# Understanding Indexing Efficiency for Approximate Nearest Neighbor Search in High-dimensional Vector Databases

by

Yuting Qin

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN COMPUTATION AND COGNITION

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2024

© 2024 Yuting Qin. This work is licensed under a CC BY-NC-ND 4.0 license.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Yuting Qin
Department of Brain and Cognitive Sciences
May 7, 2024

Certified by: Xuhao Chen
Research Scientist, Thesis Supervisor

Certified by: Arvind
Professor in Electrical Engineering and Computer Science, Thesis Supervisor

Accepted by: Mehrdad Jazayeri
Director of Education
Department of Brain and Cognitive Sciences

# Understanding Indexing Efficiency for Approximate Nearest Neighbor Search in High-dimensional Vector Databases

by

Yuting Qin

Submitted to the Department of Brain and Cognitive Sciences
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN COMPUTATION AND COGNITION

**ABSTRACT**

Deep learning has transformed almost all types of data (e.g., images, videos, documents) into high-dimension vectors, which in turn forms *Vector Databases* as the data engines of various applications. As a result, queries on vector databases have become the cornerstone for many important online services, including search, eCommerce, and recommendation systems.

In a vector database, the major operation is to search the $k$ closest vectors to a given query vector, known as $k$-Nearest-Neighbor ($k$-NN) search. Due to massive data scale in practice, Approximate Nearest-Neighbor (ANN), which builds a search *index* offline to accelerate search online, is often used instead. One of the most promising ANN indexing approaches is the graph-based approach, which first constructs a proximity graph on the dataset, connecting pairs of vectors that are close to each other, then traverse the proximity graph for each query to find the closest vectors to a query vector. The search performance, in terms of the scope of traversal that leads to convergence, is highly dependent on the quality of the graph. There exist lots of prior work on improving the graph quality with various heuristics. However, no analysis or modeling work has been done to quatitatively evaluate the heuristics and their impact on the performance. Hence, it is unclear how to pick or combine the right heuristics to build a high-quality graph.

This thesis aims to establish this connection to fill the gap. The key challenge in quantifying the heuristics is the complex tradeoff between the search accuracy and search speed, which makes it almost impossible to establish an analytical model. To this end, we propose to leverage machine learning as the modeling tool. We first build an unified framework to characterize various graph building heuristics, by decoupling the graph construction and search phases. We then extract graph attributes (e.g., diameter), and collect ground-truth performance data (e.g., search speed and accuracy) within our framework, across multiple datasets and graph configurations. Based on the collected data, we train a linear regression model to predict the search performance. We show experimental results on our model performance, and also discuss the implications on selecting heuristics that improve the quality of the indexing graphs.

Thesis supervisor: Xuhao Chen
Title: Research Scientist

Thesis supervisor: Arvind
Title: Professor in Electrical Engineering and Computer Science

# Acknowledgments

I would like to express my deepest gratitude to Arvind for providing me with the opportunity to contribute to this project, which has played an important role in my learning. I am also profoundly thankful to my advisor, Xuhao, for his continuous support and invaluable guidance throughout this project. His assistance in framing research questions, designing experiments, and expanding my technical skills has been indispensable.

I am incredibly grateful to my mom for her unwavering support throughout my college journey, and to my dad, whose encouragement and regular check-ins have kept me motivated and focused.

Finally, I extend my thanks to everyone who has inspired and supported me throughout this process. Your belief in my work has made this achievement possible, and I am eager to apply the knowledge and insights gained in my future endeavors.

# Contents

**Title page** 1

**Abstract** 2

**Acknowledgments** 3

**1 Introduction** 8

**2 Background** 11
2.1 Vector Database and ANN Search . . . . . 11
2.2 ANN Indexing Methods . . . . . 13
2.2.1 Clustering-based vs. Graph-based Indexing . . . . . 14
2.2.2 Existing Graph Indexing Methods . . . . . 14
2.2.3 Parallelization in Building Graphs . . . . . 16
2.3 ANN Search Algorithms . . . . . 17
2.4 Machine Learning for ANN . . . . . 18

**3 ANN Analysis Framework** 19
3.1 Framework Overview . . . . . 19
3.2 Decoupling of Graph Construction and Search . . . . . 21
3.3 Data Collection . . . . . 21
3.3.1 Node Features . . . . . 23
3.3.2 Edge Features . . . . . 23
3.3.3 Global Graph Features . . . . . 23
3.4 Model Training . . . . . 24
3.4.1 Training Data . . . . . 24
3.4.2 Model Architecture . . . . . 24
3.4.3 Data Preprocessing . . . . . 25

**4 Experimental Results** 29
4.1 Characterizing ANN Datasets . . . . . 29
4.1.1 SIFT1M . . . . . 29
4.1.2 Deep1M . . . . . 30
4.1.3 Distribution of Queries . . . . . 30
4.2 Model Prediction Quality . . . . . 31

4.3 Impact of Features on Predictions . . . 33
4.4 Single Node and Edge Predictions . . . 35

**5 Future Work** **37**
5.1 Incorporating Deep Models . . . 37
5.2 Utilizing the Model for Benchmarking . . . 38
5.3 Improving Existing Graph Indices . . . 38

**6 Conclusion** **39**

**7 Artifact** **40**

**References** **42**

# List of Figures


2.1 An illustration of the lune; points $u$ and $v$ would not be connected. . . . . . . . . . 16

3.1 An overview of our proposed framework. Our contributions are labeled in the diagram with the section number where we discuss them in more details. . . . . . 20
3.2 Model architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.3 Distribution of node features across graphs for SIFT. . . . . . . . . . . . . . . . . . 26
3.4 Distribution of node features across graphs for SIFT. . . . . . . . . . . . . . . . . . 26
3.5 Global feature values across graphs for SIFT. . . . . . . . . . . . . . . . . . . . . 27
3.6 Correlation for SIFT. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

4.1 Distribution of the SIFT dataset on the first 5 dimensions. There is sometimes a small bump in distribution around 120. . . . . . . . . . . . . . . . . . . . . . . . 30
4.2 Distribution of the Deep1M dataset on the first 5 dimensions. . . . . . . . . . . . . 30
4.3 Accuracy and computation cost prediction on the test set, plotting predictions against the true cost or accuracy. . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.4 Impact of all features. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.5 The distribution of the predictions for a single node in the SIFT dataset, including the computation cost, accuracy and importance predictions. . . . . . . . . . . . . 36
4.6 The distribution of the predictions for a single edge in the SIFT dataset, including the computation cost, accuracy and importance predictions. . . . . . . . . . . . . 36

# List of Tables

3.1 Features and their predicted impact on the search process. . . . . . . . . . . . . . . . 22

# Chapter 1

# Introduction

Recent advances in deep learning have enabled the transformation of various data types (e.g., images, videos, documents) into high-dimensional vectors. This capability facilitates complex semantic analysis, which is the key to numerous online services such as search [1], model serving [2]–[4], eCommerce platforms [5], and recommendation systems [6]. In order to efficiently manage large-scale vector data, there is a huge trend in building *vector databases* and vector search engines, such as Meta's Faiss [7], Google's ScaNN [8], Microsoft's DiskANN [9], and Amazon DocumentDB, that integrate vector search systems with relational databases.

In a vector database, the major operation is to search the $k$ closest vectors to a given query vector, known as $k$-Nearest-Neighbor ($k$-NN) search. In real-world databases, the data size is often massive, e.g., billions of vectors. Therefore, exact $k$-NN search in vector database is extremely expensive in terms of computation. On the other hand, the online services on top of vector databases often require vector search to complete in milliseconds. Therefore, in practice, approximate Nearest-Neighbor (ANN) [10], [11] is often used instead. In the ANN approach, we usually build a search *index* offline, which can be leveraged to accelerate the search online. When we perform the search, instead of a brute-force search used in the exact $k$-NN search, we search only the most promising subareas in the database by following the index, and report the top-$k$ selections found in these subareas. This approximate approach creates a tradeoff between the

search speed and the search quality (i.e., accuracy), where one of the key design choices is how to build a high quality index.

There are mainly two types of ANN indexing approaches: clustering-based and graph-based indexing [12]. Clustering is a conventional approach, where the dataset is clustered offline and the online search is performed only within the most promising clusters. Graph-based indexing [13]–[15], however, first constructs a proximity graph on the dataset, connecting pairs of vectors that are close to each other, and then performs a graph traversal on the proximity graph for each query to find the closest vectors to a query vector. This approach has recently emerged as a more attractive solution than clustering, as it provides a more effective way to navigate the search, without being limited by the cluster boundaries. Its search performance, in terms of the scope of graph traversal that leads to convergence, is then highly dependent on the quality of the graph.

To improve the graph quality, lots of approximations and heuristic decisions [14]–[18] have been proposed in the literature, such as which connections (edges) to include or exclude and how to integrate new data points (nodes). However, no analysis or modeling work has been done to quantitatively evaluate the heuristics and their impact on the performance. As a result, it remains unclear how to choose the right heuristics to build a high-quality graph. For instance, graph-building methods might aim to restrict the degree of nodes, the expansion rate of the graph, or the existence of short paths between nodes. These properties cannot be controlled directly during graph-building. They are interdependent, and are often approximated to speed up the graph building process. Additionally, the performance claims made by different studies are difficult to verify, as implementation details can significantly affect search speed and accuracy.

We address these challenges by proposing a novel approach to quantitatively analyze different graph construction methods. The key challenge in quantifying the heuristics is the complex tradeoff between the search accuracy and search speed, which makes it almost impossible to establish an analytical model. To this end, we propose a machine learning based approach to help us with the modeling task. We first set up an unified performance evaluation framework in Section 3.1 to evaluate various graph building heuristics, by decoupling the graph construction

phase from the search phase (Section 3.2), and applying a common search process across different graph construction heuristics. We implement all the representative heuristics in our framework to enable fair comparison and eliminate implementation related noise in evaluation. Across multiple datasets and graph configurations, we then extract graph structural attributes as input features (Section 3.3), e.g., the diameter of the graph, and collect ground-truth data by running search instances in out framework and collecting the performance metrics in terms of the search speed and accuracy. Based on the collected data, we train a linear regression model (Section 3.4) to predict the search performance.

We implement the framework and conduct evaluation on two representative ANN datasets. Experimental results show that our model makes similar evaluations of heuristics across datasets, and can make reasonable predictions for per-query search performance. Based on the prediction results, we give implications on selecting heuristics that build a high-quality graph. Future work can be built on top of our framework to enhance the reliability and accuracy of ANN searches, facilitating faster and more precise data retrieval in large-scale applications.

The major contributions of this thesis are as follows.

- We propose a ML-based, systematic modeling method for evaluating the efficiency and predicting the performance of graph-based ANN indexing methods.
- We propose a data generation and analysis method for ANN indexing, by building an unified performance characterization framework and a feature extraction pipeline.
- We conduct training on various datasets, and offer insights into how graph properties influence ANN search performance, guiding future indexing designs with high quality graphs.

# Chapter 2

# Background

We first introduce Vector Database and Approximate Nearest Neighbor (ANN) search in Section 2.1. We then discuss existing ANN indexing methods in Section 2.2, and the ANN search algorithms in Section 2.3. Finally we describe existing efforts on using machine learning techniques to improve ANN performance in Section 2.4, which inspire our work.

## 2.1 Vector Database and ANN Search

Vector databases [19], [20] have emerged as the computational engines enabling effective interaction with vector embeddings in applications. This development follows the exponential rise of vector embeddings in fields such as NLP (Natural Language Processing), computer vision, and other AI applications. As a result, companies have all developed their own vector databases to manage the vast amounts of data they generate and collect, for their applications in knowledge search, recommendation systems, model serving, etc. By mapping user interests and content properties into high-dimensional vectors, these systems can find the nearest neighbors to a user's profile vector, and provide personalized content recommendations. For example, Google developed ScaNN (Scalable Nearest Neighbors) [8] for efficient vector similarity search at scale. Their implementation focuses on pruning the search space and providing better quantization techniques to efficiently compress the dimension of vectors. Facebook developed Faiss [7] for efficient similarity

search and clustering of dense vectors, which is used in their systems for ranking and recommendation purposes. Microsoft has developed DiskANN [9], which takes a graph-based approach, and is optimized for efficiency when the data is stored on disk. They propose methods for partitioning the graph, and using product quantization to reduce the memory required. Other examples of industry vector databases include Milvus [21], LVQ [22], [23], Pinecone [24], Meilisearch [25], Chroma [26], Weaviate [27], Deeplake [28], Qdrant [29], ElasticSearch [30], Vespa [31], Vald [32], PgVector [33].

The major operation in vector databases is finding *nearest neighbors*, i.e., the closest vectors, to a given query vector. Closeness is typically expressed in terms of a dissimilarity function: the less similar the objects, the larger the function values. Formally, the nearest-neighbor (NN) search problem is defined as follows: given a set $S$ of points in a space $M$ and a query point $q \in M$, find the closest point in $S$ to $q$. A direct generalization of this problem is a $k$-NN search, where we need to find the $k$ closest points. Exact $k$-NN search in vector database is computationally expensive, as in real-world applications, there could be billions of vectors in the database, each of which in hundreds of dimensions. Various solutions to the NNS problem have been proposed, including linear search and space partitioning. Linear search is a naive approach that has a running time of $O(dN)$, where $N$ is the cardinality of $S$ and $d$ is the dimensionality of $S$. Space-partitioning (e.g., $k$-d tree), however, reduces the complexity to $O(logN)$ on average and $O(kN^{1-1/k})$ in the worst case, by using the branch and bound methodology. Despite its improvement, the informal observation usually referred to as "the curse of dimensionality" [34] states that there is no general-purpose exact solution for NNS in high-dimensional Euclidean space using polynomial preprocessing and polylogarithmic search time.

Because of the high computational cost of exact $k$-NN search, in many applications where it is acceptable to retrieve a "good guess" of the nearest neighbors, approximate nearest-neighbor (ANN) [10], [11], [34], [35] is often used instead. ANN algorithms do not guarantee to return the actual nearest neighbor in every case, in return for improved speed or memory savings. Typical ANN algorithms include locality-sensitive hashing [34], best bin first and balanced box-

decomposition tree based search. ANN search plays a crucial role in managing and navigating through large datasets, especially when dealing with high-dimensional vectors. This technique is vital for efficiently retrieving information that closely matches a query from a massive pool of data, which is common in various applications such as image search, recommendation systems, and more.

Although it has been studied for decades, the ANN search problem, especially in high dimensions, remains a critical challenge and an area of active research. Meanwhile, due to the rise of deep learning, ANN search have recently become a hot topic in both academia and industry. For instance, recently there has been a Big-ANN competition [36] and benchmarking effort [37]. There also have been lots of research work in the past decade on characterizing [38]–[41] and optimizing vector databases [42]–[48] and ANN search [49]–[53] for GPU [15], [54]–[58], storage [9], [59]–[61], cloud [62], [63], CXL [64], [65] and distribution [66], [67].

With recent advancements in LLMs, vector databases and ANN search are frequently used for managing embeddings, allowing for functionalities like semantic search and document retrieval. Vector databases are used to store the embeddings of texts or user queries, which are then used to perform fast and efficient ANN searches to retrieve the most relevant information or documents, a process often refered to as Retrieval Augmented Generation (RAG). Models can rapidly find relevant pieces of information to add to its context during generation, to provide accurate domain-specific or time-sensitive responses.

## 2.2 ANN Indexing Methods

The techniques for approximate nearest neighbor search fall into two primary categories: clustering-based and graph-based indexing [12]. In this paper, we focus on graph-based indexing, which is a promising approach for high-dimensional data. In the rest of the section, we discuss previous graph indexing methods in greater detail.

### 2.2.1 Clustering-based vs. Graph-based Indexing

Clustering-based indexing [68]–[70] works by grouping nearby data points into the same bucket within a hierarchical structure. This method is notably efficient in lower-dimensional spaces. However, its performance tends to decline in higher-dimensional spaces, with decreased recall, which means that it becomes less likely to retrieve all relevant results. Note that, for clustering-based indexing, researchers have proposed quantization techniques, effectively reducing the dimensions of the dataset [22], [23], [71], [72]. The approach is also known as IVF (inverted file), where the index contains a mapping from cluster centroids to nodes in the cluster.

Graph-based indexing [16]–[18], [73] involves constructing a graph where data points are nodes, and pairs of nodes can be connected if they are close or meet some other criteria. The search process traverses this graph, maintaining a buffer of nodes to explore, and keeping the top $M$ nodes based on traversal priorities, usually determined by their distance to the query. The geometry of the points are abstracted as edges in the graph, so this approach is less sensitive to the dimension of the data.

**Other Variants of ANN Search**. In real-world applications, we often want to perform ANN search with extra conditions. ARKGraph [74] builds an index for searches within a specific search key range, so that the returned results satisfy an additional numerical filter. SeRF [75] proposes building a segmented graph to solve the range filter problem. LM-DiskANN [60] and SPFresh [76] propose ways to dynamically update the ANN index, so that nodes can be continuously inserted or deleted, lowering the required memory.

### 2.2.2 Existing Graph Indexing Methods

We introduce the diverse graph-construction methods by categorizing them in terms of their underlying theoretical graphs. The underlying graphs including $k$-nearest-neighbor graphs (KNN), minimum spanning trees (MST), and relative neighborhood graphs (RNG). All of these graphs are difficult to compute on large datasets, and previous papers present various ways to approximate

these graphs.

**KNN-based methods** focus on constructing graphs where edges connect each node to its $k$-nearest neighbors. Since constructing an accurate KNN graph can be slow, the NN-descent algorithm [73] aims to quickly approximate the KNN graph. At each step, each node considers the neighbors of its current neighbors, to find potential closer matches and update its outgoing connections to reach those matches.

**MST-based methods** approximating the minimum spanning tree of the dataset, which is where all points in the dataset are connected with the minimum possible total edge weight. The Hierarchical Clustering-Based Graph (HCNNG) [17] is built by partitioning points into smaller clusters, then computing the MST of these clusters individually. The process is obtained $k$ times with different partitions to achieve a graph with the desired edge density, and properties of MST. The Dynamic Exploration Graph [14] is built using a very different algorithm, but also aims to minimize the total edge length of the degree-$k$ graph. The algorithm incrementally adds nodes to the graph by connecting it to its closest matches with an undirected edge, and pruning and shuffling previous matches to minimize the total edge length.

**RNG-based methods** construct variants of Relative Neighborhood Graphs (RNG), which we define formally:

The relative neighborhood graph (RNG) is an undirected graph on a set of points, where $u$ and $v$ are connected, if and only if there does not exist a third point $w$ that is closer to both points than they are to each other, i.e.

$$e(u,v) \iff \max(d(w,u), d(w,v)) > d(u,v) \ (\forall w \in G).$$

Visually, this means nodes $u$ and $v$ are connected if there are no points darker intersection area in figure 2.1. The intersection area is referred to as the *lune*. In the figure, point $w$ is closer to $u$ and $v$ (red segments) than they are to each other (blue segment), so $u$ and $v$ are not connected in the RNG graph.

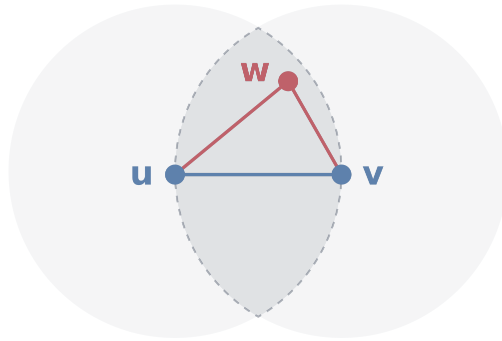


Figure 2.1: An illustration of the lune; points $u$ and $v$ would not be connected.

Popular graph construction methods that use the lune concept include Navigable Small World graphs (NSW) [18] and Navigating Spreading-Out Graphs (NSG) [16]. In NSW, nodes are incrementally added to the graph. A node $v$ is connected to its closest matches among the nodes that have already been added, with the lune restriction, that if there is a two-hop path from $v$ to $u$ where both segments are shorter, then the connection to $u$ is skipped. HNSW (Hierarchical Navigable Small World) graphs provide a variant where some nodes are added to higher layers, to provide a global-to-local search, accelerating the initial portion of the search with the longer skipping steps.

Navigating Spreading-Out Graphs are designed differently, with the goal of having monotone paths between any two points, so that no backtracking is required during search. However, the graph building process looks quite similar, where nodes are added incrementally, and they connect to the closest nodes previously added, unless the new edge would be the longest and therefore violate the lune condition.

In CAGRA, we first build a KNN graph with 2 or 3 times the desired number of edges using NN-descent, then prune edges with the most violations of the lune condition, which they refer to as detourable routes.

### 2.2.3 Parallelization in Building Graphs

Some work on approximate nearest neighbor search offer ways to parallelize the graph construction process. While the approximations introduced during parallelization is not the focus of this paper,

we include them as a future direction to pursue. In our current analysis, small changes in the graph do not greatly affect search performance, but as we iterate and improve our cost model, these changes might have a visible effect, and the tradeoff between graph-construction speed and the resulting search performance is an exciting topic to explore.

ParlayANN [77] parallelizes various incremental graph construction methods, where nodes used to be inserted sequentially. Instead, it inserts nodes in batch, and performs additional pruning and adjustments after insertion. CAGRA [15] parallelizes the calculation of detourable routes and the search process. In both works, the authors emphasize that the approximations from parallelization have little impact on the search behavior.

## 2.3 ANN Search Algorithms

In clustering-based ANN search algorithms, a query is first compared to cluster centroids, to find the nearest or most relevant clusters. Then, these clusters are fully searched, and the closest points from these clusters are returned.

The general search strategy for graph-based ANN search is best-first search (Algorithm 1). The search starts with the insertion of one or more starting nodes into the priority queue. As the search progresses, the algorithm extracts the highest priority node from the queue for expansion, examining its neighbors and evaluating their distances to the query. These neighbors are then inserted into the priority queue.

A key parameter in the search process is the query capacity $L$, which dictates the maximum number of nodes held in the priority queue at any time. This parameter directly influences the breadth and depth of the search, and a larger capacity always gives higher recall. Hence, this parameter is often varied to meet a required recall.

**Algorithm 1** Best-First Search (BFiS) [13]

```
1: Input: graph G, start P, query Q, query capacity L
2: Output: Approximate k nearest neighbors of Q
3: S ← ∅                                   ▷ Priority queue, sorted by distance to Q
4: i ← 0
5: compute dist(P, Q)
6: add P to S
7: while exists unchecked nodes in S do
8:     i ← index of first unchecked in S
9:     mark v_i as checked
10:    for u ∈ neighbors of v_i do
11:        if u is not visited then
12:            mark u as visited
13:            compute dist(u, Q)
14:            add u to S                  ▷ u is unchecked
15:        if S.size() > L then
16:            S.resize(L)
17: return first K of S
```

## 2.4 Machine Learning for ANN

Driven by recent deep learning successes across various fields, there have been intensive studies on applying machine learning techniques to improve ANN indexing and search performance. This has led to the development of numerous algorithms that outperform conventional methods, achieving state-of-the-art performance. AdaptNN [78] predicts termination conditions during runtime, and Tao [79] predicts termination using static features before the execution of a query search. Neural LSH [80] is a new learned space partitions of $\mathbb{R}^d$, where they employ supervised classification for graph partitioning, providing an improved clustering method.

# Chapter 3

# ANN Analysis Framework

## 3.1 Framework Overview

While researching previous graph construction methods, we find that graph indices often optimize for certain heuristics, such as the absence of triangles or shorter edge lengths. There are no quantitative results on how aspects of the graph might impact the search speed and accuracy, which makes designing new indices difficult.

Hence, we propose a framework for analyzing ANN search graph indices, shown in Fig. 3.1.

For each graph construction method and hyperparameter, we build a graph index. Then, we apply the same search algorithm to find the nearest neighbors of a set of queries. The two steps are decoupled (Section 3.2), which enables us to analyze the impacts of graph topology on the search performance.

Then, we discuss data collection in Section 3.3. We extract features from the graph indices, and pool node and edge-level features as inputs to the cost model, according to the search trajectory.

With the data, we train a cost model to predict the per-query search performance (Section 3.4). We preprocess features to ensure that the learned weights are interpretable, and use these features to predict the accuracy and computation cost of the search process.

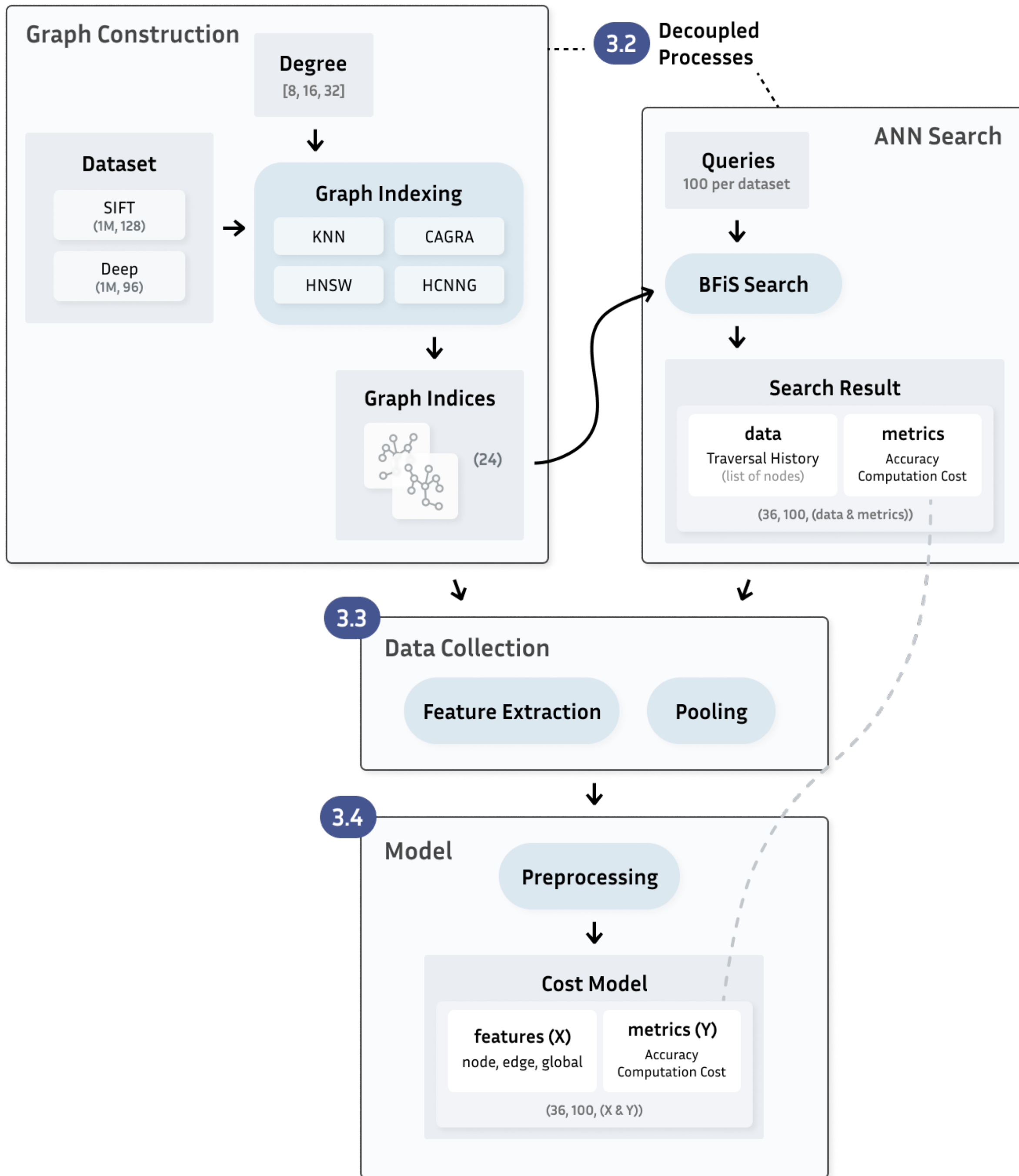


Figure 3.1: An overview of our proposed framework. Our contributions are labeled in the diagram with the section number where we discuss them in more details.

## 3.2 Decoupling of Graph Construction and Search

We implement and analyze a selection of graph construction methods, covering each theoretical graph for ANN search. Each resulting graph is saved in the same format, and can be searched the same way. We focused on the K-nearest neighbor (KNN) graph, a flattened version of Hierarchical Navigable Small World (HNSW) graph, the Cuda ANNS Graph (CAGRA), and the Hierarchical Clustering-based Nearest Neighbor Graph (HCNNG) (Section 2.2). We use the datasets SIFT1M and Deep1M (Section 4.1). For each graph, on each dataset, we build 3 versions with average degrees of 8, 16 and 32. We obtain 12 graph indices for each dataset.

We chose to reimplement the graph construction methods, so that the construction process follows a similar format. In our present analysis, it allows us to use our graph indices independent of the graph-building process. In the future, this also enables us to analyze the impact of heuristics for the graph construction process.

For each graph index, we adopted the same best first search algorithm (Section 2.3) to ensure that the differences observed in our experiments could be attributed directly to the graph structure rather than variations in the search strategy. This way, our search is decoupled from the graph construction step. In our experiments, we always used a query capacity $L$ of 200, to search for the top 50 neighbors. The query capacity cannot be inferred from the graph structure or features we collect, so we do not vary the query capacity across trials.

## 3.3 Data Collection

For the data collection module, we use data from the graph indices (feature extraction) and the search process (pooling), to prepare inputs to our cost model.

For feature extraction from graphs, we consider features at three levels: node, edge, and global, each contributing uniquely to the search dynamics.

After collecting the features of the graph indices, for each query, we only pass a subset of the

node and edge level features to the cost model for that query. The pooling step in data collection selects features of nodes and edges that were traversed during the search process, since only these nodes or edges were seen by the search engine at any time, while other nodes and edges cannot affect the search outcome.

In table 3.1, we summarize the features, including their relation to heuristics used in current graph construction methods. For example, as edge length increases, we predict that the computation cost increases and the accuracy decreases, as minimizing edge length is something all graph indices care about.

| **Name** | **Level** | **Relation to Graph Structure** | **Cost (pred)** | **Acc (pred)** |
|---|---|---|---|---|
| Degree | Node | MST has hubs | + | + |
| Clustering Coefficient | Node | lower for RNG | + | − |
| Centrality | Node | higher for MST | − | ~ |
| Edge Length Mean | Node | unsure | + | − |
| Edge Length Variance | Node | higher for HNSW | + | − |
| Edge Lengths | Edge | unsure | + | − |
| Average Hop Distance | Global | small for HNSW | − | − |
| SCC Count | Global | close to 1 unless KNN | + | − |

Table 3.1: Features and their predicted impact on the search process.

Many of the features are difficult to compute exactly for a large graph, so we rely on sampling methods to get approximations of the features. For example, we randomly sample pairs of points to compute their pairwise shortest path, which is used to compute the centrality and average hop distance. Since we are comparing between graph topologies, bias in the approximation will not be too impactful (for example, the sampled diameter will always be less than the true diameter).

Originally, we also included features that are specific to the query, such as the distance from each node to the query. However, we discovered that these features can reveal additional information about the search process, which obscures the prediction results of out cost model. In particular, if a node is close to a query, it might guess that the accuracy is high. Additionally, this information is not useful for graph index building, since it doesn't concern the graph topology.

Hence, we do not include any features that vary across queries.

### 3.3.1 Node Features

We look at node-level features such as degree, centrality (the proportion of shortest paths through it), and clustering coefficient (the proportion of its neighbors that are connected). The degree of nodes determine the expansion rate of the search, and it is not obvious whether a higher degree is always better. The average and variance of a node's outgoing edge lengths offer insights into the position of the node in the dataset, crucial for determining effective traversal paths during searches. Features like betweenness centrality are indicative of a node's centrality in the graph, suggesting its role in connecting major parts of the graph, thereby guiding more efficient search paths.

### 3.3.2 Edge Features

Edge length directly affects the search process as shorter edges typically facilitate faster convergence to the nearest neighbors. The centrality of an edge, measured by how many shortest paths pass through it, indicates its importance in the structural navigability of the graph. Additionally, edges that bridge nodes of varying centrality might be crucial for linking disparate parts of the graph.

### 3.3.3 Global Graph Features

Features such as the diameter, the number of strongly connected components, and structural densities including cycles, triangles, and $k$-clique density provide a macroscopic view of the graph. These metrics can reflect the graph's overall navigability and the complexity of its structure.

## 3.4 Model Training

We aim to train a model that predicts the goodness of a node or edge in the search process. To do so, we first construct a dataset from features to query performance, and train a regression model.

### 3.4.1 Training Data

For each ANN dataset, we build 12 different graphs (4 methods, 3 degree parameters), and run 100 queries through each graph. For each query, we collect node features of the nodes we visit during the process, features of the edges contained in the subgraph of visited nodes, and some global features outlined in the previous subsection.

Our curated dataset contains features for each graph, along with the computation cost and accuracy outcomes of that particular query on the specified graph. To track the computation cost, we use the number of distance computations required during the search process.

We split our dataset randomly into train and test sets, and report the prediction results on the test set.

### 3.4.2 Model Architecture

The model architecture is shown in Fig. 3.2. It incorporates three linear regression components, for the three feature levels, detailed in Table 3.1. Each linear layer predicts the search performance (accuracy and number of distance computations), given only the features of its corresponding level. The node and edge layers also predict an importance weight for that node or edge.

Then, the node-level predictions are pooled across all visited nodes during search, according to the (softmaxed) predicted importances. We choose to pool over the visited nodes only, because changes to other nodes will not affect the search result, so only the node features of visited nodes should contribute to the final prediction. Similarly, the edge-level predictions are pooled across all edges within the subgraph of the visited nodes, according to their predicted importances.

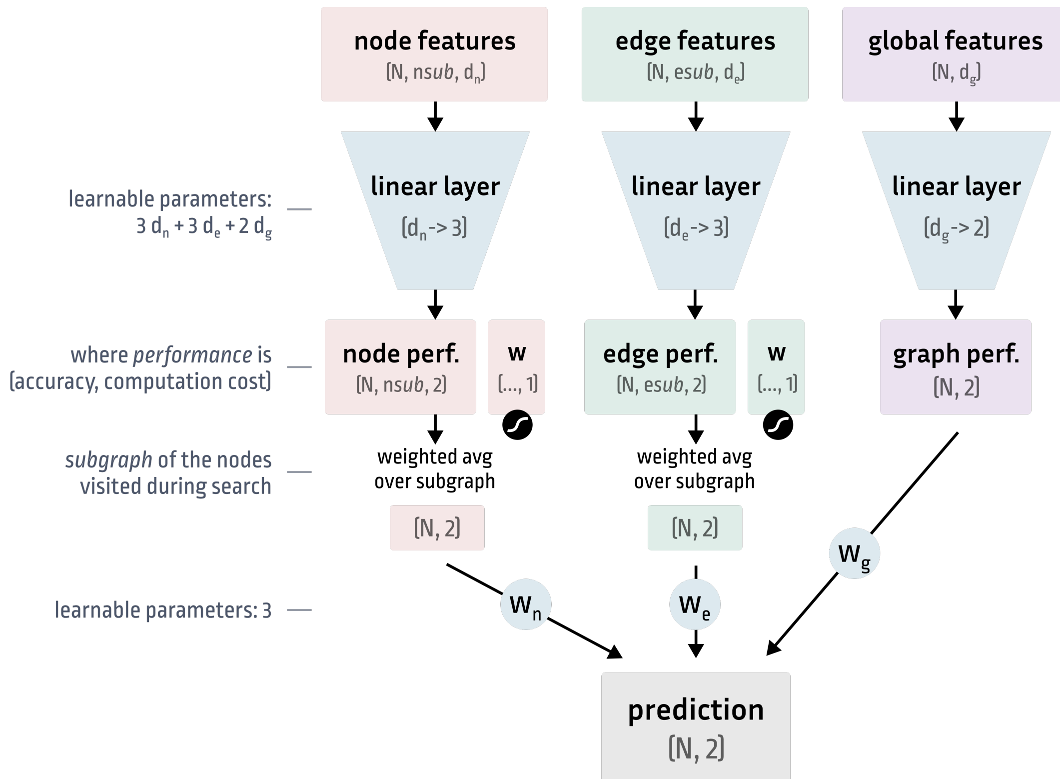


Figure 3.2: Model architecture.

At the end, we take a weighted sum of the node-level predictions, edge-level predictions, and global predictions, to form the overall prediction.

The weights in the linear layers are trainable, and the final weights for the sum are trainable.

The model design is quite simple, with few nonlinearities. We want to ensure that each node or edge impacts the prediction independently. With this setup, it is possible to use the node features linear regression layer to evaluate the impact of a single node on the overall search performance, and adjust the connections in its neighborhood during the graph construction process.

### 3.4.3 Data Preprocessing

We first check that our computed features make sense (Fig. 3.3, Fig. 3.4, Fig. 3.5): they are in a somewhat correct range, and they differ across different graphs.

We see some variance across features and graph degrees, as expected.

Before any experiments, we standardize each feature to have mean of 0 and variance of 1.

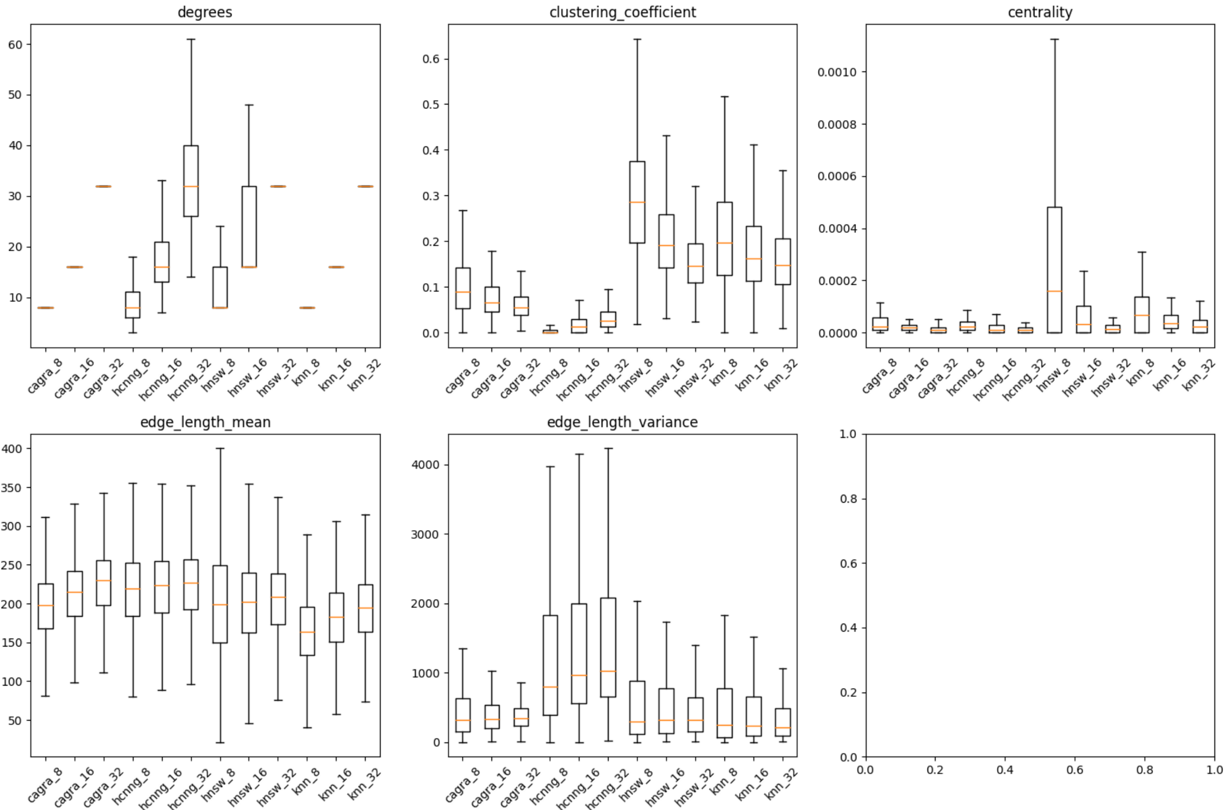


Figure 3.3: Distribution of node features across graphs for SIFT.

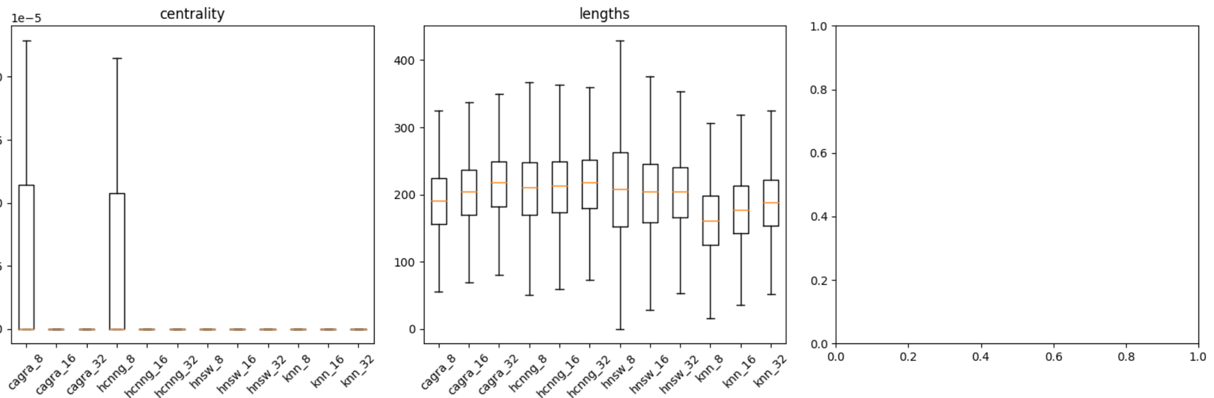


Figure 3.4: Distribution of node features across graphs for SIFT.

Then, we check whether the features we collect are correlated (Fig. 3.6). The dimension of different types of features (node, edge or global) are different. To compute the correlation between global features and node or edge features, we use the average of the node or edge features. We cannot easily compute the correlation between node and edge features, so we leave it out. We

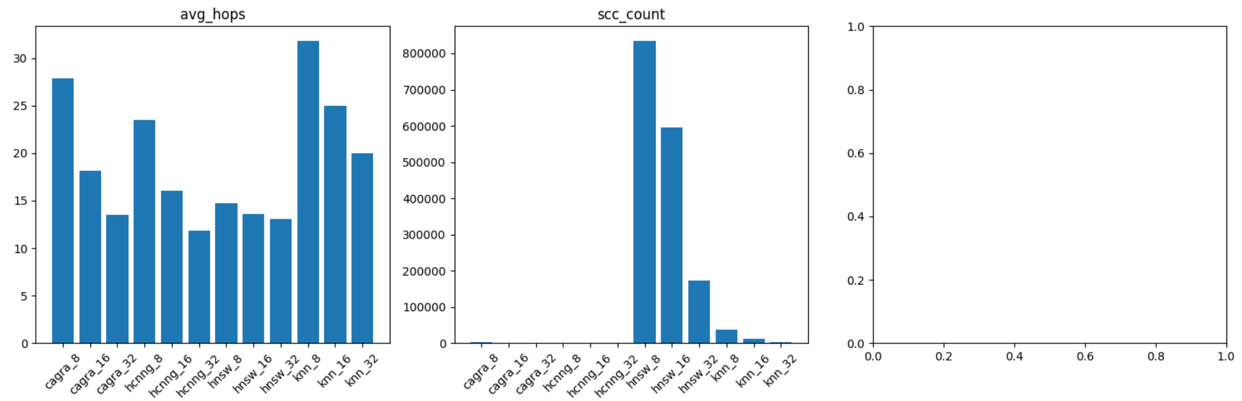


Figure 3.5: Global feature values across graphs for SIFT.

removed some that are clearly correlated. For example, clustering coefficient and 2-hop expansion rate calculate similar concepts (triangle density). Below, we show the correlation of the features we use, in the SIFT dataset.

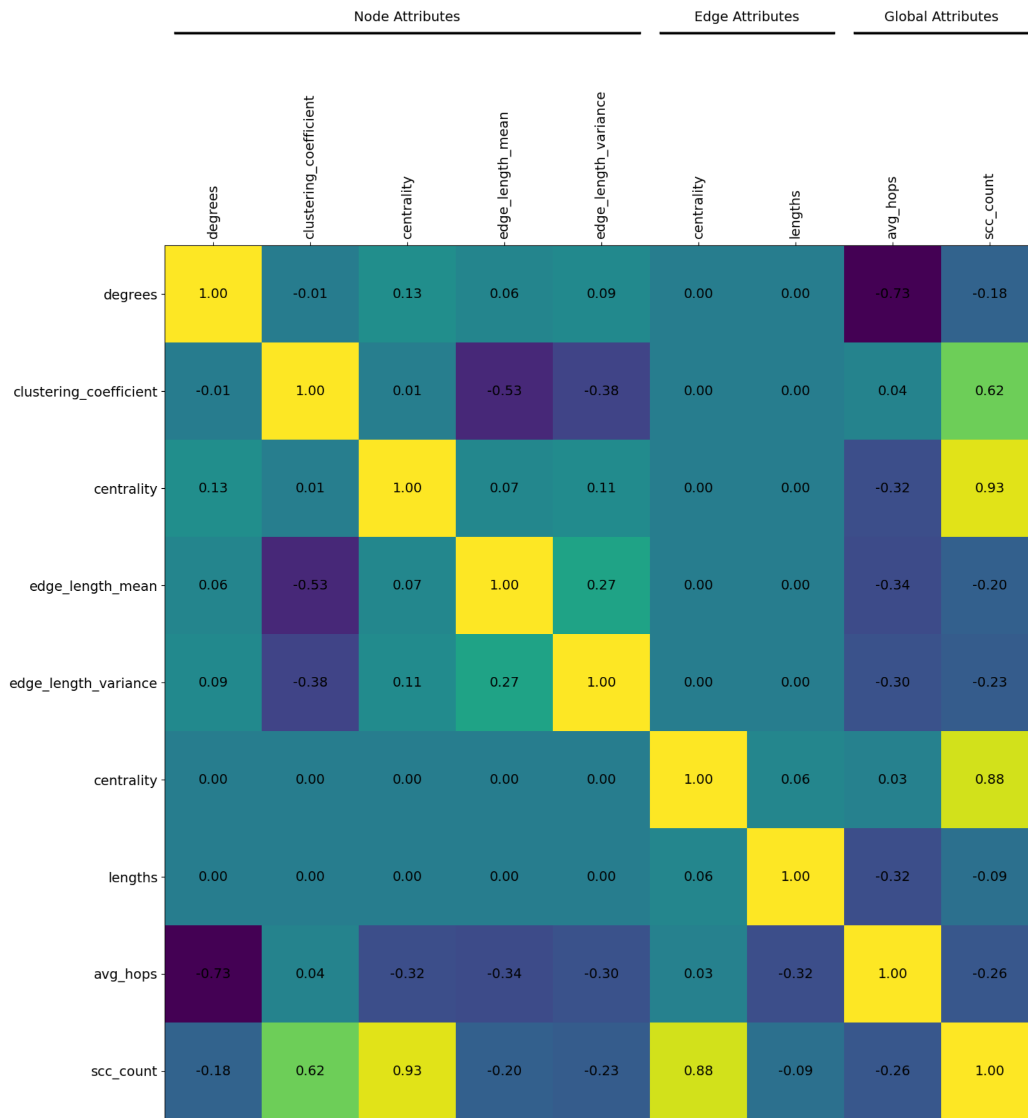


Figure 3.6: Correlation for SIFT.

# Chapter 4

# Experimental Results

In this chapter, we detail the outcomes of our analyses. We first characterize the ANN datasets that we use for our experiments. In section 4.2, we verify that our regression model has some ability to predict the search performance, based on our relatively simple features. In section 4.3, we look at the weights in the model, to understand how each feature impacts the search performance. The artifact of our evaluation can be found in Chapter 7.

## 4.1 Characterizing ANN Datasets

The performance and efficiency of graph-based approximate nearest neighbor search methods can be influenced by the characteristics of the underlying data distributions. In our experiments, we look at two datasets that are commonly used for ANN benchmarking: SIFT and Deep. They follow different distributions, and we hope that our cost model is robust enough to be used for both datasets.

### 4.1.1 SIFT1M

The Scale-Invariant Feature Transform (SIFT) dataset [81] consists of high-dimensional vectors derived from real-world images. These vectors represent distinct image features that are invariant

to image scale, rotation, and lighting changes. The SIFT dataset is widely used in computer vision and image retrieval tasks. SIFT1M is a subset of the full dataset.

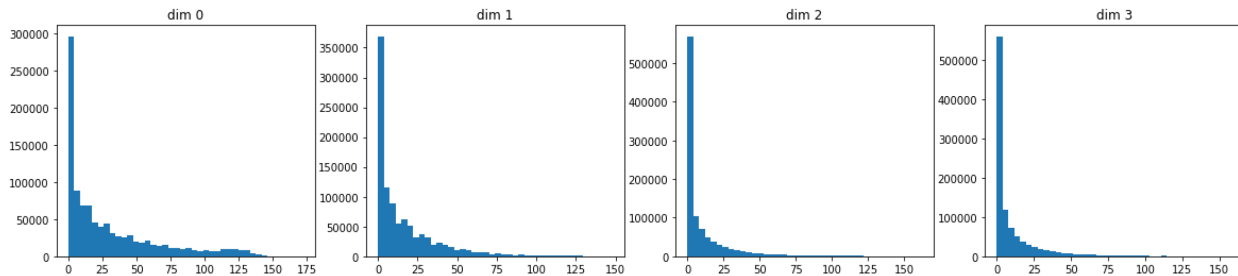


Figure 4.1: Distribution of the SIFT dataset on the first 5 dimensions. There is sometimes a small bump in distribution around 120.

### 4.1.2 Deep1M

The Deep dataset consists of image embeddings produced as the outputs from the last fully-connected layer of the GoogLeNet model [82]. Deep1M is a subset of the full dataset.

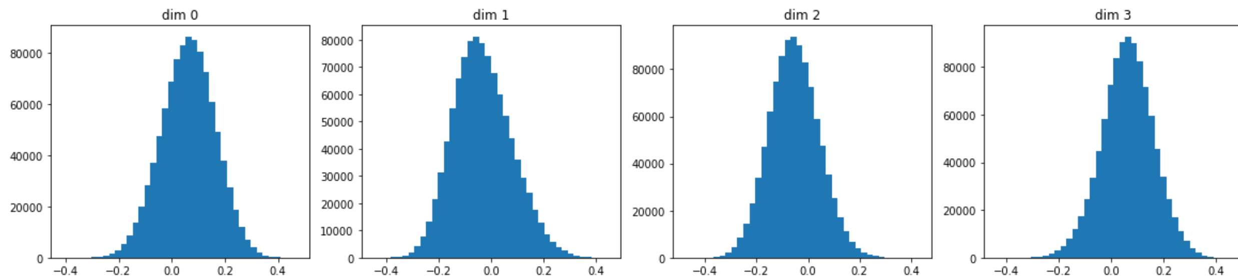


Figure 4.2: Distribution of the Deep1M dataset on the first 5 dimensions.

We see that the two datasets follow different distributions.

### 4.1.3 Distribution of Queries

In most datasets, the training data (on which the graph is built) and the test data (the query points) follow the same distribution.

However, real-world scenarios often challenge this assumption with the introduction of queries that are out-of-distribution (OOD). OOD queries are those that significantly deviate from

the distribution characteristics of the training set. Previous works have shown that OOD queries greatly affect the performance of the search algorithm.

In particular, they note that graph methods are more suited than clustering methods for OOD tasks. Clustering methods partition the data space into regions, and search in the most promising ones. While this approach is efficient for in-distribution queries, it becomes a limitation when the query falls outside these established regions. Graph-based indices demonstrate some resilience to OOD queries, However, there are still decreases in performance when the query is far from the training data.

## 4.2 Model Prediction Quality

We train our cost model, and plot the predicted computation cost and accuracy against the true search results in Fig. 4.3. The points are color coded by their graph indexing method, and they are in different shapes according to their degree.

The accuracy of an ANN search process is usually reported as the mean accuracy across all queries. To verify that our model reliably predicts this mean accuracy, we aggregate the accuracy by graph, resulting in 12 data points, and also plot them. These points closely align with the diagonal. However, to establish this correlation with higher confidence, a larger dataset is required.

First, we make some observations about the computation cost and accuracy across graph indices. We see that HNSW has a low accuracy, especially at degree 8. This indicates that our implementation does not match the official description. The recall for an official unflattened HNSW graph (with potentially a different search method and layer count), with degree 8, is around 40%. The discrepancy could be due to that we flattened the graph. The inaccurate implementation does not impact the correctness of our analysis, since we will just be using a poorer quality graph index. We also see that triangular points (degree 32) are concentrated on the top right of the graphs, indicating that a high degree graph increases distance computation costs, and increases accuracy, as we would expect.

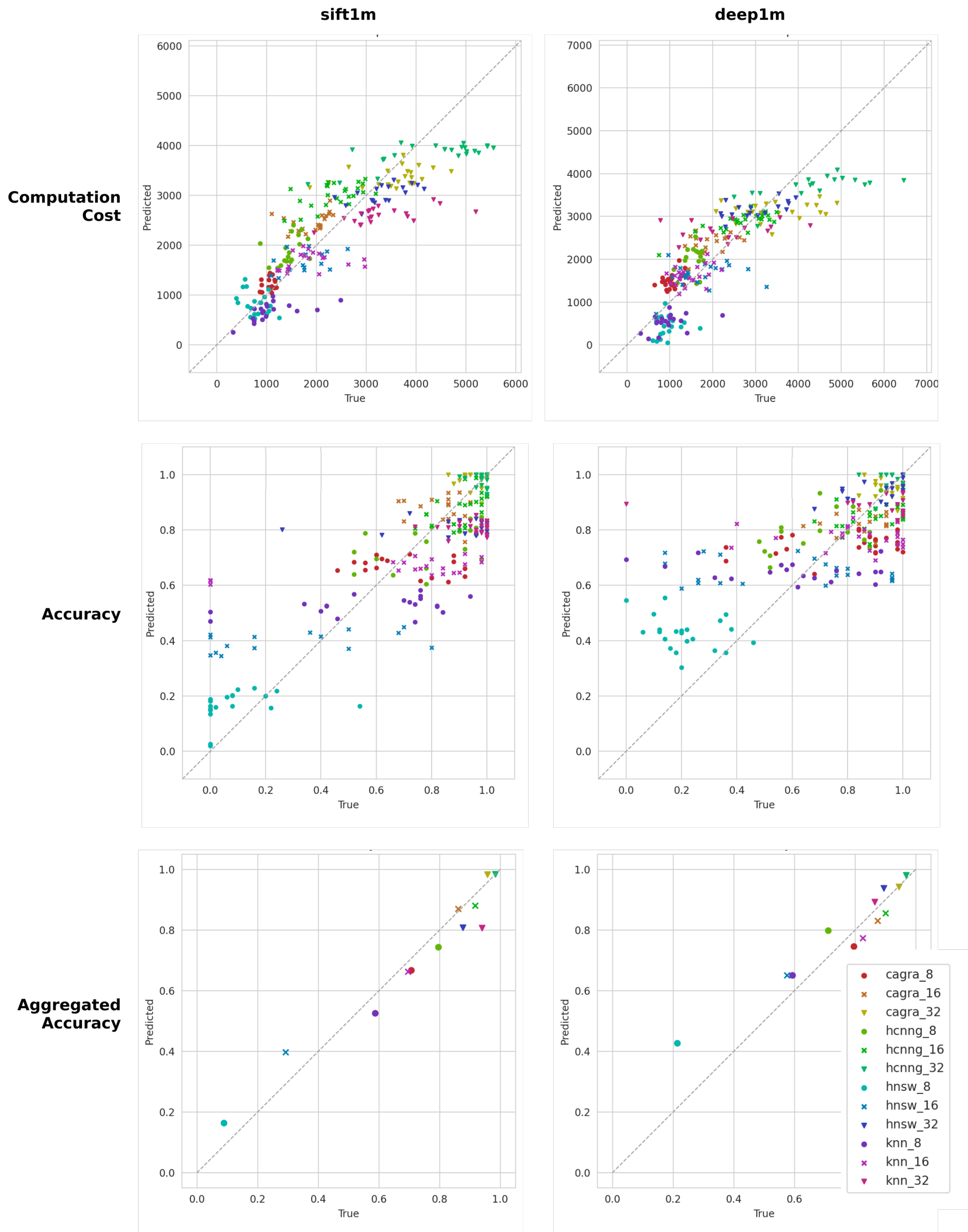


Figure 4.3: Accuracy and computation cost prediction on the test set, plotting predictions against the true cost or accuracy.

Then, we look at the quality of predictions. We see that while the model seems to predict the computation cost and accuracy to some extent (points lie close to the diagonal), it struggles to differentiate between queries. For the same graph index (same color), the model's predictions are quite similar (approximately in the same row).

It is likely that with more layers in the network or with higher order features, we can make much better predictions. However, for interpretability, we keep the simpler features to ensure that our model can in fact capture how the graph features and construction heuristics impact the search performance, without overfitting to more complicated features.

## 4.3 Impact of Features on Predictions

We visualize the impact of each feature on the prediction in Fig. 4.4, according to the corresponding weights in the model. We create a combined plot that includes node features, edge features, and overall graph features, where the different feature levels are weighed by the parameters of the final layer of the model.

Let us interpret the result. First, note that the parameter weights for both datasets are visually quite similar. The two models are trained independently, so the similarity shows that the model is stable across different data distributions, and can be applied to graphs built for new datasets.

Then, we look at the predicted impacts of each feature. Note that the two bars have opposite desirable directions: we would like to reduce the number of computations (blue bar) and increase the accuracy (orange bar).

Features where the two bars are in the same direction provide some tradeoff; features where the bars are in opposite directions show that the heuristic can be tuned in some direction.

If we increase the **degree** of the graph (entry 1), we expand to more nodes at a time, so we are less likely to miss certain regions (higher accuracy), but each expansion requires more computations. This aligns with what we observe from the results plot, and our expectations.

If we increase the **centrality** of nodes or edges (entries 3 or 6), we observe another tradeoff. As

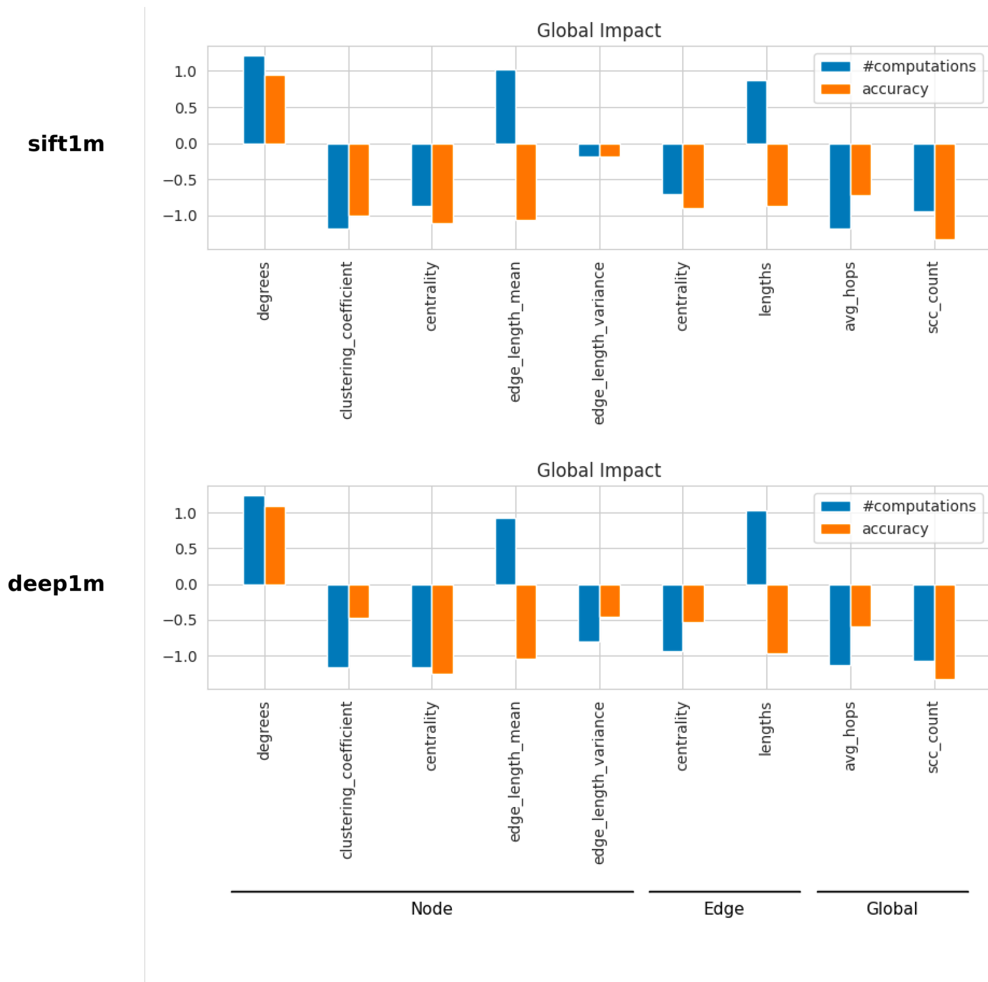


Figure 4.4: Impact of all features.

more nodes or edges among the selected subgraph lie on many shortest paths, the model predicts lower computation costs, and lower accuracy. This could be because sparser graphs tend to have nodes with high centrality, and lower degree leads to lower computation cost and accuracy.

If we decrease the **edge length** (entries 4 and 7), we get both better accuracy and smaller computation costs. Having short edge lengths is crucial in all graph indices, as they effectively translate some metric space distance to a graph hop distance. If the edge lengths are short, then the two metrics are more aligned. We would like to note, however, that since we are only comparing against other existing ANN graph indices, our samples don't include graphs with very long edge lengths, so this result might not generalize to more random graph-building methods.

If we decrease the **clustering coefficient** (triangle density) around a node, we see a higher

computation cost, and a higher accuracy. In RNG graphs, it is claimed that the third edge of a triangle can be omitted, so more nodes can be searched over the same number of expansions, allowing the search algorithm to find the nearest neighbors in the same number of hops. Since we count the distance computations, encountering more nodes will still increase the computation cost, but this result suggests that we should include more realistic metrics beyond distance computation costs. In particular, the distance to a batch of neighbors are often computed in parallel, so it would be useful to predict the number of expansions as well.

For the global features, if the graph has a low **average hop distance** between connected nodes, we see an increase in computation costs, and an increase in strongly connected components. This is largely because average hops is highly anti-correlated with degree.

Finally, if the graph has more **strongly connected components** (so it's more disconnected), the accuracy decreases, and the computation cost decreases as there is less space to explore.

To summarize, the only clear heuristic we corroborate is that graphs should have shorter edge lengths. For other features, we would want to collect data across more graphs to make generalized statements about our result.

## 4.4 Single Node and Edge Predictions

We check that the predictions for individual nodes and edges vary, so that they can be used to prune or add edges to a graph. We compute the prediction for each node or edge, weighted by the parameters in the final layer.

For the nodes, we see in Fig. 4.5 that the some nodes predict unusually high computation costs or unusually low accuracies. The nodes also vary in importance, although only by a few times due to regularization in the model.

Similarly, for the edges, we see in Fig. 4.6 that the model predicts varying results. We notice that the distributions are more regular, which could indicate that there are fewer extremely good or bad edges, or that the model does not distinguish between edges too well.

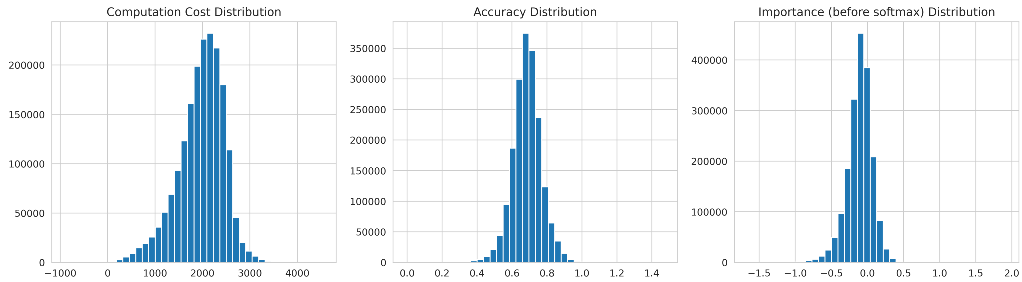


Figure 4.5: The distribution of the predictions for a single node in the SIFT dataset, including the computation cost, accuracy and importance predictions.

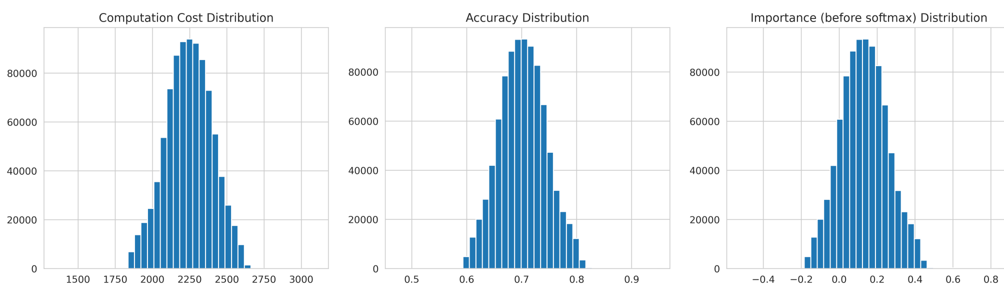


Figure 4.6: The distribution of the predictions for a single edge in the SIFT dataset, including the computation cost, accuracy and importance predictions.

# Chapter 5

# Future Work

## 5.1 Incorporating Deep Models

We would like to enhance our model's capabilities, by incorporating deeper models. We would like to integrate higher order features that capture more complex patterns of the graph topology.

To support the newly added features without overfitting, we would like to build more graph indices. We consider adding indices like the dynamic exploration graph (DEG) or the navigating spreading-out graph (NSG). We consider making modifications to existing indices, such as skipping the pruning step of HNSW or CAGRA, to obtain alternative versions of the indices. We also consider building graphs by pruning a random graph using our existing model.

For modifications on our model architecture, we propose expanding the network to include multiple layers. This can allow the model to construct more abstract representations using the current local features and heuristics. Additionally, we would like to employ trainable graph convolution layers that can dynamically learn and extract relevant node-level, edge-level and graph-level features from the data. The outputs of the graph convolution layers captures more complex local features that are impactful for the graph quality.

By integrating theses features, we hope to improve the accuracy of the cost model. While we sacrifice some interpretability, the model can still be used directly to evaluate and improve graph

indices.

## 5.2 Utilizing the Model for Benchmarking

Our analysis framework offers a way to predict the performance of newly proposed heuristics, using data from existing indices.

Currently, we train our model on two datasets, and check that the parameters are consistent for the two cases. This consistency is a positive indicator of the model's robustness and its ability to generalize across similar distributions.

In the future, we would like to apply our model more confidently to more use cases, extending to out of distribution queries, new data distributions and new heuristics. We aim to generate a broader range of plausible graph indices to enhance the model's robustness. This can allow future researchers to identify potential limitations and optimize their strategies in earlier stages of development.

## 5.3 Improving Existing Graph Indices

We would like to use the cost model to assist the construction of better graph indices. Our model predicts the cost and accuracy of each node separately. Then, given some graph index, we can evaluate the contributions of each node and edge in the index. We can add edges that improve local nodes or have high performance predictions, and remove edges that don't. If we use GCN features, then it is easier to apply the model for edge pruning, since all the features are locally computable.

# Chapter 6

# Conclusion

We demonstrated that certain graph structural properties significantly influence the efficiency of ANN searches. The regression model used in the study offers predictive insights into how specific node and edge features affect search outcomes, which can be used in the future for edge pruning. The findings suggest that certain graph features, like edge lengths and node centrality, have a critical impact on reducing computational costs and improving accuracy, while other features offer a tradeoff between computation cost and accuracy. These results pave the way for more quantitative analyses of the impact of graph structure on the search performance.

In the future, we would like to apply the current model to improve the graph construction process. We would prune existing graphs under the guidance of the cost models. Additionally, we would like to improve the capacity of the model. It will be interesting to incorporate more diverse features from a graph convolution layers. The results might be harder to interpret, but will nontheless be applicable during the graph construction process. We would also like to incorporate features that arise from parallelizing incremental graph building processes, to better understand the tradeoffs made.

# Chapter 7

# Artifact

## Abstract

This artifact appendix helps the readers reproduce the main evaluation results of this project.

## Contents

The details of the contained code and how to run graph indexing methods are described at

https://github.com/chenxuhao/Big-ANN/blob/main/README.md.

The details of data processing and model training are described at

https://github.com/storyscene/ann-local-analysis-code/blob/main/README.md

## Hosting

The source code of this artifact can be found at

https://github.com/chenxuhao/Big-ANN.

### Requirements

#### Software dependencies

This artifact requires CUDA 11.8.0 and GCC 11.2.0 or greater.

# References

[1] J. Mohoney, A. Pacaci, S. R. Chowdhury, A. Mousavi, I. F. Ilyas, U. F. Minhas, J. Pound, and T. Rekatsinas, "High-throughput vector similarity search in knowledge graphs," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–25, 2023.

[2] A. Eichenberger, Q. Lin, S. Masood, H. Min, A. Sim, J. Wang, Y. Wang, K. Wu, B. Yuan, L. Zhou, *et al.*, "Serving deep learning model in relational databases," *arXiv preprint arXiv:2310.04696*, 2023.

[3] J. Xian, T. Teofili, R. Pradeep, and J. Lin, "Vector search with openai embeddings: Lucene is all you need," in *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, 2024, pp. 1090–1093.

[4] Z. Jing, Y. Su, Y. Han, B. Yuan, C. Liu, H. Xu, and K. Chen, "When large language models meet vector databases: A survey," *arXiv preprint arXiv:2402.01763*, 2024.

[5] S. Kragting, "Anomaly detection with similarity graphs and active learning building and storing static and dynamic similarity graphs with the help of a vector database," M.S. thesis, 2022.

[6] X. Zhao, M. Wang, X. Zhao, J. Li, S. Zhou, D. Yin, Q. Li, J. Tang, and R. Guo, "Embedding in recommender systems: A survey," *arXiv preprint arXiv:2310.18608*, 2023.

[7] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," 2024. arXiv: 2401.08281 [cs.LG].

[8] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar, "Accelerating large-scale inference with anisotropic vector quantization," in *International Conference on Machine Learning*, 2020. URL: https://arxiv.org/abs/1908.10396.

[9] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[10] Y. Tian, Z. Yue, R. Zhang, X. Zhao, B. Zheng, and X. Zhou, "Approximate nearest neighbor search in high dimensional vector databases: Current research and future directions," *Bulletin of the Technical Community on Data Engineering*, vol. 47, no. 3, 2023.

[11] M. Aumüller and M. Ceccarello, "Recent approaches and trends in approximate nearest neighbor search, with remarks on benchmarking," *Data Engineering*, p. 89,

[12] Z. Wang, P. Wang, T. Palpanas, and W. Wang, "Graph-and tree-based indexes for high-dimensional vector similarity search: Analyses, comparisons, and future directions," *Data Engineering*, pp. 3–21, 2023.

[13] Z. Peng, M. Zhang, K. Li, R. Jin, and B. Ren, "Iqan: Fast and accurate vector search with efficient intra-query parallelism on multi-core architectures," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 313–328.

[14] N. Hezel, K. U. Barthel, K. Schall, and K. Jung, *Fast approximate nearest neighbor search with a dynamic exploration graph using continuous refinement*, 2023. arXiv: 2307.10479 [cs.IR].

[15] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, *Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus*, 2023. arXiv: 2308.15136 [cs.DS].

[16] C. Fu, C. Xiang, C. Wang, and D. Cai, *Fast approximate nearest neighbor search with the navigating spreading-out graph*, 2018. arXiv: 1707.00143 [cs.LG].

[17] J. Vargas Muñoz, M. A. Gonçalves, Z. Dias, and R. da S. Torres, "Hierarchical clustering-based graphs for large scale approximate nearest neighbor search," *Pattern Recognition*, vol. 96, p. 106 970, 2019, ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2019.106970. URL: https://www.sciencedirect.com/science/article/pii/S0031320319302730.

[18] Y. A. Malkov and D. A. Yashunin, *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs*, 2018. arXiv: 1603.09320 [cs.DS].

[19] T. Taipalus, "Vector database management systems: Fundamental concepts, use-cases, and current challenges," *Cognitive Systems Research*, p. 101 216, 2024.

[20] J. Jie Pan, J. Wang, and G. Li, "Survey of vector database management systems," *arXiv preprint arXiv:2310.14021*, 2023.

[21] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2614–2627.

[22] C. Aguerrebere, I. S. Bhati, M. Hildebrand, M. Tepper, and T. Willke, "Similarity search in the blink of an eye with compressed indices," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 3433–3446, Jul. 2023, ISSN: 2150-8097. DOI: 10.14778/3611479.3611537. URL: https://doi.org/10.14778/3611479.3611537.

[23] C. Aguerrebere, M. Hildebrand, I. S. Bhati, T. Willke, and M. Tepper, "Locally-adaptive quantization for streaming vector search," *arXiv preprint arXiv:2402.02044*, 2024.

[24] P. Systems, *Pinecone product overview*, 2022. URL: https://www.pinecone.io/product/.

[25] M. D. v0.27, *What is meilisearch?* Jun. 2022. URL: https://docs.meilisearch.com/learn/what_is_meilisearch/overview.html/.

[26] chroma-core, *Chroma - the open-source embedding database*, 2024. URL: https://github.com/chroma-core/chroma/blob/main/README.md.

[27] Weaviate, *The ai-native vector database*, 2024. URL: https://weaviate.io/platform.

[28] S. Hambardzumyan, A. Tuli, L. Ghukasyan, F. Rahman, H. Topchyan, D. Isayan, M. McQuade, M. Harutyunyan, T. Hakobyan, I. Stranic, *et al.*, "Deep lake: A lakehouse for deep learning," *arXiv preprint arXiv:2209.10785*, 2022.

[29] Qdrant, *Qdrant. efficient, scalable, fast.* 2024. URL: https://qdrant.tech/qdrant-vector-database/.

[30] Elastic, *Elasticsearch*, 2024. URL: https://github.com/elastic/elasticsearch/blob/main/README.asciidoc.

[31] vespa, *Big data + ai, online.* 2024. URL: https://vespa.ai/.

[32] V. team, *Vald a highly scalable distributed vector search engine*, 2024. URL: https://vald.vdaas.org/.

[33] pgvector, *Pgvector*, 2024. URL: https://github.com/pgvector/pgvector/blob/master/README.md.

[34] 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*, 1998, pp. 604–613.

[35] T. Liu, A. Moore, K. Yang, and A. Gray, "An investigation of practical approximate nearest neighbor algorithms," *Advances in neural information processing systems*, vol. 17, 2004.

[36] Big-ANN, *Neurips'23 competition track: Big-ann*, 2023. URL: https://big-ann-benchmarks.com/neurips23.html.

[37] big-ann-benchmarks, *Practical vector search: Neurips 2023 competition*, 2023. URL: https://github.com/harsha-simhadri/big-ann-benchmarks/.

[38] 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*, vol. 32, no. 8, pp. 1475–1488, 2019.

[39] R. Cheng, Y. Peng, X. Wei, H. Xie, R. Chen, S. Shen, and H. Chen, "Characterizing the dilemma of performance and index size in billion-scale vector search and breaking it with second-tier memory," *arXiv preprint arXiv:2405.03267*, 2024.

[40] Y. Zhang, S. Liu, and J. Wang, "Are there fundamental limitations in supporting vector data management in relational databases? a case study of postgresql," ICDE, 2024.

[41] Z. Wang, H. Xiong, Z. He, P. Wang, *et al.*, "Distance comparison operators for approximate nearest neighbor search: Exploration and benchmark," *arXiv preprint arXiv:2403.13491*, 2024.

[42] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang, *et al.*, "{Vbase}: Unifying online vector similarity search and relational queries via relaxed monotonicity," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 377–395.

[43] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, "Acorn: Performant and predicate-agnostic search over vector embeddings and structured data," *arXiv preprint arXiv:2403.04871*, 2024.

[44] J. Pan, J. Wang, and G. Li, "Vector database management techniques and systems," in *Companion of the International Conference on Management of Data (SIGMOD)*, 2024.

[45] Y. Jin, Y. Wu, W. Hu, B. M. Maggs, X. Zhang, and D. Zhuo, "Curator: Efficient indexing for multi-tenant vector databases," *arXiv preprint arXiv:2401.07119*, 2024.

[46] T. Yang, W. Hu, W. Peng, Y. Li, J. Li, G. Wang, and X. Liu, "Vdtuner: Automated performance tuning for vector data management systems," *arXiv preprint arXiv:2404.10413*, 2024.

[47] Z. Bao, L. Liao-Liao, Z. Wu, Y. Zhou, D. Fan, M. Aibin, and Y. Coady, "Delta tensor: Efficient vector and tensor storage in delta lake," *arXiv preprint arXiv:2405.03708*, 2024.

[48] R. Wu, J. Meng, J. J. Xu, H. Wang, and K. Rong, "Rethinking similarity search: Embracing smarter mechanisms over smarter data," *arXiv preprint arXiv:2308.00909*, 2023.

[49] S. Gollapudi, N. Karia, V. Sivashankar, R. Krishnaswamy, N. Begwani, S. Raz, Y. Lin, Y. Zhang, N. Mahapatro, P. Srinivasan, *et al.*, "Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 3406–3416.

[50] M. Wang, L. Lv, X. Xu, Y. Wang, Q. Yue, and J. Ni, "An efficient and robust framework for approximate nearest neighbor search with attribute constraint," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[51] J. Gao and C. Long, "High-dimensional approximate nearest neighbor search: With reliable and efficient distance comparison operations," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–27, 2023.

[52] Y. Chen, R. Zheng, Q. Chen, S. Xu, Q. Zhang, X. Wu, W. Han, H. Yuan, M. Li, Y. Wang, *et al.*, "Onesparse: A unified system for multi-index vector search," in *Companion Proceedings of the ACM on Web Conference 2024*, 2024, pp. 393–402.

[53] G. Gupta, A. Rao, T. Mai, R. A. Rossi, X. Chen, S. Mitra, and A. Shrivastava, "Near neighbor search for constraint queries," in *2023 IEEE International Conference on Big Data (BigData)*, IEEE, 2023, pp. 1707–1715.

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

[55] F. Groh, L. Ruppert, P. Wieschollek, and H. P. A. Lensch, "GGNN: graph-based GPU nearest neighbor search," *CoRR*, vol. abs/1912.01059, 2019. arXiv: 1912.01059. URL: http://arxiv.org/abs/1912.01059.

[56] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1033–1044. DOI: 10.1109/ICDE48307.2020.00094.

[57] W. Chen, J. Chen, F. Zou, Y.-F. Li, P. Lu, and W. Zhao, "Robustiq: A robust ann search method for billion-scale similarity search on gpus," in *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, ser. ICMR '19, Ottawa ON, Canada: Association for Computing Machinery, 2019, pp. 132–140, ISBN: 9781450367653. DOI: 10.1145/3323873.3325018. URL: https://doi.org/10.1145/3323873.3325018.

[58] B. H. Meyer, A. T. R. Pozo, and W. M. N. Zola, "Warp-centric k-nearest neighbor graphs construction on gpu," *50th International Conference on Parallel Processing Workshop*, 2021. URL: https://api.semanticscholar.org/CorpusID:237609458.

[59] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie, "Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment," *Proceedings of the ACM on Management of Data*, vol. 2, no. 1, pp. 1–27, 2024.

[60] Y. Pan, J. Sun, and H. Yu, "Lm-diskann: Low memory footprint in disk-native dynamic graph-based ann indexing," in *2023 IEEE International Conference on Big Data (BigData)*, IEEE, 2023, pp. 5987–5996.

[61] K. Tatsuno, D. Miyashita, T. Ikeda, K. Ishiyama, K. Sumiyoshi, and J. Deguchi, "Aisaq: All-in-storage anns with product quantization for dram-free information retrieval," *arXiv preprint arXiv:2404.06004*, 2024.

[62] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, *et al.*, "Manu: A cloud native vector database management system," *arXiv preprint arXiv:2206.13843*, 2022.

[63] Y. Su, Y. Sun, M. Zhang, and J. Wang, "Vexless: A serverless vector data management system using cloud functions," in *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 2024.

[64] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "Bridging software-hardware for cxl memory disaggregation in billion-scale nearest neighbor search," *ACM Transactions on Storage*, 2024.

[65] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "{Cxl-anns}:{software-hardware} collaborative memory disaggregation and computation for {billion-scale} approximate nearest neighbor search," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 585–600.

[66] Y. Cao, J. Liu, H. Qi, J. Gui, K. Li, J. Ye, and C. Liu, "Scalable distributed hashing for approximate nearest neighbor search," *IEEE Transactions on Image Processing*, vol. 31, pp. 472–484, 2021.

[67] R. Fathi, A. R. Molla, and G. Pandurangan, "Efficient distributed algorithms for the k-nearest neighbors problem," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 527–529.

[68] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," *Proceeding VLDB '99 Proceedings of the 25th International Conference on Very Large Data Bases*, vol. 99, May 2000.

[69] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979, ISSN: 00359254, 14679876. URL: http://www.jstor.org/stable/2346830 (visited on 05/24/2024).

[70] Rapidsai, *Rapidsai/raft: Raft contains fundamental widely-used algorithms and primitives for data science, graph and machine learning.* 2022. URL: https://github.com/rapidsai/raft.

[71] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[72] R. Wang and D. Deng, “Deltapq: Lossless product quantization code compression for high dimensional similarity search,” *Proceedings of the VLDB Endowment*, vol. 13, no. 13, pp. 3603–3616, 2020.

[73] W. Dong, C. Moses, and K. Li, “Efficient k-nearest neighbor graph construction for generic similarity measures,” in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW ’11, Hyderabad, India: Association for Computing Machinery, 2011, pp. 577–586, ISBN: 9781450306324. DOI: 10.1145/1963405.1963487. URL: https://doi.org/10.1145/1963405.1963487.

[74] C. Zuo and D. Deng, “Arkgraph: All-range approximate k-nearest-neighbor graph,” *Proceedings of the VLDB Endowment*, vol. 16, no. 10, pp. 2645–2658, 2023.

[75] C. Zuo, M. Qiao, W. Zhou, F. Li, and D. Deng, “Serf: Segment graph for range-filtering approximate nearest neighbor search,” *Proceedings of the ACM on Management of Data*, vol. 2, no. 1, pp. 1–26, 2024.

[76] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang, *et al.*, “Spfresh: Incremental in-place update for billion-scale vector search,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 545–561.

[77] M. D. Manohar, Z. Shen, G. E. Blelloch, L. Dhulipala, Y. Gu, H. V. Simhadri, and Y. Sun, *Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms*, 2024. arXiv: 2305.04359 [cs.IR].

[78] C. Li, M. Zhang, D. G. Andersen, and Y. He, “Improving approximate nearest neighbor search through learned adaptive early termination,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20, Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2539–2554, ISBN: 9781450367356. DOI: 10.1145/3318464.3380600. URL: https://doi.org/10.1145/3318464.3380600.

[79] K. Yang, H. Wang, B. Xu, W. Wang, Y. Xiao, M. Du, and J. Zhou, “Tao: A learning framework for adaptive nearest neighbor search using static features only,” *arXiv preprint arXiv:2110.00696*, 2021.

[80] Y. Dong, P. Indyk, I. Razenshteyn, and T. Wagner, “Learning space partitions for nearest neighbor search,” *arXiv preprint arXiv:1901.08544*, 2019.

[81] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004. URL: https://api.semanticscholar.org/CorpusID:174065.

[82] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” *CoRR*, vol. abs/1409.4842, 2014. arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.