measure based on the idea of block order.

We divide the two ordered lists into L parts, called blocks, with each part containing the same number of objects (assuming N is a multiple of L). After the division, we have $l^o = [l_1^o, l_2^o, \ldots, l_L^o]$, and $l^s = [l_1^s, l_2^s, \ldots, l_L^s]$, where l_e^o and l_e^s ($e \in [1, L]$) are the subset of l^o and l^s, respectively. Then for each block, if the objects in l_e^o are not preserved in the corresponding l_e^s, this incurs a penalty of 1, otherwise, the penalty is 0. Therefore, given any l^s induced by $\mathbf{w}_i$, we define

the *loss function* as:

$$J^*(\mathbf{w}_i) = \sum_{e=1}^{L} \sum_{x \in l_e^o} \mathbf{1}_{\mathbf{r}(x) \in [t \cdot (e-1), t \cdot e)} \tag{5.4}$$

$t = \frac{N}{L}$ is the length of the bucket, i.e., the number objects in each partition l_e^o $(1 \leq e \leq L)$. That is, we penalize points that their ranking in the original space and in the embedding space are not in the *same* block. If we let the block size be the page size, then this agrees with our page-based sequential access in the query processing, as all point IDs will be accessed as long as they are in the same page.

Therefore, our *final loss function* for all sorted lists can be written as:

$$J^*(\mathbf{W}) = \sum_{i=1}^{m} J^*(\mathbf{w}_i) + \lambda \cdot \|\mathbf{W}^\top \mathbf{W} - \mathbf{I}\|_F, \tag{5.5}$$

where $\|\mathbf{M}\|_F = \sum_{i,j} \mathbf{M}[i,j]^2$ is the Frobenius norm of a matrix and is a common type of quadratic regularizer [94], λ is the hyper-parameter that controls the degree of regularization, and $\mathbf{W}$ is the concatenation of all $\mathbf{w}_i$ $(i \in [1, m])$. Note that $\mathbf{W}^\top \mathbf{W} = \mathbf{I}$ forces all projection vectors to be orthogonal to each other, such that our model is able to learn m different hash functions.

5.3.2 Relaxation and Optimization

The loss function (5.5) is neither convex nor smooth due to the indicator function. This means that it is hard to optimize the function numerically. We adopt a common approach in Machine Learning that relaxes the discrete function into a continuous and differentiable surrogate loss function and optimize this surrogate instead.

Relaxation

We utilize the fact that the *sigmoid function* (as shown in Fig. 5.1), $\sigma(z) = \frac{1}{1+\exp(-z)}$, is a continuously differentiable approximation to the indicator function $\mathbf{1}_p$.

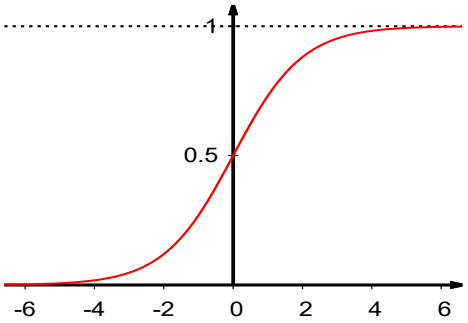


Figure 5.1: The Sigmoid Function

We first replace the absolute value function in (5.1) by taking the square to both sides of the predicate. Afterwards, we apply the sigmoid relaxation and obtain the *approximate rank position* for point x as:[2]

$$\tilde{\mathbf{r}}(x) = \sum_{j=1}^{N} \sigma((h(q) - h(x))^2 - (h(q) - h(x^j))^2) + 1 \tag{5.6}$$

Similarly, the loss functions in (5.4) and (5.5) are also relaxed as

$$J(\mathbf{w}_i) = \sum_{e=1}^{L} \sum_{x \epsilon l_e^o} (\sigma(t \cdot (e-1) - \tilde{\mathbf{r}}(x)) + \sigma(\tilde{\mathbf{r}}(x) - t \cdot e)) \tag{5.7}$$

and

$$J(\mathbf{W}) = \sum_{i=1}^{m} J(\mathbf{w}_i) + \lambda \cdot \|\mathbf{W}^\top \mathbf{W} - \mathbf{I}\|_F. \tag{5.8}$$

[2]We omit i and q in $\tilde{\mathbf{r}}_i(q, x)$ for simplicity.

Progressive Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a fundamental and popular numerical optimization method. To minimize a scalar function with vector parameters, i.e., $f(\mathbf{w})$, the Gradient Descent (GD) algorithm starts with carefully initialized parameter values, and updates it to the next value that maximally minimizes the function value in a small neighborhood by following the negative of the gradient ($\nabla f = [\frac{\partial f}{\partial \mathbf{w}_1}, \ldots, \frac{\partial f}{\partial \mathbf{w}_m}]^\intercal$) direction. SGD is its stochastic version, where the gradient is estimated using a random sample of the training data. SGD drastically reduces the gradient computation cost but may get noisy gradients. In practice, a mini-batched SGD achieves the balance between GD and SGD by estimating the gradient on a mini-batch of B random samples, where B is a hyper-parameter.

Consider our relaxed final loss function (5.8), it is still non-convex. We cannot guarantee to obtain the optimal solution and hence there are several strategies to learn a sufficiently good solution. One strategy is to learn the entire set of parameters $\mathbf{W}$ simultaneously, and another is to learn each function (i.e., the corresponding $\mathbf{w}_i$) one by one in an incremental manner. We found that the latter approach typically leads to better performance and hence we introduce its details below.

Assum that we have learned the first $i-1$ linear hash functions, i.e., $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{i-1}\}$ are learned. We formulate the optimization problem for the next function as

$$J_{\text{inc}}(\mathbf{w}_i) = J(\mathbf{w}_i) + \mu \cdot (\mathbf{w}_i^\intercal \mathbf{w}_i - 1)^2 + \lambda \cdot \sum_{j=1}^{i-1} (\mathbf{w}_j^\intercal \mathbf{w}_i)^2 \tag{5.9}$$

The last two items in Eqn. (5.9) are: (i) The first is the additional regularization that encourages $\mathbf{w}_i$ to be an unit vector; (ii) The second is the orthogonality

regularization. The square is used to guarantee that the last item (i.e., the sum) is always positive.

Next, we apply the SGD algorithm to minimize the $J_{\text{inc}}(\mathbf{w}_i)$. Firstly, we need to obtain the gradient of the $\tilde{\mathbf{r}}(x)$ in (5.6) with respect to $\mathbf{w}_i$, which is:

$$\nabla_{\mathbf{w}_i}\tilde{\mathbf{r}}(x) = \sum_{j=1}^{N}\sigma'(z)\nabla_{\mathbf{w}_i}z \tag{5.10}$$

where

$$\sigma'(z) = \sigma(z)\cdot(1-\sigma(z)) \tag{5.11}$$

$$z = (h(q)-h(x))^2 - (h(q)-h(x^j))^2 \tag{5.12}$$

$$\nabla_{\mathbf{w}_i}z = 2((h(q)-h(x))(q-x) - (h(q)-h(x^j))(q-x^j)) \tag{5.13}$$

Then the gradient of $J_{\text{inc}}(\mathbf{w}_i)$ in (5.9) is:

$$\begin{aligned}\nabla_{\mathbf{w}_i}J_{\text{inc}}(\mathbf{w}_i) = \sum_{e=1}^{L}\sum_{x\epsilon l_e^o}\Big(\nabla_{\mathbf{w}_i}\tilde{\mathbf{r}}(x)\cdot(\sigma'(z_1)-\sigma'(z_2))\Big)\\ + 4\mu(\mathbf{w}_i^\top\mathbf{w}_i-1)\mathbf{w}_i + 2\lambda\sum_{j=1}^{i-1}\mathbf{w}_j^\top\mathbf{w}_i\mathbf{w}_j\end{aligned} \tag{5.14}$$

where the $z_1 = \tilde{\mathbf{r}}(x) - t\cdot e$ and $z_2 = t\cdot(e-1) - \tilde{\mathbf{r}}(x)$.

Now consider using the SGD with a mini-batch of size B, i.e., the mini-batch consists of B objects randomly sampled from the training query set Q. The gradient descent updating rule for $\mathbf{w}_i$ is:

$$\mathbf{w}_i^{(o+1)} = \mathbf{w}_i^{(o)} - lr\cdot\frac{1}{B}\sum_{j=1}^{B}\nabla_{\mathbf{w}_i}J_{\text{inc}}(\mathbf{w}_i, q^j) \tag{5.15}$$

where lr is the learning rate and o is the iteration index, and the$\{q^1,\ldots,q^B\}$ is

Algorithm 4: ***Optimized Incremental SGD***($\bar{D}$, Q, m)

Input : $\bar{D}$: The training dataset,
Q: The training query set,
m: The number of projection vectors
Output: The parameters of the linear mapping functions $\mathbf{W}$

Calculate the ground-truth lists l^o for each query in Q with respect to $\bar{D}$ in $\mathcal{R}^d$;
for $i = 1$ **to** m **do**
  if $i = 1$ **then**
    $\mathbf{w}_i \leftarrow$ a vector where each component is sampled from the Gaussian distribution $N(0, 1)$;
  else
    $\mathbf{w}_i \leftarrow$ a randomly sampled vector from the null space of $\{\mathbf{w}_1, \ldots, \mathbf{w}_{i-1}\}$;
  $\mathbf{w}_i \leftarrow \frac{\mathbf{w}_i}{\sqrt{\mathbf{w}_i{}^\top \mathbf{w}_i}}$; /* normalize */;
  for $j \leftarrow 1$ **to** max_iteration **do**
    Randomly sample a batch of queries from Q and obtain their associated order lists from l^o;
    Calculate the gradient (Eqn. (5.14)) for this batch of queries;
    Update the $\mathbf{w}_i$ (Eqn. (5.15));
return $\mathbf{W} = \{\mathbf{w}_1, \ldots, \mathbf{w}_m\}$;

a random subset of Q. We give out the complete mini-batch SGD algorithm for learning the data-sensitive linear functions in Algorithm 4.

There is still a performance issue with the algorithm in that one iteration over the mini-batch needs $\mathcal{O}(B \cdot N^2)$, which is unacceptable if N is large. We take the following measures to mitigate the performance problem by sub-sampling. Specifically, we can interpret the first item in the gradient in (5.14) as the expectation over all x in the dataset and then approximate it using a set of random samples $\mathcal{S}$:

$$\mathbf{E}_{x \epsilon D} [\![\nabla_{\mathbf{w}_i} \tilde{\mathbf{r}}(x) \cdot (\sigma'(z_1) - \sigma'(z_2))]\!] \tag{5.16}$$

$$\approx \frac{1}{|\mathcal{S}|} \sum_{x \epsilon \mathcal{S}} \nabla_{\mathbf{w}_i} \tilde{\mathbf{r}}(x) \cdot (\sigma'(z_1) - \sigma'(z_2)) \tag{5.17}$$

In addition, instead of using the entire reference dataset, we use a training dataset $\bar{D}$ of size γN, which is randomly sampled from the reference dataset, to compute the rank position and subsequently all the losses. So $\bar{D}$, instead of D, is used to invoke Algorithm 4. Therefore, with the above measures, we can reduce the gradient computation cost to $\mathcal{O}(B \cdot |\mathcal{S}| \cdot \gamma N)$, where $0 < \gamma < 1$.

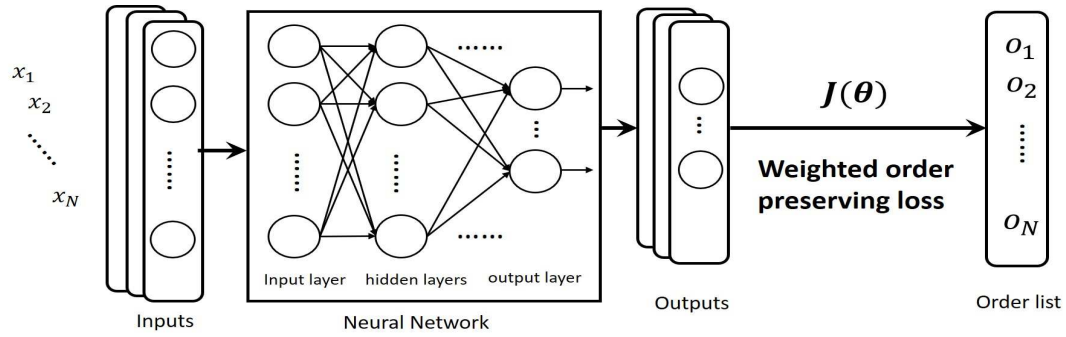


Figure 5.2: The Architecture of Our Non-Linear Hash Learning

5.4 Learning to Index by Neural Network

In this section, we consider learning non-linear mapping functions via Deep Neural Networks (DNN).

5.4.1 DNN Architecture

We choose to model non-linear mapping functions by a DNN, due to its strong modelling power and its success in numerous application areas. However, it is more natural to train the m hash functions collectively, which essentially map the d-dimensional point x into a m-dimensional embedding x^*, or $x^* = \mathcal{H}(x; \boldsymbol{\theta})$, where $\mathcal{H}$ can deemed as a DNN and $\boldsymbol{\theta}$ represents all the parameters of the DNN.

We design the architecture of our DNN as shown in Fig. 5.2. It consists of five fully-connected layers, denoted as: I1-H2-H3-H4-O5. H1 is the input layer which

receives the input dataset features, and the number of input units is equal to the dimensionality of the dataset. H2-H4 are three hidden layers, each of which contains 512 units. ReLU is used as the activation function for each hidden layer. O5 is the output layer containing m units.

5.4.2 Objective Function

Our DNN-based model requires us to design a new loss function rather than those presented in the previous Section. For example, the orthogonality constraints in our previous loss function is inapplicable for DNN.

Nevertheless, we can still apply the same order-preserving idea. We use the following smoothly differentiable surrogate rank position function:

$$\tilde{\mathbf{r}}(x;\boldsymbol{\theta}) = \sum_{j=1}^{N} \sigma(\|\mathcal{H}(q;\boldsymbol{\theta}),\mathcal{H}(x;\boldsymbol{\theta})\| - \|\mathcal{H}(q;\boldsymbol{\theta}),\mathcal{H}(x^j;\boldsymbol{\theta})\|) + 1 \tag{5.18}$$

This function is relaxed from the $\mathbf{1}_p$ function. We penalize how far is the ranking of x away from its groundtruth (in the original space) instead of whether x is preserved in corresponding bucket.

Then, the loss function can be formulated as:

$$J(\boldsymbol{\theta}) = \sum_{i=1}^{N} \beta_i \log\left((\tilde{\mathbf{r}}(x^i;\boldsymbol{\theta}) - g(x^i))^2 + 1\right) \tag{5.19}$$

where the $g(x^i)$ is the ranking of x^i with respect to query q in the original space R^d, β_i is the weight computed for x^i, defined as

$$\beta_i = \exp\left(-\frac{\|q,x^i\|}{\max_{1\le j\le N}\|q,x^j\|}\right) \tag{5.20}$$

$\log(1+z)$ is used to encourage the model to pay more attention to near-by points rather than faraway points.

Finally, we use the Adam optimizer [67] to train the network in a mini-batch manner.

Note that the sub-sampling strategy introduced in the linear model is also applied into our non-linear model to reduce the training cost. Specifically, a training dataset $\bar{D}$ of size γN is used to calculate the rank function $\tilde{\mathbf{r}}(x;\boldsymbol{\theta})$ (5.18), and the gradient of $J(\boldsymbol{\theta})$ (5.19) is computed over a random subset $\mathcal{S}$ of $\bar{D}$. Therefore, the training cost of our DNN-based model over a mini-batch of size B is $\mathcal{O}(\eta \cdot B \cdot |\mathcal{S}| \cdot \gamma N)$, where the η is due to the computation cost of the neural network.

Remark 3. *We remark that the linear model is easier to be optimized, thus, is faster in training than DNN-based model. However, DNN-based model can learn complex non-linear mapping functions, which makes it have a better performance than linear model on many datasets (see Section 5.6).*

5.5 Discussion

In this section, we discuss some issues related to our ANNS framework and the learning to hash models.

For our query processing in Algorithm 3, there are two points need to be discussed. Firstly, one could change the candidate condition such that points that have been seen on more than $\lceil \alpha \cdot m \rceil$ lists are added to the candidate set, reminiscent of the strategy used in MEDRANK [29], C2LSH [34] and QALSH [53]. However, we experimentally found that $\alpha = 1$ always achieves the best performance (see Fig. 5.4(b)). Secondly, we access each list based on the closeness of the pages to the embedded query instead of other choices such as keeping distance lower or upper bounds as in the threshold algorithm [30]. This is because (i) it is not always possible to keep track of the lower/upper bounds (e.g., when

$\mathcal{H}$ is a non-linear mapping function), and (ii) due to the curse of dimensionality, the lower/upper bounds on low dimensional embedding spaces are very loose and do not help much in early stopping the query processing in practice.

Another thing is that can our learning to hash models be applied to other distance metrics and space? In this chapter, we focus on the ANNS in metric space $\mathcal{R}^d$, taking the Euclidean distance as distance metric. The key point of our methods is to map original data points into sorted lists by learned functions, followed by the sequential ANN search. The purpose of our learned functions is to learn the similarity ordering information from the original space. From this perspective, our framework is independent to the actual distance metric. In other words, our models can be easily applied to other distance metrics, such as cosine distance and inner product, by providing the corresponding original distance order (i.e., ground-truth). Nevertheless, it is unclear whether this will lead to good performance for other distance metrics and spaces. We will leave this as a future work.

5.6 Experiments

In this section, we conduct comprehensive experiments to verify the efficiency and effectiveness of our proposed methods, compared with the state-of-the-art I/O efficient ANNS algorithms.

5.6.1 Experimental Settings

Algorithms. We compare the two proposed algorithms with four external memory-based ANNS algorithms. Below are the algorithms evaluated in the experiments.

- I-LSH. The random hash based incremental LSH algorithm proposed in [81].

- PQBF. The ANNS algorithm proposed in [83] where the product quantization technique is employed.
- AOSKNN. The PCA based ANNS algorithm proposed in [43] where the R-tree is employed.
- M-tree. M-tree is a general tree-based data structure that can support ANNS in Euclidean metric space [97].
- OPFA and NeOPFA. Our proposed ANNS algorithms where we learn *linear* and *non-linear* mapping functions (Sections 5.3 and 5.4, respectively) using the relaxed, block-based loss functions.

Datasets and query load. Six widely-used large scale high-dimensional datasets are used for experiments: **Gist**[3], **Deep**[4], **UQvideo**[5], **Tiny**[6], **Deep1B**[7], **Sift1B**[8]. *Gist* is an image dataset which contains about 1 million data points with 960 features. *Deep* contains deep neural codes of natural images, which contains about 1 million data points with 256 dimensions. *UQvideo* is a video dataset with each objects being 256 dimensions. *Tiny* is also a image dataset which consists of around 80 million images, each being a 384 feature vector. *Deep1B* contains 1 billion of 96-dimensional DEEP descriptors [8]. *Sift1M* consists of 1 billion 128-dimensional SIFT feature vectors. For each dataset, after the deduplication, we randomly select 1,000 data points and reserve them as the query points. The details of each dataset are summarized in Table 5.1.

Training and Implementation. For each dataset, we randomly sample two different subsets as training dataset and training query set (i.e., $\bar{D}$ and Q in Algorithm 4, respectively). Specifically, for *Gist*, *Deep* and *UQvideo* datasets, we sample 20k data

[3] http://corpus-texmex.irisa.fr/
[4] https://yadi.sk/d/I_yaFVqchJmoc
[5] http://staff.itee.uq.edu.au/shenht/UQ_VIDEO/
[6] http://horatio.cs.nyu.edu/mit/tiny/data
[7] http://sites.skoltech.ru/compvision/noimi/
[8] http://corpus-texmex.irisa.fr/

Datasets		Statistics		
		N	d	**Data Type**
Million Scale	**Gist**	982,677	960	Image
	Deep	1,000,000	256	Image
	UQvideo	3,038,478	256	Video
	Tiny	79,302,017	384	Image
Billion Scale	**Deep1B**	1,000,000,000	96	Image
	Sift1B	1,000,000,000	128	Image

Table 5.1: Statistics of Datasets

points as training dataset and 10k data points as training query set for each of them, respectively. For *Tiny*, we sample 100k data points and 20k data points as training dataset and training query set. For *Deep1B* and *Sift1B*, we sample 1M data points and 0.2M data pints as training dataset and training query set for each of them, respectively. After sampling, the remaining part of each dataset is regarded test datset for ANNS. The batch size B is set to be 200 and 100 for linear and non-linear learning methods, respectively, for all datasets. The termination of two learning methods depends on the convergence of the training procedure, where the max_iteration of the linear method is set within $[50, 400]$ for the datasets evaluated. When choosing the sub-sample $\mathcal{S}$ (See the end of Subsection 5.3.2), we chose the following strategy: for a given training query q and a training dataset, denote the k-NN points of q as $\mathcal{S}_+$, and the rest of the points as $\mathcal{S}_-$. The final $\mathcal{S}$ consists of the entire $\mathcal{S}_+$ and a random sample in $\mathcal{S}_-$.

Note that, same as [83], we apply the K-means data partition for all datasets in the experiment for better search efficiency. Specifically, the K-means clustering method is leveraged to partition the datasets, and then the learned functions are used to index each partition separately. After that, the partition (i.e., a subset of data points) closest to the query point based on the Euclidean distance is selected for querying processing. The partition number is set to be 10 for *Gist*, *Deep* and *UQvideo*, and 64 for the other three datasets.

Evaluation Metrics. We use 6 important metrics to evaluate the performance of the algorithms: ratio, recall, I/O costs, running time, pre-processing time and index

Parameters	Values
The number of sorted lists (m)	$5, 10, 15, 20, \mathbf{25}, 30$
The number of buckets (L)	$5, \mathbf{10}, 15, 20, 25$
Orthogonality regularization factor (λ)	$1, \mathbf{20}, 40, 60, 80$
Additional regularization factor (μ)	$0.1, 1, \mathbf{2}, 4, 6$

Table 5.2: Parameter Settings of OPFA

size.

- **Ratio**. This is the ratio between distances of the approximate kNN results to the query and those of the actual kNN results, to measure the quality of ANN results, which is widely used in the literatures [34, 106, 53, 83]. Given a query q, let $\{p_1, p_2, \ldots, p_k\}$ be the approximate kNN to q returned by an ANN method, and $\{o_1, o_2, \ldots, o_k\}$ be the true kNN. Then the ratio is defined as:

$$ratio = \frac{1}{k} \sum_{i=1}^{k} \frac{\|q, p_i\|}{\|q, o_i\|} \tag{5.21}$$

  Clearly, ratio close to 1.0 means the ANNS algorithm returns better results and vice versa. The average ratio of a set of queries is reported in the experiments.

- **Recall**. Recall is the ratio between the number of true kNNs found in the approximate kNN set and the value of k. It measures how many true kNNs can be found by ANNS methods.

- **I/O cost**. The page size b is set to be 4096 for all algorithms in the experiments. We assume the index and dataset reside in the external memory before the query is issued (i.e., cold startup). A unit I/O cost is a random I/O and we set the cost of a sequential I/O as 0.01 for the index accessing according to the profiling of our hardware system running the experiments. During the distance verification, we firstly sort the point IDs and then sequentially access the data points in the external memory. Thus, the cost of a sequential I/O for the verification is set to be 0.1 according to the profiling of our hardware system. The average I/O cost of a set of queries is used in the experiments.

- **Search time**. It is the wall clock time for running a query. We report the average search time among a set of test queries.
- **Preprocessing time**. We report the preprocessing time of the algorithms, including the training time (i.e., the learning of hashing functions) and index construction time (i.e., generate embeddings for every point and build index).
- **Index size**. We also report the size of the indexes generated by the algorithms.

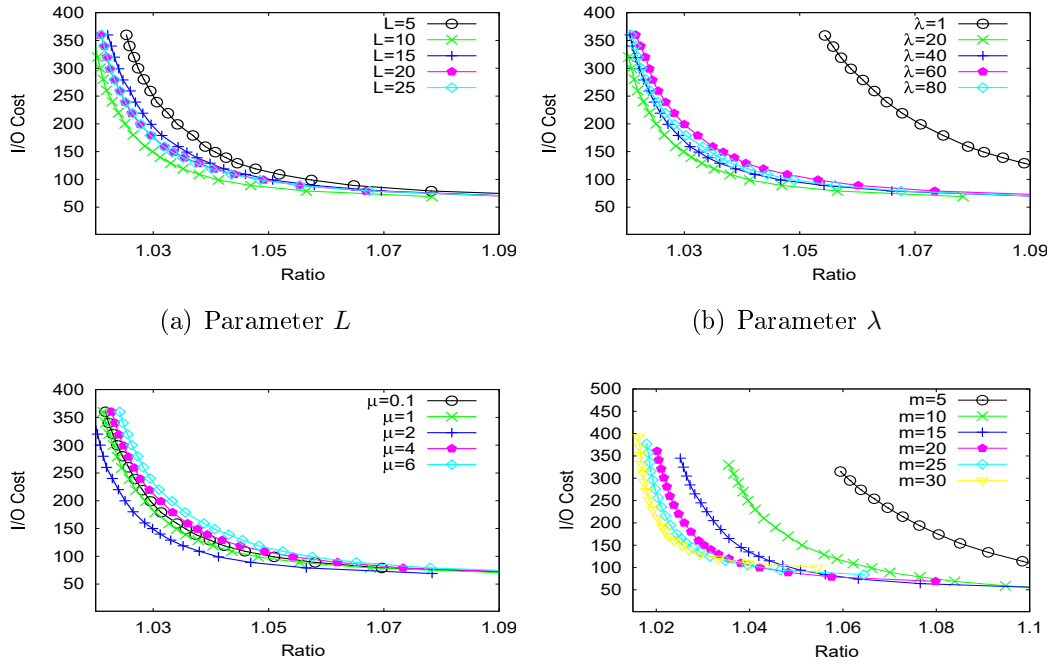
Parameter Setting By default, the k value of k-ANNS is set to be 20, which may vary from 10 to 100 in the experiments. The parameters of the algorithms are set to default values as suggested by the original authors unless otherwise specified. Particularly, for PQBF [83], the number of PQB-trees K' is set to be 64, the number of PQB-trees θ selected to perform ANNS is set to be 4 and the ratio ϵ is set to be 0.4. In AOSKNN algorithm [83], the dimensionality of PCA (m) is set to 6. The precision ε and the relaxation factor λ is set to 0.9 and 2, respectively. In I-LSH algorithm [81], the approximate ratio c is set to be 2 for *Tiny*, *Deep1b* and *Sift1B* datasets and 1.7 for other datasets for a good overall performance. The success possibility δ is set to be $1/2 - 1/e$. The setting of the candidate size T in our algorithm depends on the value k and the corresponding dataset, which can be tuned by users for satisfactory performance. Table 5.2 shows the possible values of the four parameters in OPFA and their default values (in bold fonts). The impact of these parameters will be evaluated in Subsection 5.6.2, where we also evaluate the impact of the number of lists m for NeOPFA.

Our two ANNS algorithms are implemented in standard C++ and the source codes of PQBF [83], AOSKNN [43], I-LSH [81] are provided by the original authors. The source code of M-tree [97] is from GitHub (`https://github.com/erdavila/M-Tree`). Note that we can only successfully setup the main-memory version of M-tree, and we use it in the performance evaluation. We count the number of nodes accessed during the querying as the I/O costs, where we set the node size of M-tree to be the page size (i.e., 4096 bytes). The algorithms are compiled with G++ with -O3 in Linux.

All experiments are performed on a machine with Intel Xeon Gold 2.7GHz CPU and Redhat Linux System, with 180G main memory.

5.6.2 Parameter Tuning

In this subsection, we investigate the impact of the parameters for our proposed methods, and decide the default settings by tuning parameters on *Deep* datasets. Note that we do not report the parameter tuning details of neural network in NeOPFA algorithm since it is beyond the focus of this paper.

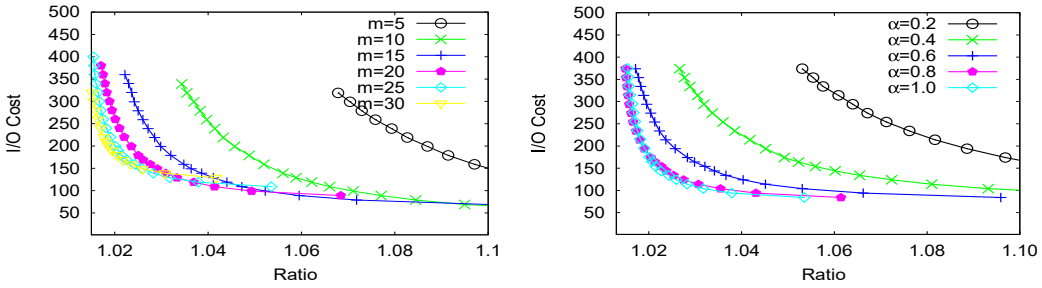


(a) Parameter L (b) Parameter λ

(c) Parameter μ (d) Parameter m

Figure 5.3: The impact of parameters of OPFA on Deep

We first investigate the four parameters of OPFA. The first three parameters are from the linear hash function, i.e., the number of buckets L, the regularization factor λ and factor μ. And the last one, i.e., the number of sorted lists m, is from our ANNS framework. For the NeOPFA, we only need to tune the number of lists m in the experiment given the setting of neural network. We tune the candidate size T to plot



(a) Parameter m (b) Parameter α

Figure 5.4: The impact of parameters of NeOPFA on Deep

the curve of I/O cost *w.r.t* ratio for each parameter. When tuning one parameter, the other parameters are set to be the default values in Table 5.2.

The impact of L on OPFA is plotted in Fig. 5.3(a) where the trade-offs between I/O cost and ratio are reported with L varying from 5 to 25. Smaller L leads to a larger mismatch, but less training time. In addition, larger L makes the loss function harder to be optimized. We observe that OPFA achieves a trade-off for $L = 10$, which is used in the following experiments. Recall that λ is orthogonality regularization and μ is the additional regularization that encourages the learned projection vectors to be unit vectors. The impacts of λ and μ are plotted in Fig. 5.3(b)-(c). As expected, the larger λ leads to a better ratio but incurs higher I/O cost, since the algorithm needs to access more pages to find out the first candidate that has been seen on all m sorted lists, and vice versa. Compared with λ, μ is smaller such that the optimization focuses more on the loss function and the orthogonality regularization. We observe that OPFA achieves a good overall performance when $\lambda = 20$ and $\mu = 2$.

For the impact of m for OPFA and NeOPFA, the results are plotted in Fig. 5.3(d) and Fig. 5.4(a). m has a significant influence on the trade-off between I/O costs and accuracy where a larger m leads to better accuracy, but higher I/O cost. It is reported that both OPFA and NeOPFA reach a good trade-off when $m = 25$, and lager m cannot distinctly achieve a better performance.

We also investigate the situation discussed in Section 5.5. That is, an object may become a candidate after $\lceil \alpha \cdot m \rceil$ hits, i.e., the object appears $\lceil \alpha \cdot m \rceil$ times during the

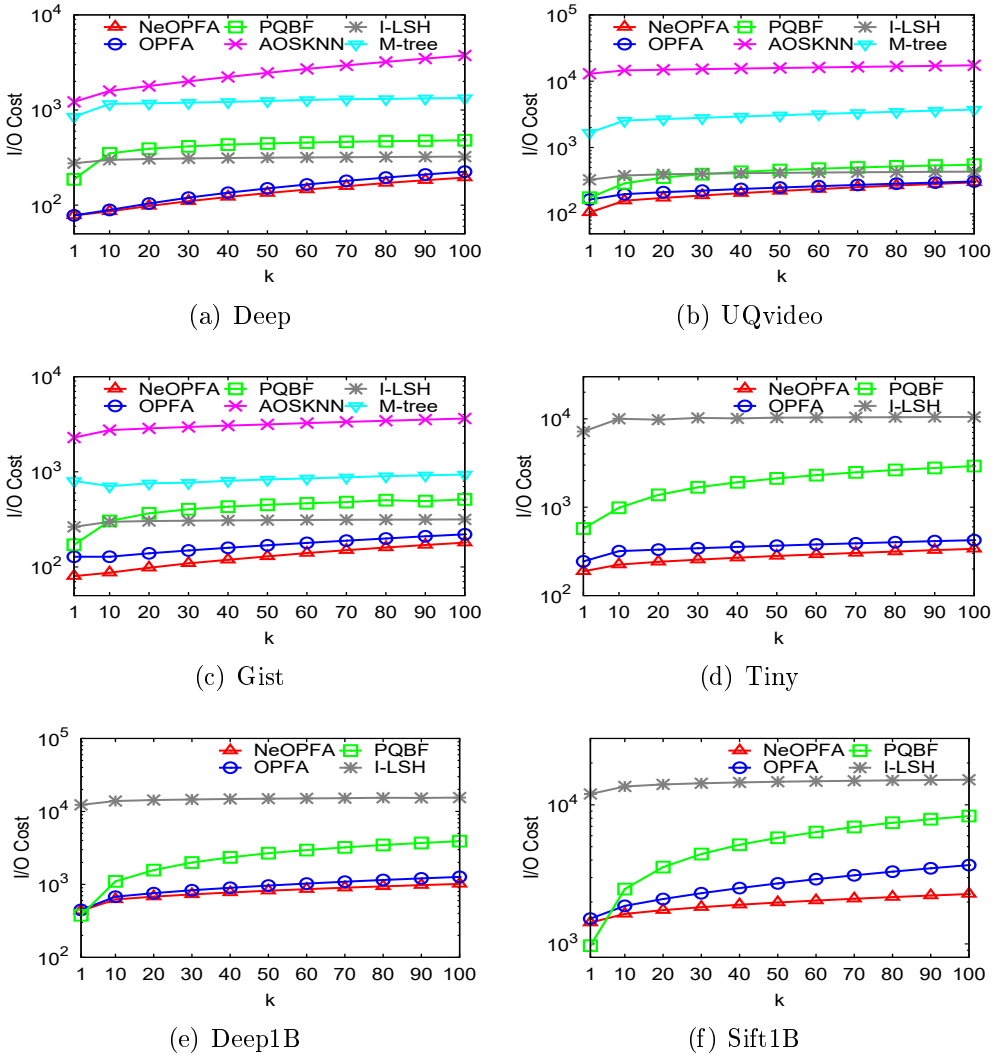
search with $0 < \alpha \leq 1$. In Fig. 5.4(b), we report the trade-offs between I/O costs and ratio for different α values given $m = 25$ on *Deep* dataset for NeOPFA. It is shown that NeOPFA achieves the best overall performance when $\alpha = 1$. This confirms the effectiveness of our search strategy in Algorithm 3.

5.6.3 Performance Comparison

In this subsection, we present comprehensive experimental results for the six algorithms on all datasets, in terms of the six evaluation metrics. To evaluate the performance of the algorithms on k-ANNS, k is set to be $\{1, 10, 20, \ldots, 100\}$. **Note** that the experimental results of AOSKNN and M-tree on *Tiny*, *Deep1B* and *Sift1B* datasets are not available because they failed to build up the indices under the current system settings.

I/O Cost. Fig. 5.5 reports the I/O costs of six algorithms on six datasets where k varies from 1 to 100. It is shown that OPFA and NeOPFA outperform the other four ANNS techniques by a large margin especially when $k \geq 10$. The IO cost of NeOPFA is around 68%-89.3% of OPFA on most of the datasets. Though they share the same ANNS framework, the non-linear hash functions learnt by neural networks are more powerful than the linear hash functions, with higher training cost (see Fig. 5.9). Note that OPFA shows good performance for $k = 1$ on some datasets, however, its I/O cost increases quickly when $k \geq 10$, which is larger than our methods. Due to the use of random I/O, the overall performance of PQBF, AOSKNN and M-tree are not competitive. It is interesting that PQBF outperforms AOSKNN and M-tree, which is because of the good performance of PQ method. M-tree outperforms AOSKNN on most of datasets. Although I-LSH can also take advantage of the sequential I/O, the poor quality of the random hash leads to a larger number of lists to be accessed for a decent search accuracy. Thus, the I/O cost of I-LSH is larger than our proposed algorithms, especially on the three very large datasets.

Ratio. We evaluate the average ratio of k-ANN queries for all algorithms on six datasets by varying k. The experimental results are plotted in Fig. 5.6. In addition



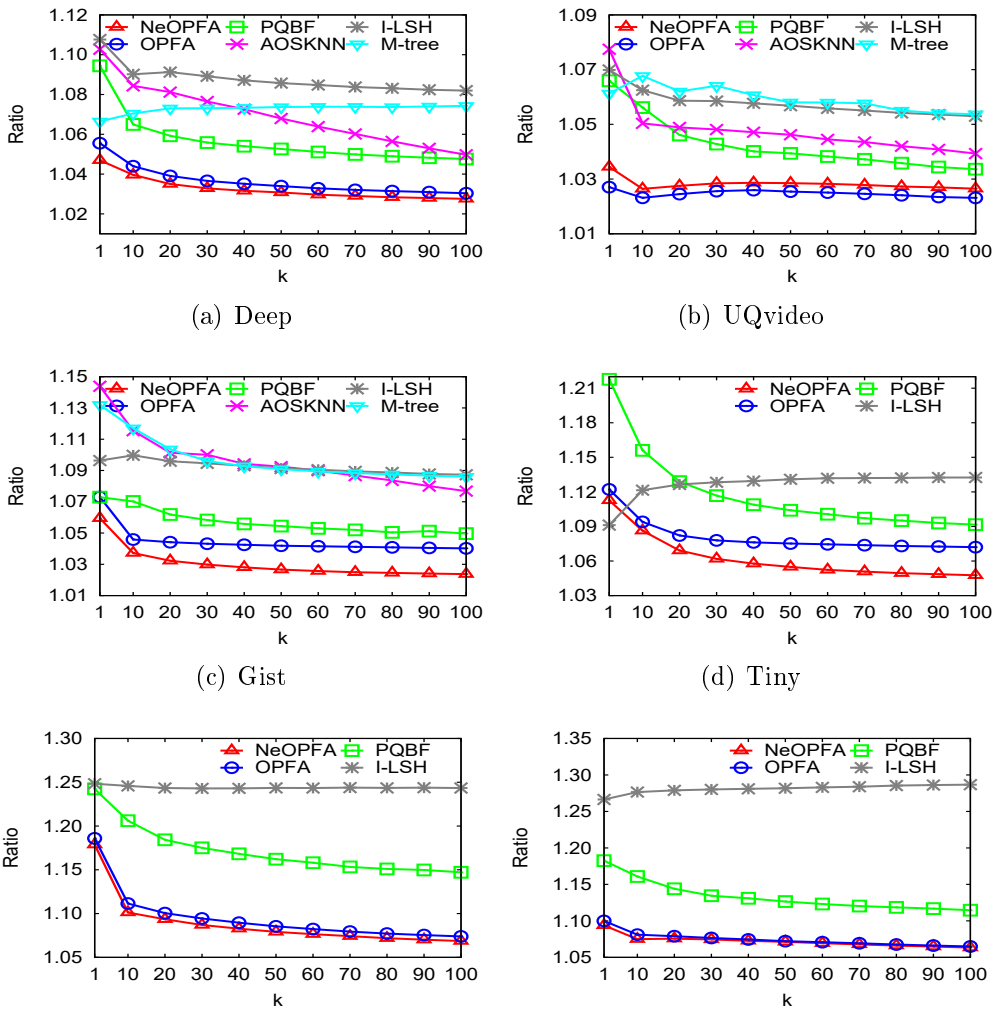
(a) Deep (b) UQvideo

(c) Gist (d) Tiny

(e) Deep1B (f) Sift1B

Figure 5.5: I/O Cost with respect to k on all datasets

to the superior performance in terms of I/O costs, We still observe that NeOPFA and OPFA also outperform the other four algorithms on the accuracy of the search results, thanks to the high quality data-sensitive hashing functions and the sequential I/O accesses. The large gap can be observed on *Deep1B* and *Sift1B* datasets. As expected, NeOPFA performs better than OPFA, especially on *Gist* and *Tiny*. This is because the new loss function used in NeOPFA can better preserve order information in the high dimensional space and the neural network can learn sophisticated non-linear hash functions. Among six algorithms, I-LSH has the worst performance on most of datasets



(a) Deep (b) UQvideo

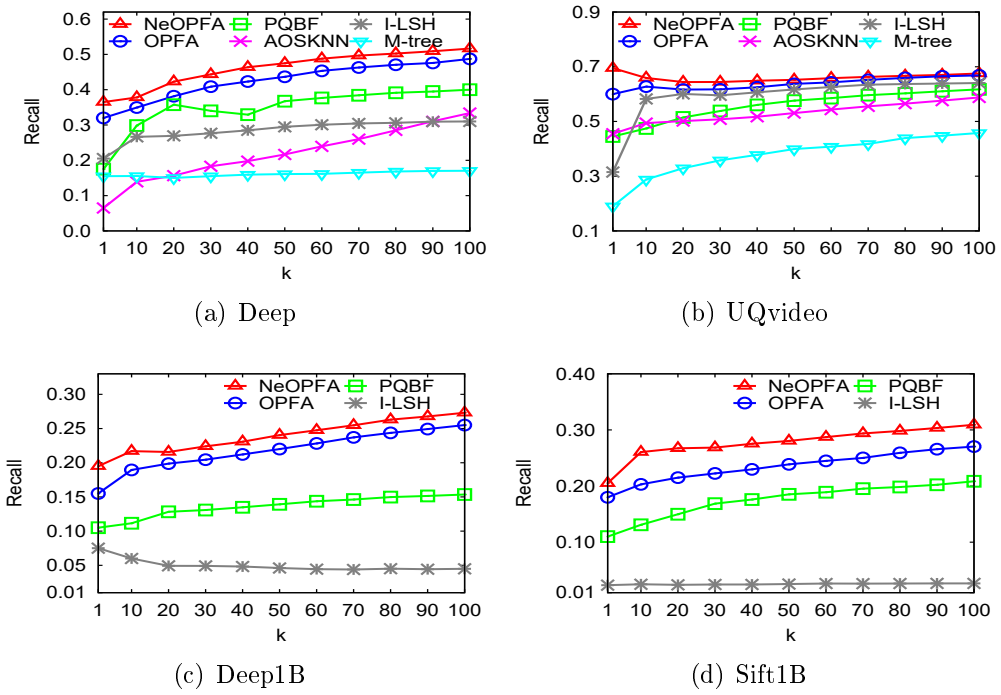
(c) Gist (d) Tiny

(e) Deep1B (f) Sift1B

Figure 5.6: Ratio with respect to k on all datasets

because of the use of data-independent random hash functions. Same as the I/O cost, PQBF demonstrates better performance compared to AOSKNN and M-tree.

Recall. The experimental results of the recall with respect to k for the six algorithms on four datasets are plotted in Fig. 5.7. Consistent with the observations in *Ratio*, NeOPFA and OPFA have the highest recall when compared with the other ANNS methods. For example, given k = 100, the recalls of NeOPFA and OPFA are 0.51 and 0.48, while the recalls of PQBF, I-LSH, AOSKNN and M-tree are 0.40, 0.31, 0.33 and 0.17, respectively, on *Deep*. NeOPFA has a higher recall than OPFA. PQBF performs



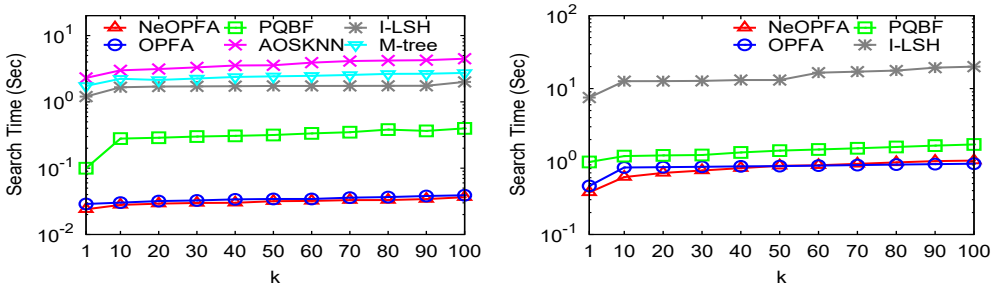
(a) Deep (b) UQvideo

(c) Deep1B (d) Sift1B

Figure 5.7: Recall with respect to k

better than AOSKNN, I-LSH and M-tree on most of datasets.

Search Time. Although the focus of this chapter is the external-memory algorithms, we also evaluate the running time of the algorithms. Due to the space limitation, we just report the results on two datasets, which can be seen in Fig. 5.8. It is shown that both NeOPFA and OPFA outperform the state-of-the-art algorithms due to their I/O efficiency. In terms of other four algorithms, PQBF is faster than I-LSH, AOSKNN and M-tree.



(a) Gist (b) Tiny

Figure 5.8: Search Time with respect to k

Datasets	Index Size (MB)					
	NeOPFA	OPFA	PQBF	AOSKNN	I-LSH	M-tree
Gist	102.5	98.4	84.6	144.2	849.7	21.6
Deep	102.8	100.1	70.4	148.7	864.6	20.7
UQvideo	306.9	304.2	210.6	443.9	2,662.4	63.6
Tiny	8,092.5	8,089.6	5,836.8	-	39,014.4	-
Deep1B	102,402.4	102,400.0	75,673.6	-	491,929.6	-
Sift1B	102,402.4	102,400.0	75,673.6	-	491,929.6	-

Table 5.3: Index Sizes of All Algorithms (in Megabytes)

Index Size. The index sizes of six methods on six datasets are reported in Table 5.3. We observe that the index size of M-tree is the smallest on *Gist*, *Deep* and *UQvideo*, since it just needs to store the object IDs and some distance information for pruning. The index size of PQBF is the second smallest, followed by our methods. The index size of OPFA is 67.3% – 68.5% of the index size by AOSKNN, and 11.4% – 20.8% of that by I-LSH. The index size of NeOPFA is slightly larger than OPFA because of the parameters of neural network kept.

Preprocessing Time. We report the preprocessing time in Fig. 5.9. Preprocessing time considers learning the mapping functions, generating the embeddings, and the index construction for NeOPFA and OPFA. I-LSH does not need to learn the hash functions and the generation of random hash functions is very efficient. Thus, it has the best performance in terms of preprocessing time. PQBF ranks the second due to the efficiency of PQ code generation. As expected, NeOPFA spends more time on preprocessing than OPFA as the the neural network learning takes substantial amount of time.

5.6.4 Summary

Based on the experimental results, we have the following observations:

- Among the six algorithms, I-LSH is the only algorithm with theoretical guarantee. It also enjoys the efficient sequential I/O accesses and the worst case performance theoretical guarantee. However, the overall performance of I-LSH is far from

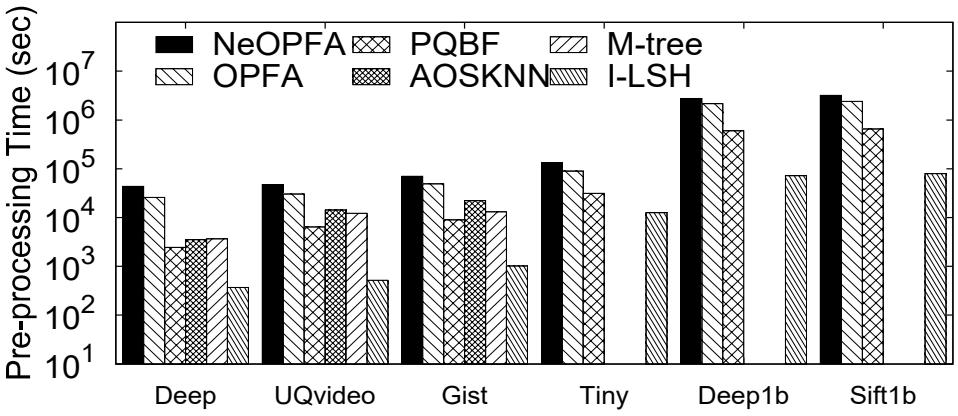


Figure 5.9: Pre-processing Time on All Datasets

satisfactory on six real-life datasets because of the use of random hash functions. M-tree is a general indexing technique for metric space which can support ANN search. But, as demonstrated in experiments, its performance is not competitive due to the random I/Os performed on the index.

- By utilizing the existing learning to hash approaches (i.e., PQ and PCA), the hash values of PQBF and AOSKNN are data-sensitive. However, the I/O efficiency is not considered in the learning of their hash functions. Moreover, random I/Os are invoked in the search of the index. As reported in the experiments, PQBF and AOSKNN are outperformed by our proposed methods in terms of I/O cost, accuracy and search time.

- Our proposed leaning to hash techniques directly optimize the hash functions against the index (i.e., the lists). Thus the I/O efficiency is considered by the objective functions of the machine learning tasks. Moreover, sequential I/Os are invoked during the search. These make two proposed algorithms OPFA and NeOPFA achieve a good trade-off between I/O cost (search time) and accuracy. As expected, the non-linear hash functions learnt from sophisticated neural network can enhance the performance, at the cost of more training time.

5.7 Conclusion

In this chapter, we develop two I/O efficient indexing and query processing methods to achieve highly efficient I/O performance for approximate nearest neighbor search (ANNS) in high dimensional space. Our methods are based on a framework that uses sequential I/Os for finding candidates for a query based on indexes learned from the data. We consider both linear and non-linear functions to construct the learned index with novel objective functions. Comprehensive experiments on six high-dimensional benchmarking datasets with objects up to 1 billion, show that our proposed methods outperform the state-of-the-art I/O-focused ANNS techniques in terms of I/O efficiency, search accuracy.

Chapter 6

Approximate Nearest Neighbor Search: An Experimental Evaluation

6.1 Overview

In this chapter, we conduct a comprehensive experimental evaluation for the state-of-the-art approximate nearest neighbour search (ANNS) algorithms. This work is published in [75] and the rest of this chapter is organized as follows. Section 6.2 gives out the scope of this evaluation by imposing some constraints. Section 6.3 presents the detailed descriptions for the 19 representative state-of-the-art ANNS algorithms that we evaluate in this chapter. Section 6.4 introduces our new proposed graph-based approximate method, DPG. The comprehensive and systematic experimental evaluation is conducted in Section 6.5. Further analyses for algorithms are presented in Section 6.6. Finally, we summarize this chapter in Section 6.7.

6.2 Evaluation Scope

The problem of ANNS on high dimensional data has been extensively studied in various literatures such as databases, theory, computer vision, and machine learning. Over hundreds of algorithms have been proposed to solve the problem from different perspectives, and this line of research remains very active in the above domains due to its importance and the huge challenges. To make a comprehensive yet focused comparison of ANNS algorithms, in this chapter, we restrict the scope of this evaluation by imposing the following constraints.

1. Representative and competitive ANNS algorithms. We consider the state-of-the-art algorithms in several domains, and omit other algorithms that have been dominated by them unless there are strong evidence against the previous findings.

2. No hardware specific optimizations. Not all the implementations we obtained or implemented have the same level of sophistication in utilizing the hardware specific features to speed up the query processing. We pay more attention on the query technology itself, and weaken the implementation skills. Therefore, we modified several implementations so that no algorithm uses multiple threads, multiple CPUs, SIMD instructions, hardware pre-fetching, or GPUs.

3. Dense vectors. We treat the input data as dense vectors, taking no account of the specific processing for sparse data.

4. Support the Euclidian distance. The Euclidean distance is one of the most widely used measure on high-dimensional datasets. It is also supported by most of the ANNS algorithms.

5. Exact k-NN as the ground truth. In several existing works, each data point has a label (typically in classification or clustering applications) and the labels are regarded as the ground truth when evaluating the recall of approximate k-NN algorithms. In this chapter, we use the exact k-NN points as the ground truth as this works for all datasets and majority of the applications.

Prior Benchmark Studies. There are two recent ANNS benchmark studies: [92]

and ann-benchmark [15]. The former considers a large number of other distance measure in addition to the Euclidean distance, and the latter does not disable general implementation tricks. In both cases, their studies are less comprehensive than ours, e.g., with respect to the number of algorithms and datasets evaluated.

6.3 The State-of-the-art ANNS Algorithms

In this section, we present the key technical details of the 19 state-of-the-art ANNS algorithms that involve in our experimental evaluation. According to the literature review in Chapter 2, we divide these approximate algorithms into four categories: *LSH-based*, *L2H-based*, *Partition-based* and *Graph-based*.

6.3.1 LSH-based methods

Locality sensitive hashing is widely studied in the past due to its theoretical guarantee and ease implementation. In this category, we evaluate three recent LSH-based methods: SRS [106], QALSH [53] and FALCONN [4]. The details of these three algorithms are presented below.

The idea of SRS [106] is to project high dimensional data points into low dimensional representations, which are then organized by a multi-dimensional index that supports the incremental k-NN search. The 2-stable random projection is used during the mapping stage. In the query processing step, SRS first perform an incremental exact k-NN query on the constructed index, among which each retrieved candidate will experience the early-termination test. Finally, the candidate with smallest distance to query is returned.

In traditional LSH functions, the buckets are partitioned without considering the query, which would cause that the points closer to query may be partitioned into different buckets. To address this issue, QALSH [53] proposes the query-aware LSH functions. In the pre-processing step, QALSH performs the 2-stable random projection for each data point, and then the B^+-tree is employed to organize the projections.

When a query arrives, the projection of query is used an "anchor" for bucket partition. Then the k-NN search is simulated as conducting a range query on the B^+-tree.

FALCONN [4] is an efficient and well-tested LSH-based ANN library, which is mainly designed for the cosine similarity. However, it can also be used for ANN search under Euclidean distance which is employed in this chapter. The basic idea is utilizing a multiprobe scheme to speed up the cross-polytope LSH [111] while maintaining the good theoretical guarantee as Spherical LSH [6]. The hash families in FALCONN are implemented with multi-probe LSH in order to minimize the memory cost.

6.3.2 L2H-based methods

Learning to hash methods aim at learning hash functions from data distribution such that the similarity relationship between data points can be kept in the hash coding space. The representative methods in this category include Anchor Graph Hashing (AGH) [82], Scalable Graph Hashing (SGH) [59], Neighbor Sensitive Hashing (NSH) [98], Neighborhood APProximation index (NAPP) [92], Selective Hashing (SH) [37], Optimal Product Quantization (OPQ) [38] and Composite Quantization (CQ) [127].

One critical drawback of the existing unsupervised hashing methods is that they usually need to specify a global distance measure, which can not capture local low-dimensional manifold structure existing in many real-world data. To handle this case, AGH [82] adopts a neighborhood graph-based hashing strategy to automatically capture the neighborhood structure underlying in the data to learn appropriate compact codes in an unsupervised manner.

AGH and some graph-based hashing methods directly exploit the neighborhood structure to guide the hashing learning procedure. However, they need to compute the pairwise similarities between any two data points, which limits their application for large-scale datasets. Instead, SGH [59] employs a feature transformation approach to effectively approximate the whole neighborhood graph without explicitly computing the similarity graph matrix. Thus, the $O(N^2)$ computation overhead and large memory cost are avoided in SGH.

The goal of hashing-based methods is that the original distance relationship among data points can be preserved in the Hamming space (i.e., hash coding space) after hashing projection. To achieve this, existing hashing techniques tend to place the separators uniformly, while this may make the points close to query be assigned with different hash codes. Instead, the NSH [98] changes the shape of the projection functions which impose a larger slope when the original distance between a pair of objects is small, allowing the Hamming distance (i.e., projection distance) to remain stable beyond a certain distance. In other words, if the distance between query q and point p is smaller than a threshold, the above property will apply to q. To handle different possible queries, NSH chooses multiple pivots and limits the maximal average distances between a pivot and its closest neighbor pivot to ensure that there will be at least one nearby pivot for any new query.

Permutation methods assess the similarity among objects based on their relative distances to the pre-specified reference points, rather than the real distance values directly. The motivation of this kind of methods is that the nearest neighbours usually have similar pivot rankings to that of the query. The basic idea of permutation methods is to randomly choose m pivots from the data points and then for each point p, the pivots are sorted in the ascending order of their distances from p. Such that a permutation of the pivots is essentially a low-dimensional embedding for a data point. In the querying stage, those points that have the permutations sufficiently close to that of query will be retrieved, and finally the distance verification is conducted. Neighborhood APProximation index (NAPP) [92] is one of the most efficient implementation for Permutation-based methods, which relies on inverted index for searching.

Original LSH aims at finding points falling into a fixed radius of the query point, i.e., radius search. For the k-NN search, the radii required for different queries may vary by orders of magnitude, which depends on the density of the region around the query. Thus, it is hard for global hashing functions used by many learning-based methods to capture the diverse local patterns that are crucial for k-NN search. Selective Hashing (SH) [37] is especially designed for k-NN search problem. The key idea is to create

multiple LSH indices with different granularities (i.e., radii). Then, each data point is only stored in one selected index, with certain granularity that is especially effective for k-NN searches near it. Data points in dense regions are stored in the index with small granularity, while data points in sparse regions are stored in the index with large granularity. In the querying step, the algorithm will push down the query and examine the cell with suitable granularity.

Optimized Product Quantization (OPQ) [38] is an extension of product quantization (PQ) [57], which optimizes the index by minimizing the quantization distortion with respect to the space decompositions and the quantization codewords. Composite Quantization (CQ) [127] is another representative learning-based algorithm, which is the generalization of PQ. Similar to PQ, the idea of CQ is to approximate a vector using the composition of several elements selected from several dictionaries and to represent this vector by a short code composed of the indices of the selected elements.

6.3.3 Partition-based Algorithms

Partition-based methods have been widely studied to answer the nearest neighbour search problem. In this category, we evaluate two representative *hyperplane* partitioning algorithms, Annoy [14] and FLANN [91]. In addition, we also evaluate a classical *pivoting* partitioning method, Vantage-Point tree [18] (VP-tree), in the experiments.

FLANN is an automatic NNS algorithm configuration method which selects the most suitable algorithm from randomized kd-tree [103], hierarchical k-means tree [33], and linear scan approaches for a particular data set. Differ form traditional kd-tree, the *randomized kd-tree* in FLANN utilizes a hyperplane of the selected splitting dimension to partition the data points, among which the splitting dimension is chosen from the dimensions that have the largest variance on input data. The search step is a depth-first search algorithm prioritized by some heuristic scoring functions. For the *hierarchical k-means tree* in FLANN, the key idea is to partition the data points at each level into K regions using K-means clustering method. Then the same approach is applied recursively to the data points in each region. FLANN selects one of the three algorithms

by minimizing a cost function which considers the indexing time, searching time and memory cost to determine the suitable algorithm.

Annoy [14] is an empirically engineered algorithm that has been used in the recommendation engine in spotify.com. The basic idea of Annoy is to construct multiple hierarchical 2-means trees where each tree is independently constructed by recursively partitioning the data points. The querying process is carried out by traversing trees from the roots to the leaves closest to query with the help of a priority queue.

Different from the hyperplane partitioning methods, VP-tree [18] recursively divides data points based a randomly chosen pivot π. For each partition, a mean value r of the distances from π to the data points of this partition is achieved. Then the pivot-centered ball with radius r is used to divide the space: the inner objects are allocated to the left subtree, while the outer objects are allocated to the right subtree. In querying stage, the triangle inequality can be easily used to prune undesirable partitions, and the k-NN search is simulated as a range search with a decreasing radius.

6.3.4 Graph-based Algorithms

The key idea of graph-based methods is to build a proximity graph as an index and then the search can be easily performed by greedily traversing the graph. This kind of methods enjoy the popularity of nearest neighbour search community due to its prominent performance on the ANNS problem. In this category, we evaluate four representative algorithms, including KGraph [28], Small World (SW) [87], Hierarchical Navigable Small World (HNSW) [89] and Rank Cover Tree (RCT) [50].

KGraph is basically a K-NN graph where there are K out-going edges for each node (i.e., data point), pointing to its K nearest data points. The computation of exact K-NN graph is costly, KGraph tends to build the approximate one. The construction relies on a simple principle: *A neighbor's neighbor is probably also a neighbor*. During the indexing procedure, the neighbors' neighbors, including K-NN and reverse K-NN points, are used to improve neighbourhood of each data point. Some optimization techniques are used to accelerate the graph construction. The query searching could

be efficiently executed by iteratively expanding neighbors' neighbors in a best-first search strategy following the edges.

A small world (SW) [87] method is essentially a variant of a navigable small world graph. Previous K-NN graph preserves the K nearest neighbors of each node, while SW keeps two kinds of connections for each edge. The construction of small world is a bottom-top procedure that inserts all the data points consecutively. Specifically, for each new incoming point p, the nearest neighbors of p from the current graph are first retrieved, which are then connected to p. As more and more objects are inserted into the graph, links that previously served as short-range links now become long-range links making a navigable small world. The querying process is the greedy search with multi-restarts.

Hierarchical navigable small world (HNSW) [89] is an extension of small world. HNSW incrementally builds a multi-layer structure that consists of hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. The produced structure still has the property of navigable small world graph while making the links separated by their characteristic distance scales. The searching algorithm is an iterative greedy search starting from the top layer and finishing at the zero layer.

Rank cover tree (RCT) [50] is a hierarchical tree structure, which is similar to SASH [52] and Cover Tree [16]. However, it entirely avoids to use the numerical constraints such as triangle inequality for pruning. The index of RCT is constructed by inserting the nodes from high levels to low levels, during which the nodes in level i are randomly chosen from that of level $i-1$ with a certain probability. The searching algorithm starts from the root of the tree, and iteratively visits the nodes most similar to query at each level until reaching the bottom level.

6.4 Diversified Proximity Graph

The experience and insights we gained from this study enable us to engineer a new method, Diversified Proximity Graph (DPG), which constructs a different proximity

graph to achieve better and more robust search performance.

6.4.1 Motivation

A great number of papers employed the heuristic - "for each node, the closer Voronoi neighbors are more likely to be the neighbors" in the construction of their proximity graph. The heuristic requires the angle between edges must be at least 60°. However, it is a very weak constraint and can not guarantee the selected edges are sufficient for ANN search.

The construction of K-NN graph mainly consider the distances of neighbors for each data point, but intuitively we should also consider the *coverage* of the neighbors. As shown in Fig. 6.1, the two closest neighbors of the point p are a_3 and a_4, and hence in the 2-NN graph p cannot lead the search to the NN of q (i.e., the node b) although it is close to b. Since $a_1, \ldots, a_4$ are clustered, it is not cost-effective to retain both a_3 and a_4 in the K-NN list of p. This motivates us to consider the direction diversity (i.e., angular dissimilarity) of the K-NN list of p in addition to the distance, leading to the diversified K-NN graph. Regarding the example, including a_3 and b is a better choice for the K-NN list of p.

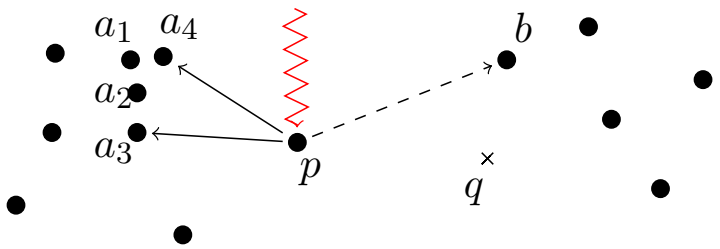


Figure 6.1: Motivation of Diversified Proximity Graph

Now assume we have replaced edge (p, a_4) with the edge (p, b) (i.e., the dashed line in Fig. 6.1), but there is still another problem. As we can see that there is no incoming edge for p because it is relatively far from two clusters of points (i.e., p is not 2-NN of these data points). This implies that p is isolated, and two clusters are disconnected in

the example. This is not uncommon in high dimensional data due to the phenomena of "hubness"[99] where a large portion of data points rarely serve as K-NN of other data points, and thus have no or only a few incoming edges in the K-NN graph. This motivates us to also use the reverse edges in the diversified K-NN graph; that is, we keep an bidirected diversified K-NN graph as the index, and we name it *Diversified Proximity Graph* (DPG).

6.4.2 Diversified Proximity Graph

The construction of DPG is a diversification of an existing K-NN graph, followed by adding reverse edges.

Given a reference data point p, the dissimilarity of two points x and y in p's K-NN list $\mathcal{L}$ is defined as the angle of $\measuredangle xpy$, denoted by $\theta(x, y)$. We aim to choose a subset of κ data points, denoted by $\mathcal{S}$, from $\mathcal{L}$ so that the average angle between two points in $\mathcal{S}$ is maximized; or equivalently, $\mathcal{S} = \arg\max_{\mathcal{S}\subseteq\mathcal{N},|\mathcal{S}|=\kappa} \sum_{o_i,o_j\in\mathcal{S}} \theta(o_i, o_j)$.

The above problem is NP-hard [72]. Hence, we design a simple greedy heuristic. Initially, $\mathcal{S}$ is set to the closest point of p in $\mathcal{L}$. In each of the following $\kappa - 1$ iterations, a point is moved from $\mathcal{L} \setminus \mathcal{S}$ to $\mathcal{S}$ so that the average pairwise angular similarity of the points in $\mathcal{S}$ is minimized. Then for each data point u in $\mathcal{S}$, we include both edges (p, u) and (u, p) in the diversified proximity graph. The time complexity of the diversification process is $O(\kappa^2 KN)$ where N is the number of data points, and there are totally at most $2\kappa N$ edges in the diversified proximity graph.

It is critical to find a proper K value for a desired κ in the diversified proximity graph as we need to find a good trade-off between diversity and proximity. In our empirical study, the DPG algorithm usually achieves the best performance when $K = 2\kappa$. Thus, we set $K = 2\kappa$ for the diversified proximity graph construction. This greedy algorithm has the time complexity of $O(\kappa^2 KN)$. We actually implemented a simplified version whose complexity is $O(K^2 N)$, which has only slightly worse performance than the full greedy version, but significantly fewer diversification time. Note that the search process of the DPG is the same as that of KGraph.

6.5 Experiments

In this section, we present a comprehensive experimental evaluation and analysis for the 19 state-of-the-art approximate NNS techniques.

6.5.1 Experimental Settings

Compared Algorithms

We test 19 existing representative ANNS algorithms (Section 6.3) from four categories and our proposed diversified proximity graph (DPG) method (Section 6.4). All of the modified source codes used in this experiment are public available on GitHub [107].
(1) LSH-based Methods. We evaluate Query-aware LSH [53] (**QALSH**[1], PVLDB'15), SRS [106] (**SRS**[2], PVLDB'14) and FALCONN [4] (**FALCONN**[3],NIPS'15) in this class.
(2) L2H-based Methods. We evaluate Scalable Graph Hashing [59] (**SGH**[4], IJCAI'15), Anchor Graph Hashing [82] (**AGH**[5], ICML'11) and Neighbor-Sensitive Hashing [98] (**NSH**[6], PVLDB'15) from binary-encoded learning to hash methods. In order to do non-exhaustive search, We organize the hash codes with the hierarchical clustering tree[91].

We also evaluate selective Hashing [37] (**SH**[7], KDD'15), Neighborhood APProximation index [92] (**NAPP**[8] PVLDB'15), Optimal Product Quantization [38] (**OPQ**[9], TPAMI'14), and Composite Quantization [127] (**CQ**[10], ICML'11). Note that we use the inverted multi-indexing technique [11] [11] to perform non-exhaustive search for **OPQ**.

[1] http://ss.sysu.edu.cn/~fjl/qalsh/qalsh_1.1.2.tar.gz
[2] https://github.com/DBWangGroupUNSW/SRS
[3] https://github.com/FALCONN-LIB/FALCONN
[4] http://cs.nju.edu.cn/lwj
[5] http://www.ee.columbia.edu/ln/dvmm/downloads
[6] https://github.com/pyongjoo/nsh
[7] http://www.comp.nus.edu.sg/~dsh/index.html
[8] https://github.com/searchivarius/nmslib
[9] http://research.microsoft.com/en-us/um/people/kahe
[10] https://github.com/hellozting/CompositeQuantization
[11] http://arbabenko.github.io/MultiIndex/index.html

For **AGH**, **NSH** and **OPQ**, we use 10% samples as training set.

(3) Partition-based Methods. We evaluate **FLANN**[12][91] (TPAMI'14), **FLANN-HKM**, **FLANN-KD**, **Annoy**[13] and an advanced Vantage-Point tree [18] (**VP-tree**[8], NIPS'13) in this class.

(4) Graph-based Methods. We evaluate Small World Graph [87] (**SW**[8], IS'14), Hierarchical Navigable Small World [89](**HNSW**[8], TPAMI'18), K-NN graph [28, 27] (**KGraph**[14], WWW'11), Rank Cover Tree [50] (**RCT**[15], TPAMI'15), and our Diversified Proximity Graph (**DPG**[15]).

Considering of the comparison fairness, we would like to focus on the algorithm perspective of the existing methods. we turned off all hardware-specific optimizations in the implementations of the methods. Specifically, we disabled distance computation using SIMD and multi-threading in KGraph, `-ffast-math` compiler option in Annoy, multi-threading in FLANN, and distance computation using SIMD, multi-threading, prefetching technique implemented in the NonMetricSpaceLib, i.e., SW, NAPP, VP-tree and HNSW). In addition, we disabled the optimized search implementation used in HNSW.

Computing Environment. All C++ source codes are complied by g++ 4.7, and MATLAB source codes (only for index construction of some learning to hash algorithms) are compiled by MATLAB 8.5. All experiments are conducted on a Linux server with Intel Xeon 8 core CPU at 2.9GHz, and 32G memory.

Datasets and Query Workload

We deploy 18 real datasets used by existing works which cover a wide range of applications including image, audio, video and text. We also use two synthetic datasets. Table 6.1 summarizes the characteristics of the datasets including the number of data points, dimensionality, *Relative Contrast* (RC [46]), *local intrinsic dimension-*

[12] `http://www.cs.ubc.ca/research/flann`
[13] `https://github.com/spotify/annoy`
[14] `https://github.com/aaalgo/kgraph`
[15] `https://github.com/DBWangGroupUNSW/nns_benchmark`

ality (LID [2]), and data type. RC indicates the ratio of the mean distance and NN distance for the data points, and smaller RC value implies harder dataset. LID calculates the local intrinsic dimensionality and a higher LID value implies harder dataset. We mark the first four datasets in Table 6.1 with asterisks to indicate that they are "hard" datasets compared with others according to their RC and LID values.

Datasets	N ($\times 10^3$)	d	**RC**	**LID**	**Data Type**
Nus*	269	500	1.67	24.5	Image
Gist*	983	960	1.94	18.9	Image
Rand*	1,000	100	3.05	58.7	Synthetic
Glove*	1,192	100	1.82	20.0	Text
Cifa	50	512	1.97	9.0	Image
Audio	53	192	2.97	5.6	Audio
Mnist	69	784	2.38	6.5	Image
Sun	79	512	1.94	9.9	Image
Enron	95	1,369	6.39	11.7	Text
Trevi	100	4,096	2.95	9.2	Image
Notre	333	128	3.22	9.0	Image
Yout	346	1,770	2.29	12.6	Video
Msong	922	420	3.81	9.5	Audio
Sift	994	128	3.50	9.3	Image
Deep	1,000	128	1.96	12.1	Image
Ben	1,098	128	1.96	8.3	Image
Gauss	2,000	512	3.36	19.6	Synthetic
Imag	2,340	150	2.54	11.6	Image
UQ-V	3,038	256	8.39	7.2	Video
BANN	10,000	128	2.60	10.3	Image

Table 6.1: Dataset Summary

Query Workload. Following the convention, we randomly remove 200 data points as the query points for each dataset. The average performance of the k-NN searches is reported. In this chapter, k is equal to 20 by default.

6.5.2 Evaluation Measures

For each algorithm, we retrieve T points based on its searching process and then rerank these candidates using original features. The search quality is measured using

recall, precision, accuracy and mAP. The **recall** is the ratio of the true nearest items in the retrieved T items to k. The **precision** is defined as the ratio of the number of retrieved true items to T. F-score (F1 score) is the harmonic mean of precision and recall: $F1 = 2 * precision * recall/(precision + recall)$. Then **mean average precision(mAP)** is computed as the mean of average precisions over all the queries. Accuracy is equal to $\sum_{i=0}^{i=k} \frac{dist(q,kANN(q)[i])}{dist(q,kNN(q)[i])}$, where q is a query, $kNN(q)[i]$ is q's i-th true nearest neighbor, and $kANN(q)[i]$ is i-th nearest neighbor estimated by one ANNS algorithm for q.

The search efficiency is usually measured as the time taken to return the search results for a query. For most of the algorithms (expect for Graph-based methods), we could vary the number of the retrieved points T to get different pair of recall/precision/accuracy and its corresponding search time. Since exact k-NN can be found by a brute-force linear scan algorithm, we use its query time as the baseline and define the **speedup** as $\frac{\bar{t}}{t'}$, where $\bar{t}$ is query time for linear scan and t' is the search time at a specific recall or T. For instance, we come up with a speedup 10 if an algorithm takes 1 second while the linear scan takes 10 seconds.

In addition to evaluating the search performance, we also evaluate other aspects such as index construction time, index size, index memory cost and scalability.

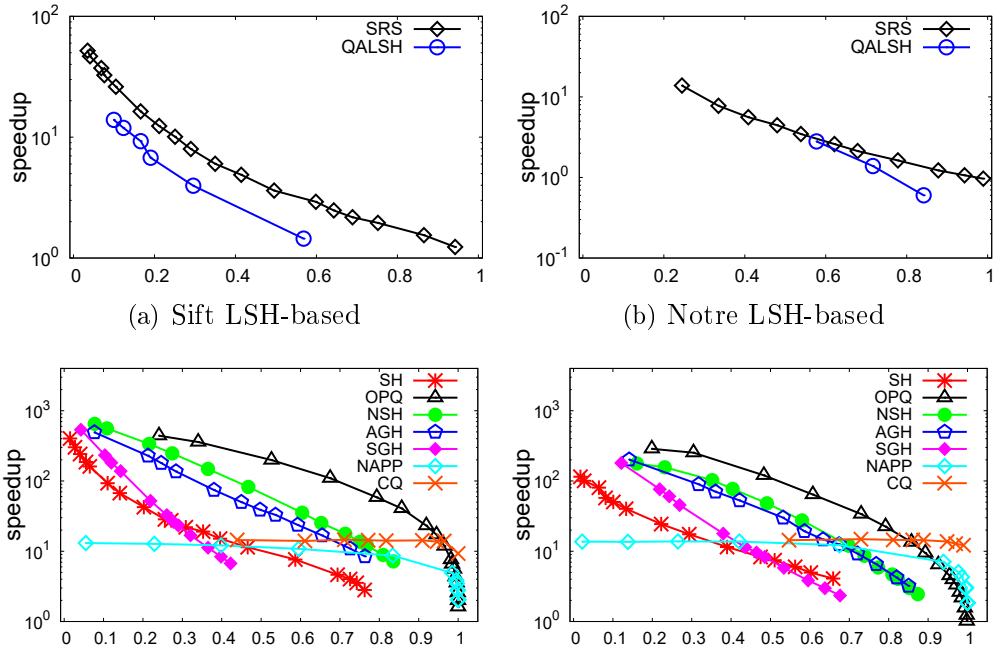
6.5.3 Comparison with Each Category

In this subsection, we evaluate the trade-offs between speedup and recall of all the algorithms on Sift and Notre data in each category. Given the large number of algorithms in the hashing-based category, we evaluate them in LSH-based and learning to hash-based subcategories separately. The goal of this round of evaluation is to select several algorithms from each category as the representatives in the second round evaluation (Subsection 6.5.4).

LSH-based Methods

Fig. 6.2(a) and (b) plot the trade-offs between the speedup and recall of two most recent data-independent algorithms SRS and QALSH on Sift and Notre. Note that, as FALCONN doesn't provide theoretical guarantee on L2 distance, we evaluate it in the second round.

As both algorithms are originally external memory based approaches, we evaluate the speedup by means of the total number of pages of the dataset divided by the number of pages accessed during the search. It shows that SRS consistently outperforms QALSH, and the similar trend is observed on other datasets. Thus, SRS is chosen as the representative in the second round evaluation where a cover-tree based in-memory implementation will be used.



(a) Sift LSH-based (b) Notre LSH-based

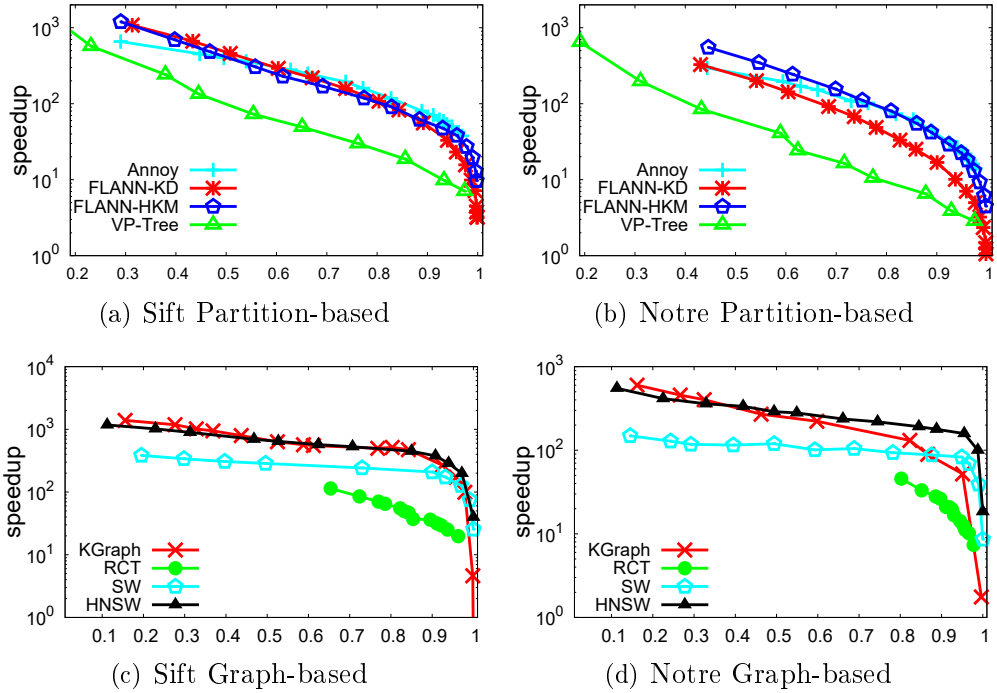
(c) Sift L2H-based (d) Notre L2H-based

Figure 6.2: Speedup vs Recall for LSH-based and L2H-based Methods

Learning to Hash-based Methods

We evaluate seven learning to hash based algorithms including OPQ, NAPP, SGH, AGH, NSH , SH and CQ. Fig. 6.2(c) and (d) demonstrate that, of all methods, the search performance of OPQ beats other algorithms by a big margin. Due to the linear scan search employed in the CQ, it shows a poor performance in terms of speedup@recall. In fact, CQ is very competitive in recall@T, but the indexing time of CQ is extremely high. So it is hard to apply in practice.

For most of datasets, Selective Hashing has the largest index size because it requires multiple long hash tables to achieve high recall. The index time value of OPQ has a strong association with the length of the sub-codeword and dimension of the data point. Nevertheless, the index construction time of OPQ still turns out to be very competitive compared with other algorithms in the second round evaluation. Therefore, we choose OPQ as the representative of the learning to hash based methods.



(a) Sift Partition-based (b) Notre Partition-based

(c) Sift Graph-based (d) Notre Graph-based

Figure 6.3: Speedup vs Recall for Partition-based and Graph-based Methods

Partition-based Methods

We evaluate three algorithms in this category: FLANN, Annoy and VP-tree. To better illustrate the performance of FLANN, we report the performance of both randomized kd-tree and hierarchical k-means tree, namely FLANN-KD and FLANN-HKM, respectively. Note that among 20 datasets, the randomized kd-tree method (FLANN-KD) is chosen by FLANN in five datasets: Enron, Trevi, UQ-V, BANN and Gauss. The linear scan is used in the hardest dataset Rand, and the hierarchical k-means tree (FLANN-HKM) is employed in the remaining 14 datasets.

Fig. 6.3(a) and (b) show that Annoy and FLANN-HKM have good performance on both datasets. For all datasets, Annoy, FLANN-HKM and FLANN-KD can obtain the highest performance on different datasets.

As the search performance of VP-tree is not competitive compared to FLANN and Annoy under all settings, it is excluded from the next round evaluation.

Graph-based Methods

In the category of Graph-based methods, we evaluate four existing techniques: KGraph, SW, HNSW and RCT. Fig. 6.3(c) and (d) show that the search performance of KGraph, SW and HNSW substantially outperforms that of RCT on Sift and Notre. Because HNSW is an improvement version of SW and can achieve better performance on all datasets, we discard SW from the next round evaluation.

Although the construction time of KGraph and HNSW are relatively large, due to the outstanding search performance, we choose them as the representatives of the graph-based methods. Note that we delay the comparison of DPG to the second round.

6.5.4 Second Round Evaluation

In the second round evaluation, we conduct comprehensive experiments on *eight* representative algorithms: SRS, FALCONN, OPQ, FLANN, Annoy, HNSW, KGraph, and DPG.

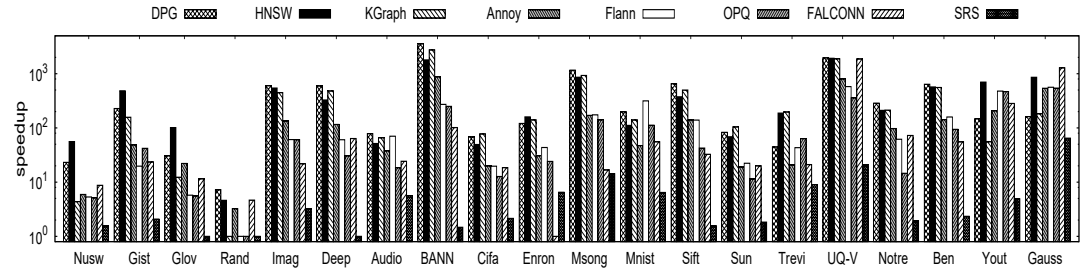


Figure 6.4: Speedup with Recall of 0.8

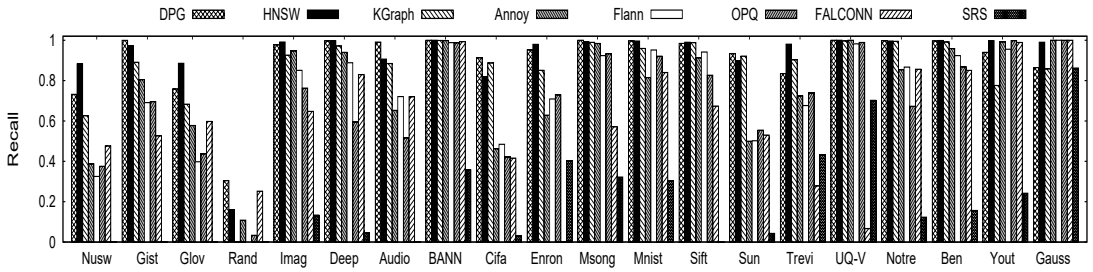
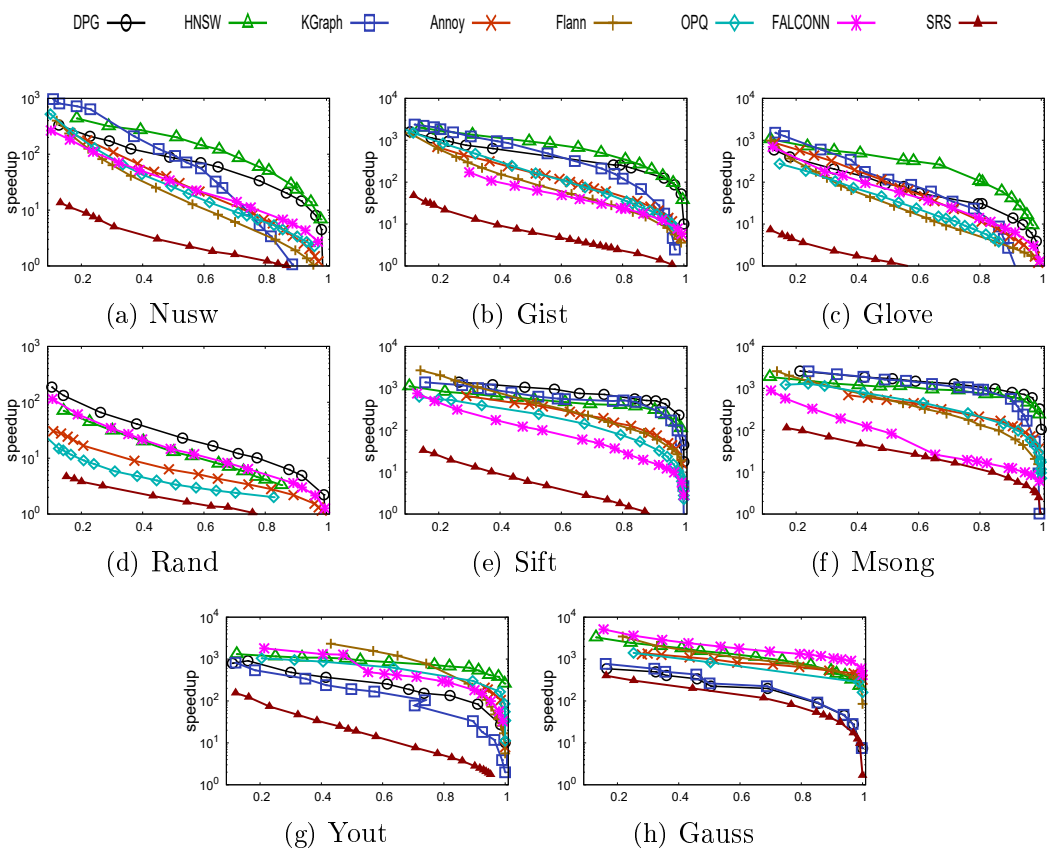


Figure 6.5: Recall with Speedup of 50

Comparison of Search Quality and Time

In the first set of experiments, Fig. 6.4 reports the speedup of eight algorithms when they reach the recall around 0.8 on 20 datasets. Note that the speedup is set to 1.0 if an algorithm cannot outperform the linear scan algorithm. Among eight algorithms, DPG and HNSW have the best overall search performance and KGraph follows. It is shown that DPG enhances the performance of KGraph, especially on hard datasets: Nusw, Gist, Glove and Rand. As reported thereafter, the improvement is also more significant on higher recall. For instance, DPG is ranked after KGraph on four datasets under this setting (recall 0.8), but it eventually surpasses KGraph on higher recall. Overall, DPG and HNSW have the best performance for different datasets. Not surprisingly, SRS is slower than other competitors with a huge margin as it does not exploit the data distribution. Similar observations are reported in Fig. 6.5, which depicts the recalls achieved by the algorithms with speedup around 50.

Fig. 6.6 illustrates speedup of the algorithms on eight datasets with recall varying from 0 to 1. It further demonstrates the superior search performance of DPG on high recall. The overall performance of HNSW, KGraph and Annoy are also very competitive,



(a) Nusw (b) Gist (c) Glove

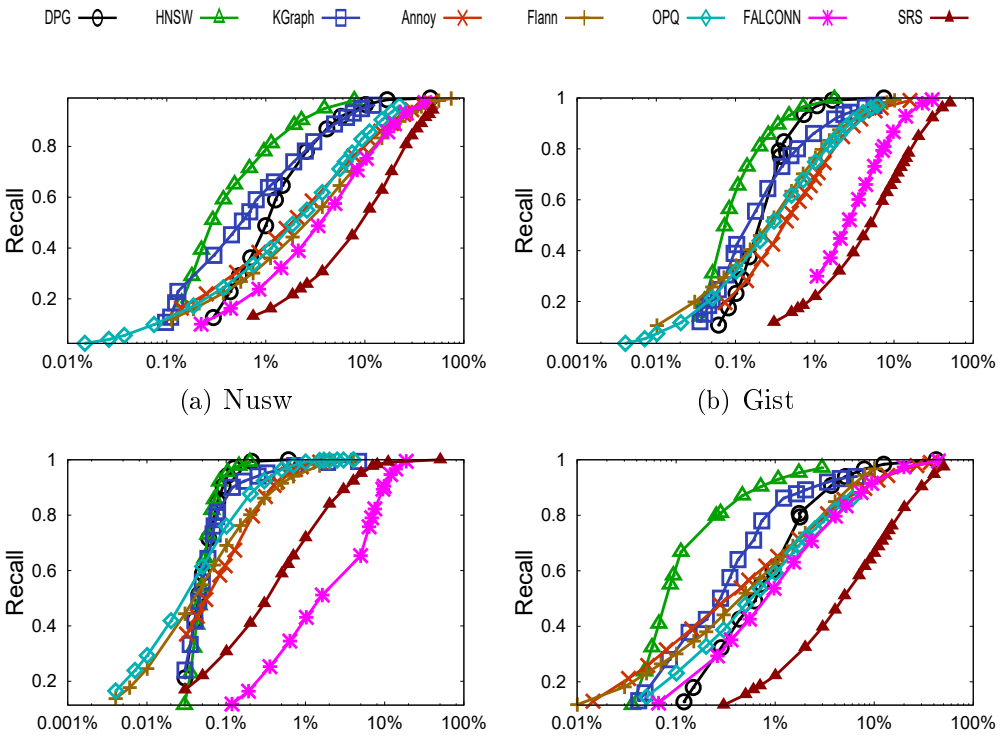
(d) Rand (e) Sift (f) Msong

(g) Yout (h) Gauss

Figure 6.6: Speedup vs Recall on Different Datasets

followed by FLANN. It is shown that the performance of both DPG and KGraph are ranked lower than that of HNSW, Annoy, FLANN and OPQ in Fig. 6.6(h) where the data points are clustered. As thereafter discussed in Section 6.6, Annoy, FLANN and OPQ essentially use the variants of k-means approach, and hence can well handle the clustered data. HNSW uses a heuristic to increase the probability of building the links between the clusters. FALCONN significantly outperforms SRS on all datasets, and it also surpasses the tree-based methods and learning to hash methods on some hard datsets, such as Glove, Nusw and random. It is worth noting that FALCONN even outperforms graph-based methods for Gauss dataset.

In Fig. 6.7, we evaluate the recalls of the algorithms against the percentage of data points accessed. As the search of most Graph-based methods starts from random en-



(a) Nusw (b) Gist

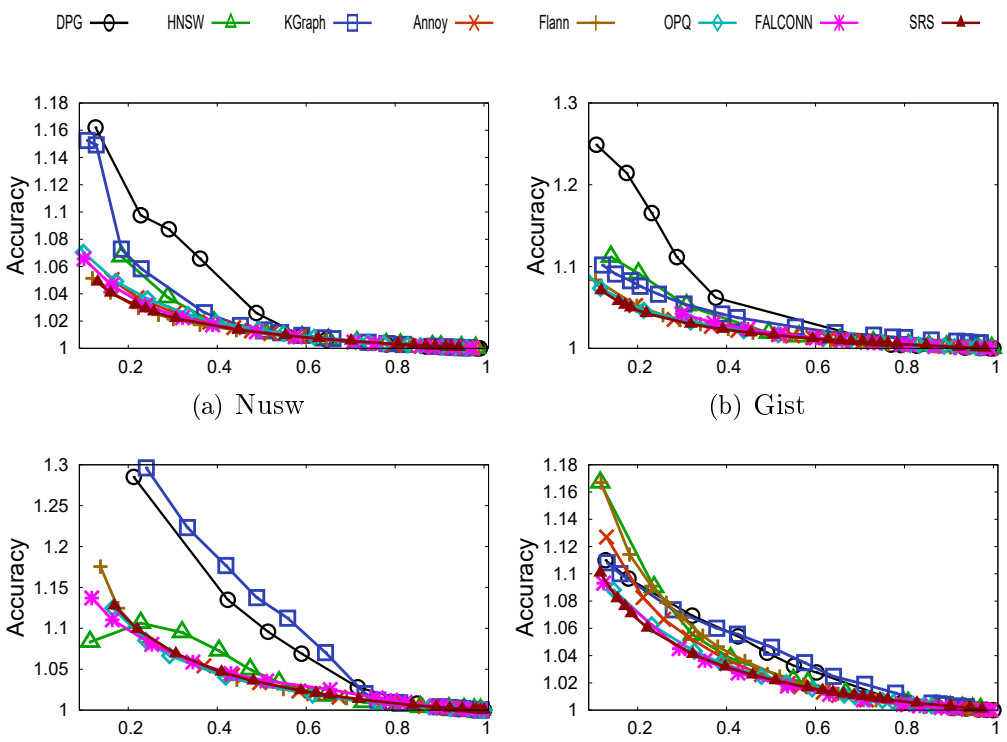
(c) Msong (d) Glove

Figure 6.7: Recall vs Percentage of Data Points Accessed

trance points and then gradually approaches the results while other algorithms initiate their search from the most promising candidates, the search quality of Graph-based methods is outperformed by Annoy, FLANN and even OPQ at the beginning stage. While, benefiting from HNSW's hierarchical structure, it could continue the search from the element which is the local optimum in the previous layer. The entries of each layer are carefully selected to ensure it could locate the closer points of the query quickly.

Fig. 6.8 shows the range search quality for a specific recall. Smaller accuracy indicates the closer of the results to the query, so the search quality is better. SRS and FALCONN which are designed for c-ANN search outperform all other algorithms.

In Figs. 6.12 and 6.13, we evaluate the precision@recall and F1@recall for each algorithm and report the mAP in Table 6.2. The results further verify our observations in Fig. 6.7. Tree-based methods and HNSW could find the closer neighbors after



(a) Nusw (b) Gist

(c) Msong (d) Glove

Figure 6.8: Accuracy vs Recall

retrieving small points.

Comparison of Indexing Cost

In addition to search performance, we also evaluate the index size, memory cost and construction time. Fig. 6.9 reports ratio of the index size (exclude the data points) and the data size. Except Annoy, the index sizes of all algorithms are smaller than the corresponding data sizes. The index sizes of DPG, KGraph, HNSW, SRS and FALCONN are irrelevant to the dimensionality because a fixed number of neighbor IDs and projections are kept for each data point. Consequently, they have a relatively small ratio on data with high dimensionality (e.g., Trevi). Overall, OPQ and SRS have the smallest index sizes, less than 5% among most of the datasets, followed by FALCONN, HNSW, DPG, KGraph and FLANN. It is shown that the rank of the index size of FLANN varies dramatically over 20 datasets because it may choose three possible index structures. Annoy needs to maintain a considerable number of trees for a good search quality, and

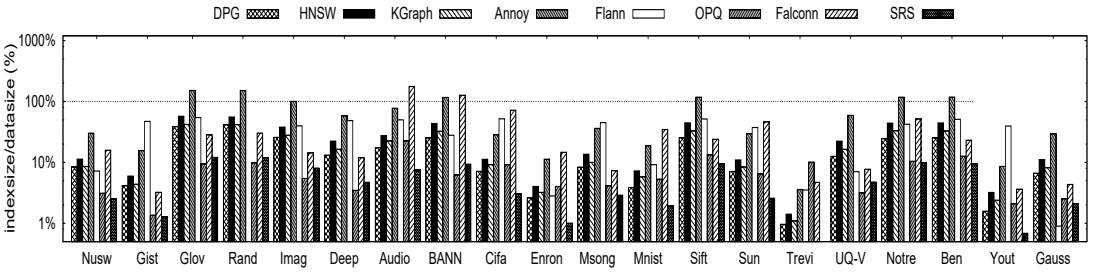


Figure 6.9: The Ratio of Index Size and Data Size (%)

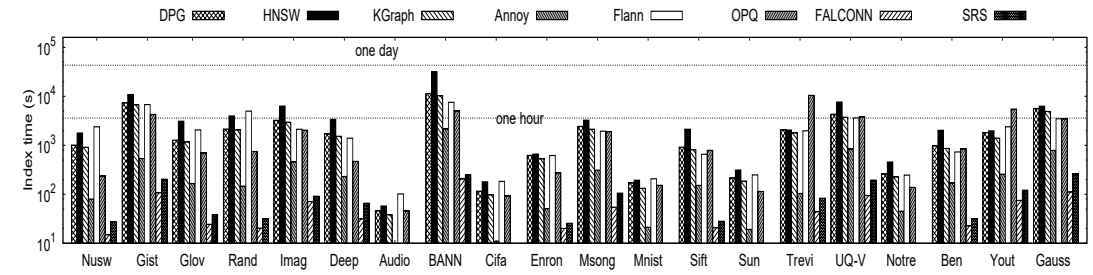


Figure 6.10: Index Construction Time (seconds)

hence has the largest index size.

Fig. 6.10 reports the index construction time on 20 datasets. FALCONN has the smallest index construction time among all the test algorithms. SRS ranks the second. The construction time of OPQ is related to the dimensionality because of the calculation of the sub-codewords (e.g., Trevi). HNSW, KGraph and DPG have similar construction time. Compared with KGraph, DPG does not spend large extra time for the graph diversification. Nevertheless, they can still build the indexes within one hour for 16 out of 20 datasets.

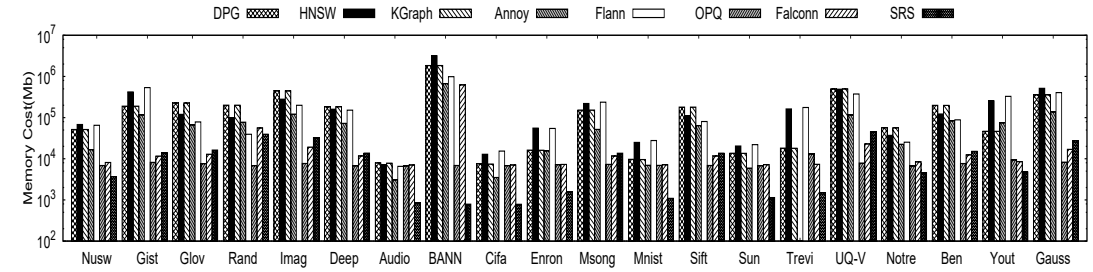
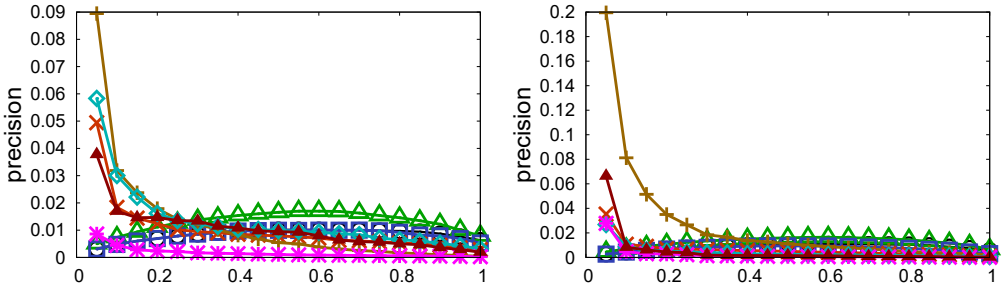
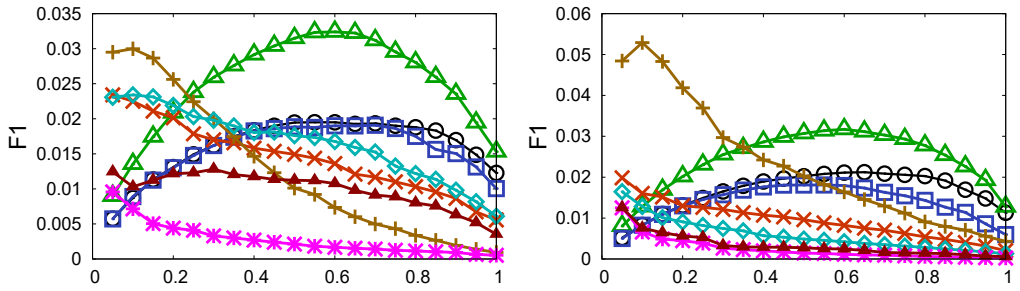


Figure 6.11: Index Memory Cost (MB)



(a) Nusw (b) Gist

Figure 6.12: Precision vs Recall



(a) Nusw (b) Gist

Figure 6.13: F1 score vs Recall

Fig. 6.11 reports the index memory cost of the algorithms on 20 datasets. OPQ needs less memory resource to build index, so it is very efficient for large-scale datasets.

6.5.5 Summary

Table 6.3 ranks the performance of the eight algorithms from various perspectives including search performance, index size, index construction time, index memory cost, and scalability. We also indicate that SRS is the only one with theoretical guarantee of searching quality, and it is very easy to tune the parameters for search quality and search time.

Below are some recommendations for users according to our comprehensive evaluations.

- When there are sufficient computing resources (both main memory and CPU) for

Algorithm	Nusw	Gist	Msong	Glove
DPG	0.008386	0.008586	0.01876	0.004906
HNSW	0.013122	0.012627	0.024439	0.012414
KGraph	0.008147	0.007161	0.017455	0.005867
Annoy	0.010082	0.006304	0.025659	0.019304
FLANN	0.012074	0.026154	0.069939	0.044106
OPQ	0.013121	0.004076	0.072857	0.003116
FALCONN	0.001661	0.002457	0.002444	0.002584
SRS	0.010264	0.005244	0.022778	0.002837

Table 6.2: mAP for each algorithm

the off-line index construction, and sufficient main memory to hold the resulting index, DPG and HNSW are the best choices for ANNS on high dimensional data due to their outstanding search performance in terms of robustness to the datasets, result quality, search time and search scalability.

- Except DPG and HNSW, we also recommend Annoy, due to its excellent search performance, construction cost, and robustness to the datasets. Moreover, it can provide a good trade-off between search performance and index size/construction time.

- If the construction time is a big concern, FALCONN would be a good choice because of its small construction time and good search performance.

- To deal with large scale datasets (e.g., 1 billion of data points) with moderate computing resources, OPQ and SRS are good candidates due to their small memory cost and construction time. It is worthwhile to mention that, SRS can easily handle the data points updates and have theoretical guarantee, which distinguish itself from other seven algorithms.

Category	Search Performance	Search Scalability		Theoretical Guarantee
		Datasize	Dim	
DPG	**1st**	**=1st**	6th	No
HNSW	**1st**	**=1st**	5th	No
KGraph	3rd	**=1st**	8th	No
Annoy	4th	6th	=3rd	No
FLANN	5th	**=1st**	7th	No
OPQ	6th	5th	=3rd	No
FALCONN	7th	8th	2nd	No
SRS	8th	7th	**1st**	**Yes**

Category	Index			Index Scalability	
	Size	Memory	Time	Datasize	Dim
DPG	4th	7th	7th	=5th	**=1st**
HNSW	7th	6th	8th	=7th	5th
KGraph	5th	7th	6th	=5th	**=1st**
Annoy	8th	4th	3rd	=7th	6th
FLANN	6th	5th	5th	4th	8th
OPQ	2nd	**1st**	4th	**1st**	=3th
FALCONN	3rd	2nd	**1st**	2nd	=3th
SRS	**1st**	3rd	2nd	3rd	7th

Table 6.3: Ranking of the Algorithms Under Different Criteria

6.6 Further Analyses

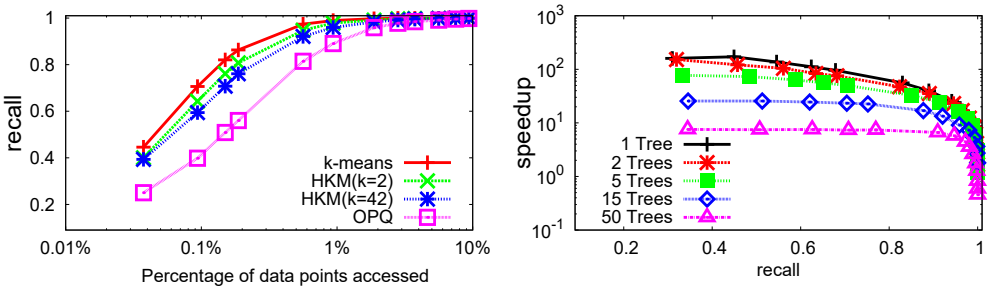
In this section, we analyze the most competitive algorithms in our evaluations, grouped by category, in order to understand their strength and weakness.

6.6.1 Space Partition-based Approach

Our comprehensive experiments show that Annoy and FLANN have the best performance among the space partition-based methods. Note that FLANN chooses FLANN-HKM in most of the datasets. In addition, OPQ divides the space by utilizing the production of k-means on M disjoint subspaces. Therefore, all three algorithms are based on k-means space partitioning. We define the algorithms who borrow the idea of k-means as **k-means-like** algorithms.

We identify that a key factor for the effectiveness of **k-means-like** algorithms is that the large number of clusters, typically $\Theta(N)$. Note that we cannot *directly* apply k-means with $k = \Theta(N)$ because (i) the index construction time complexity of k-means is linear to k, and (ii) the time complexity to identify the partition where the query is located takes $\Theta(N)$ time. Both OPQ and FLANN-HKM/Annoy achieve this goal indirectly by using the ideas of subspace partitioning and recursion, respectively.

We conduct experiments to understand which idea is more effective. We consider the goal of achieving **k-means-like** space partitioning with approximately the same number of non-empty partitions. Specifically, for Audio dataset, we consider the following choices: (i) Use OPQ with 2 subspaces and each has 256 clusters. The number of effective partitions k (i.e., non-empty partitions) is counted. Finally, $k = 18,611$. (ii) Use original FLANN-HKM to build a tree with roughly k leaf nodes. After carefully tuning the value of branching factor L, we found the number of total leaf nodes is 18000 when $L = 42$. (iii) Use FLANN-HKM with $L = 2$ and modify the stopping condition as the points in each leaf node is no larger than m. using this approach, the number of all leaf nodes k could be decided. After tuning the value of m, we get $k = 17898$ for $L = 2$. (iv) Use k-means directly with $k = 18,611$ to generate the partitions.



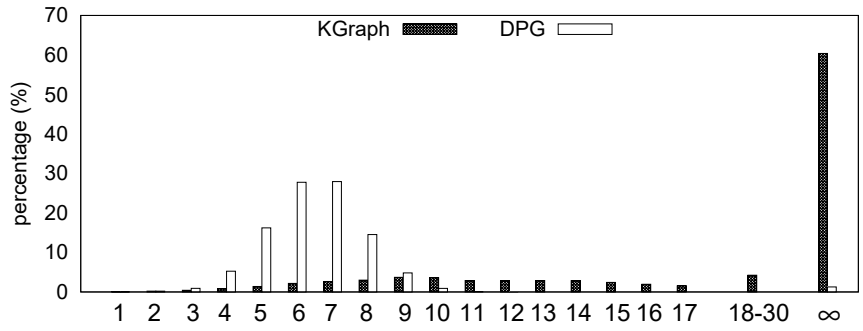
(a) Partition Quality (b) Multiple HKM Trees

Figure 6.14: Analyses of Space Partitioning-based Methods

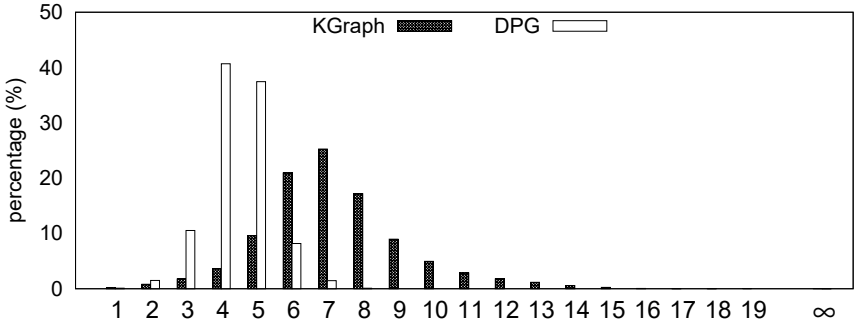
Fig. 6.14(a) reports the recalls of the above choices on Audio against the percentage of data points accessed. Partitions are accessed based on the ascending order of the distances of their centers to the query. We can see that OPQ-based partition has the worst performance, followed by (modified) FLANN-HKM with $L = 42$, and then $L = 2$. k-means has the best performance, although the performance differences between the latter three are not significant. Therefore, our analysis suggests that hierarchical k-means-based partitioning is the most promising direction so far.

Our second analysis is to investigate whether we can further boost the search performance by using multiple hierarchical k-means trees. Note that Annoy already uses multiple trees and it significantly outperforms a single hierarchical k-means tree in FLANN-HKM on most of the datasets. It is natural to try to enhance the performance of FLANN-HKM in a similar way.

We set up an experiment to construct multiple FLANN-HKM trees. In order to build different trees, we perform k-means clustering on a set of random samples of the input data points. Fig. 6.14(b) shows the resulting speedup vs recall where we use up to 50 trees. We can see that it is not cost-effective to apply multiple trees for FLANN-HKM on Audio, mainly because the trees obtained are still similar to each other, and hence the advantage of multiple trees cannot offset the extra indexing and searching overheads.



(a) Min # of hops to any 20-NNs (Yout)



(b) Min # of hops to any 20-NNs (Gist)

Figure 6.15: minHops Distributions of KGraph and DPG

6.6.2 Graph-based Approach

Our first analysis is to understand why KGraph, DPG and HNSW work very well (esp. attaining a very high recall) in most of the datasets. Our preliminary analysis indicates that this is because (i) the k-NN points of a query are typically *closely connected* in the proximity graph, and (ii) most points are *well connected* to at least one of the k-NN points of a query. (ii) means there is a high empirical probability that one of the p entry points selected randomly by the search algorithm can reach one of the k-NN points of the query, and (i) ensures that most of the k-NN points can be returned. By well connected, we mean there are many paths from an entry point to one of the k-NN point, hence there is a large probability that the "hills" on one of the path is low enough so that the search algorithm will not stuck in the local minima.

We also investigate why KGraph does not work well on some datasets and why DPG and HNSW works much better. KGraph does not work on Yout and Gauss mainly

because both datasets have many well-separated clusters. Hence, the index of KGraph has many disconnected components. Thus, unless one of the entrance points used by its search algorithm is located in the same cluster as the query results, there is no or little chance for KGraph to find any near point. On the other hand, mainly due to the diversification step and the use of the reverse edges in DPG, there are edges linking points from different clusters, hence resulting in much improved recalls. Similarly, in HNSW, the edges are also well linked.

For example, we conduct the experiment where we use the NN of the query as the entrance point of the search on Yout. KGraph then achieves 100 percent recall. In addition, we plot the distribution of the minimum number of hops (i.e., the length of the shortest path, denoted as minHops) between a data point and any of the k-NN points of a query for the indexes of KGraph and DPG on Yout and Gist in Fig. 6.15. We can observe that

- For KGraph, there are a large percentage of data points that cannot reach any k-NN points (i.e., those corresponding to ∞ hops) on Yout (60.38 percent), while the percentage is low on Gist (0.04 percent).
- The percentages of the ∞ hops are much lower for DPG (1.28 percent on Yout and 0.005 percent on Gist).
- There is no ∞ hops for HNSW on both datasets.
- DPG and HNSW have much more points with small minHops than KGraph, which contributes to making it easier to reach one of the k-NN points. Moreover, on Yout, HNSW has the most points with small minHops over three algorithms, which results in a better performance as shown in Fig. 6.6(g).

6.7 Conclusion

NNS is an fundamental problem with both significant theoretical values and empowering a diverse range of applications. It is widely believed that there is no practically

competitive algorithm to answer exact NN queries in sublinear time with linear sized index. A natural question is whether we can have an algorithm that *empirically* returns *most of* the k-NN points in a *robust* fashion by building an index of size $O(N)$ and by accessing at most αN data points, where α is a small constant (such as 1 percent).

In this chapter, we evaluate many state-of-the-art algorithms proposed in different research areas and by practitioners in a comprehensive manner. We analyze their performance and give practical recommendations.

Due to various constraints, the study in this chapter is inevitably limited. In our future work, we will (i) consider high dimensional sparse data; (ii) use more complete, including exhaustive method, to tune the algorithms; (iii) consider other distance metrics.

Finally, our understanding of high dimensional real data is still vastly inadequate. This is manifested in many heuristics with no reasonable justification, yet working very well in *real* datasets. We hope that this study opens up more questions that call for innovative solutions by the entire community.

Chapter 7

Epilogue

In this thesis, we investigate the problem of nearest neighbour search (NNS) in high dimensional space, which is a fundamental and significant technique in many applications. For the exact NNS problem, we propose a new embedding technique, combining linear and non-linear methods, to devise a tight distance lower bound. Following the *filter-and-verify* paradigm, we develop an efficient exact NNS algorithm using the new lower bounding technique for pruning. In terms of the approximate NNS problem, we propose a novel data-sensitive indexing and query processing framework with an emphasis on optimizing the I/O efficiency. The proposed index is learned by two learning to hash methods with novel objective functions. We conduct performance evaluations on many real-world datasets to show the efficiency and effectiveness of our proposed approaches. Finally, we conduct a comprehensive and systematic experimental study for the state-of-the-art approximate NNS methods, including 19 algorithms and 20 datasets. We present comprehensive experimental results and detailed analysis for the comparison methods. Motivated by the evaluation, we develop a new graph-based approximate NNS algorithm that can achieve high recall and search efficiency on majority of the datasets under a wide range of settings.

Even though hundreds of nearest neighbour search algorithms were proposed in the literature, with the development of Artificial Intelligence techniques like the deep learn-

ing and reinforcement learning, we still have the opportunity to push this problem to have a better performance. For example, a deep learning-based model can be designed to devise a better embedding for the original data that can maximally preserve the similarity information. In contrast to the traditional partition-based methods, machine learning-based methods can be used to learn a better partition from data.

REFERENCES

[1] A. Abdullah, A. Andoni, R. Kannan, and R. Krauthgamer. Spectral approaches to nearest neighbor search. In *FOCS*, pages 581–590, 2014.

[2] L. Amsaleg, O. Chelly, T. Furon, S. Girard, M. E. Houle, K. Kawarabayashi, and M. Nett. Estimating local intrinsic dimensionality. In *SIGKDD*, 2015.

[3] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS'2006*, pages 459–468.

[4] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal lsh for angular distance. In *Advances in neural information processing systems*, pages 1225–1233, 2015.

[5] A. Andoni, P. Indyk, H. L. Nguyen, and I. Razenshteyn. Beyond locality-sensitive hashing. In *SODA*, pages 1018–1028, 2014.

[6] A. Andoni and I. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *STOC*, 2015.

[7] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[8] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *CVPR*, pages 2055–2063, 2016.

[9] A. Babenko and V. S. Lempitsky. Additive quantization for extreme vector compression. In *CVPR'2014*, pages 931–938.

[10] A. Babenko and V. S. Lempitsky. Tree quantization for large-scale similarity search and classification. In *CVPR'2015*, pages 4240–4248.

[11] A. Babenko and V. S. Lempitsky. The inverted multi-index. In *CVPR*, pages 3069–3076, 2012.

[12] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the Acm*, 18(9):509–517, 1975.

[13] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the Acm*, 23(4):214–229, 1980.

[14] E. Bernhardsson. Annoy at github `https://github.com/spotify/annoy`, 2015.

[15] E. Bernhardsson. Benchmarking nearest neighbors `https://github.com/erikbern/ann-benchmarks`, 2016.

[16] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM, 2006.

[17] M. Boguñá, D. V. Krioukov, and K. C. Claffy. Navigability of complex networks. *Nature Physics*, abs/0709.0303(abs/0709.0303), 2008.

[18] L. Boytsov and B. Naidan. Learning to prune in metric and non-metric spaces. In *NIPS*, 2013.

[19] D. Cai, X. Gu, and C. Wang. A revisit on deep hashings for large-scale content based image retrieval. 2017.

[20] M. A. Carreira-Perpinan and R. Raziperchikolaei. Hashing with binary autoencoders. In *Computer Vision and Pattern Recognition*, pages 557–566, 2015.

[21] R. Caruana, N. Karampatziakis, and A. Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *ICML*, pages 96–103, 2008.

[22] L. Cayton. Fast nearest neighbor retrieval for bregman divergences. In *ICML'2008*, pages 112–119.

[23] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *STOC*, 2008.

[24] 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*, pages 253–262. ACM, 2004.

[25] T. Do, A. Doan, and N. Cheung. Discrete hashing with deep neural network. *CoRR*, 2015.

[26] T. Do, A. Doan, and N. Cheung. Learning to hash with binary deep neural network. In *ECCV'2016*, pages 219–234, 2016.

[27] W. Dong. Kgraph. `http://www.kgraph.org`, 2014.

[28] W. Dong, M. Charikar, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

[29] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD*, pages 301–312, 2003.

[30] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[31] X. Feng, J. Cui, Y. Liu, and H. Li. Effective optimizations of cluster-based nearest neighbor search in high-dimensional space. *Multimedia Systems*, 23(1):139–153, 2017.

[32] C. Fu and D. Cai. EFANNA : An extremely fast approximate nearest neighbor search algorithm based on knn graph. *CoRR*, abs/1609.07228, 2016.

[33] K. Fukunaga and P. M. Narendra. A branch and bound algorithms for computing k-nearest neighbors. *IEEE Trans. Computers*, 24(7):750–753, 1975. hierachical k-means tree.

[34] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 541–552. ACM, 2012.

[35] R. Gan. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition*, pages 1106–1113, 2012.

[36] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. DSH: data sensitive hashing for high-dimensional k-nnsearch. In *SIGMOD*, 2014.

[37] J. Gao, H. V. Jagadish, B. C. Ooi, and S. Wang. Selective hashing: Closing the gap between radius search and k-nn search. In *SIGKDD*, 2015.

[38] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(4):744–755, 2014.

[39] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[40] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[41] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(12):2916–2929, 2013.

[42] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383, 1998.

[43] Y. Gu, Y. Guo, Y. Song, X. Zhou, and G. Yu. Approximate order-sensitive k-nn queries over correlated high-dimensional data. *TKDE*, 30(11):2037–2050, 2018.

[44] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI*, pages 1312–1317, 2011.

[45] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[46] J. He, S. Kumar, and S. Chang. On the difficulty of nearest neighbor search. In *ICML*, 2012.

[47] J. He, W. Liu, and S. Chang. Scalable similarity search with optimized kernel hashing. In *SIGKDD*, pages 1129–1138, 2010.

[48] J. Heo, Z. Lin, X. Shen, J. Brandt, and S. Yoon. Shortlist selection with residual-aware distance estimator for k-nearest neighbor search. In *CVPR'2016*, pages 2009–2017.

[49] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[50] M. E. Houle and M. Nett. Rank-based similarity search: Reducing the dimensional dependence. *IEEE transactions on pattern analysis and machine intelligence*, 37(1):136–150, 2014.

[51] M. E. Houle and M. Nett. Rank-based similarity search: Reducing the dimensional dependence. *IEEE TPAMI*, 37(1):136–150, 2015.

[52] M. E. Houle and J. Sakuma. Fast approximate similarity search in extremely high-dimensional data sets. In *ICDE*, pages 619–630, 2005.

[53] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.

[54] Y. Hwang, B. Han, and H.-K. Ahn. A fast nearest neighbor search algorithm by nonlinear embedding. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3053–3060. IEEE, 2012.

[55] 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*, pages 604–613. ACM, 1998.

[56] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[57] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.

[58] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP'2011*, pages 861–864.

[59] Q. Jiang and W. Li. Scalable graph hashing with feature transformation. In *IJCAI*, pages 2248–2254, 2015.

[60] Z. Jin, Y. Hu, Y. Lin, D. Zhang, S. Lin, D. Cai, and X. Li. Complementary projection hashing. In *ICCV'2013*, pages 257–264, 2013.

[61] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE Trans. Cybernetics*, 44(11):2167–2177, 2014.

[62] Y. Jing and S. Baluja. Visualrank: Applying pagerank to large-scale image search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1877–1890, 2008.

[63] I. T. Jolliffe. Principal component analysis and factor analysis. *Principal component analysis*, pages 150–166, 2002.

[64] A. Joly and O. Buisson. Random maximum margin hashing. In *CVPR'2011*, pages 873–880.

[65] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *Proceedings of the 16th International Conference on Multimedia 2008*, pages 209–218, 2008.

[66] Y. Kalantidis and Y. S. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *CVPR'2014*, pages 2329–2336.

[67] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[68] N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[69] J. M. Kleinberg. Navigation in a small world. *Nature*, 406(6798):845, 2000.

[70] W. Kong and W. Li. Isotropic hashing. In *26th Annual Conference on Neural Information Processing Systems*, pages 1655–1663, 2012.

[71] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[72] C.-C. Kuo, F. Glover, and K. S. Dhir. Analyzing and modeling the maximum diversity problem by zero-one programming*. *Decision Sciences*, 24(6):1171–1185, 1993.

[73] M. Li, Y. Zhang, Y. Sun, W. Wang, I. W. Tsang, and X. Lin. An efficient exact nearest neighbor search by compounded embedding. In *International Conference on Database Systems for Advanced Applications*, pages 37–54. Springer, 2018.

[74] M. Li, Y. Zhang, Y. Sun, W. Wang, I. W. Tsang, and X. Lin. I/o efficient approximate nearest neighbour search based on learned functions. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 289–300. IEEE, 2020.

[75] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data - experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2020.

[76] Y.-C. Liaw, M.-L. Leou, and C.-M. Wu. Fast exact k nearest neighbors search using an orthogonal search tree. *Pattern Recognition*, 43(6):2351–2358, 2010.

[77] K. Lin, J. Lu, C. Chen, and J. Zhou. Learning compact binary descriptors with unsupervised deep neural networks. In *CVPR'2016*, pages 1183–1192, 2016.

[78] Y. Lin, R. Jin, D. Cai, S. Yan, and X. Li. Compressed hashing. In *CVPR'2013*, pages 446–451, 2013.

[79] V. E. Liong, J. Lu, G. Wang, P. Moulin, and J. Zhou. Deep hashing for compact binary codes learning. In *CVPR'2015*, pages 2475–2483, 2015.

[80] W. Liu, C. Mu, S. Kumar, and S. Chang. Discrete graph hashing. In *Annual Conference on Neural Information Processing Systems 2014*, pages 3419–3427.

[81] W. Liu, H. Wang, Y. Zhang, W. Wang, and L. Qin. I-LSH: I/O efficient c-approximate nearest neighbor search in high-dimensional space. In *ICDE*, pages 1670–1673, 2019.

[82] W. Liu, J. Wang, S. Kumar, and S. Chang. Hashing with graphs. In *ICML*, pages 1–8, 2011.

[83] Y. Liu, H. Cheng, and J. Cui. Pqbf: I/o-efficient approximate nearest neighbor search by product quantization. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 667–676. ACM, 2017.

[84] J. Lu, V. E. Liong, and J. Zhou. Deep hashing for scalable image search. *IEEE Trans. Image Processing*, 26(5):2352–2367, 2017.

[85] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[86] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Intelligent probing for locality sensitive hashing: multi-probe lsh and beyond. *Proceedings of the Vldb Endowment*, 10(12):2021–2024, 2017.

[87] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.*, 45:61–68, 2014.

[88] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, 2016.

[89] 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*, 2018.

[90] P.-G. Martinsson, V. Rokhlin, and M. Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, 30(1):47–68, 2011.

[91] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[92] B. Naidan, L. Boytsov, and E. Nyberg. Permutation search methods are efficient, yet faster search is possible. *PVLDB*, 8(12):1618–1629, 2015.

[93] G. Navarro. Searching in metric spaces by spatial approximation. *VLDB J.*, 11(1):28–46, 2002.

[94] J. Nocedal and S. Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[95] M. Norouzi and D. J. Fleet. Cartesian k-means. In *CVPR'2013*, pages 3017–3024.

[96] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, pages 1186–1195, 2006.

[97] C. Paolo, P. Marco, and Z. Pavel. M-tree: An efficient access method for similarity search in metric spaces. *PVLDB*, pages 426–435, 1997.

[98] Y. Park, M. J. Cafarella, and B. Mozafari. Neighbor-sensitive hashing. *PVLDB*, 9(3):144–155, 2015.

[99] M. Radovanovic, A. Nanopoulos, and M. Ivanovic. Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research*, 11:2487–2531, 2010.

[100] S. Ramaswamy and K. Rose. Adaptive cluster distance bounding for high-dimensional indexing. *IEEE Transactions on Knowledge and Data Engineering*, 23(6):815–830, 2011.

[101] R. Salakhutdinov and G. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.

[102] F. Shen, Y. Xu, L. Liu, Y. Yang, Z. Huang, and H. T. Shen. Unsupervised deep hashing with similarity-adaptive and discrete optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[103] C. Silpa-Anan and R. I. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, 2008.

[104] D. Song, W. Liu, R. Ji, D. A. Meyer, and J. R. Smith. Top rank supervised binary coding for visual search. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1922–1930, 2015.

[105] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[106] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *PVLDB*, 8(1):1–12, 2014.

[107] Y. Sun, W. Wang, Y. Zhang, and W. Li. Nearest neighbor search benchmark `https://github.com/DBWangGroupUNSW/nns_benchmark`, 2016.

[108] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576. ACM, 2009.

[109] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems (TODS)*, 35(3):20, 2010.

[110] K. Terasawa and Y. Tanaka. Spherical LSH for approximate nearest neighbor search on unit hypersphere. In *WADS'2007*, pages 27–38.

[111] K. Terasawa and Y. Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *Workshop on Algorithms and Data Structures*, pages 27–38. Springer, 2007.

[112] W. S. Torgerson. Multidimensional scaling: I. theory and method. *Psychometrika*, 17(4):401–419, 1952.

[113] H. Wang, J. Cao, L. Shu, and D. Rafiei. Locality sensitive hashing revisited: filling the gap between theory and algorithm analysis. In *CIKM*, pages 1969–1978, 2013.

[114] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD*, pages 839–848. ACM, 2018.

[115] J. Wang, W. Liu, S. Kumar, and S. Chang. Learning to hash for indexing big data - A survey. *Proceedings of the IEEE*, 104(1):34–57, 2016.

[116] J. Wang, W. Liu, A. X. Sun, and Y. Jiang. Learning hash codes with listwise supervision. In *ICCV'2013*, pages 3032–3039, 2013.

[117] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.

[118] J. Wang, J. Wang, N. Yu, and S. Li. Order preserving hashing for approximate nearest neighbor search. In *MM'2013*, pages 133–142.

[119] J. Wang, T. Zhang, J. Song, N. Sebe, and H. T. Shen. A survey on learning to hash. *CoRR*, 2016.

[120] Q. Wang, Z. Zhang, and L. Si. Ranking preserving hashing for fast similarity search. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[121] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.

[122] G. Wu, L. Liu, Y. Guo, G. Ding, J. Han, J. Shen, and L. Shao. Unsupervised deep video hashing with balanced rotation. In *Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3076–3082, 2017.

[123] Y. Xia, K. He, F. Wen, and J. Sun. Joint inverted indexing. In *ICCV*, pages 3416–3423, 2013.

[124] Z. Xia, X. Feng, J. Peng, and A. Hadid. Unsupervised deep hashing for large-scale visual search. In *International Conference on Image Processing Theory TOOLS and Applications*, pages 1–5, 2017.

[125] J. Yang, W. L. Zhao, C. H. Deng, H. Wang, and S. Moon. *Fast Nearest Neighbor Search Based on Approximate k-NN Graph.* 2018.

[126] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.

[127] T. Zhang, C. Du, and J. Wang. Composite quantization for approximate nearest neighbor search. In *ICML'2014*, pages 838–846.

[128] Y. M. Zhang, K. Huang, G. Geng, and C. L. Liu. Fast knn graph construction with locality sensitive hashing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 660–674, 2013.

[129] W. L. Zhao, J. Yang, and C. H. Deng. Scalable nearest neighbor search based on knn graph. 2017.

[130] Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *SIGMOD*, pages 2023–2037, 2016.