Helping Out a Neighbor

A Study of Partition-based Algorithms for
k-Approximate Nearest Neighbor Search

Malte Helin Johnsen

A thesis presented for the degree of
Master of Science in Software Design

Supervised by Martin Aumüller

IT UNIVERSITY OF COPENHAGEN

Denmark
3rd of June, 2024

Abstract

The k-Approximate Nearest Neighbor Search problem is a fundamental algorithmic problem that has applications in many domains of computer science. At a high level, an algorithmic solution to this problem is composed of an index structure and a search strategy. In recent years novel search strategies have been proposed for tree-based index structures, which have been shown to improve the performance over the standard Lookup Search strategy. An experimental evaluation of these search strategies for locality-sensitive hashing-based index structures has remained unexplored.

This thesis introduces a novel search strategy, the Quick-select Natural Classifier Select, and experimentally evaluates this and three other search strategies across tree-based and locality-sensitive hashing-based index structures on datasets commonly used for benchmarking k-ANNS algorithms. The results demonstrate that novel search strategies outperform the standard Lookup Search strategy when paired with the two locality-sensitive hashing-based index structures, LSH and C2LSH. For the LSH index structure, the Natural Classifier Search strategy achieves a performance increase of 145% on the Fashion-MNIST dataset, while the Voting Search strategy achieves a performance increase of 65% on the SIFT dataset.

Additionally, the experimental evaluation shows that the Natural Classifier Search strategy is outperformed by the Voting Search strategy for both the LSH and RP-Forest index structures on the SIFT and GloVe-100 datasets. Finally, the LSH and RP-Forest index structures with Voting Search achieve the best performance of the evaluated algorithms on the GloVe-100 dataset.

Contents

1 Introduction **1**
- 1.1 Research Question . . . 1
- 1.2 Thesis Structure . . . 2
- 1.3 Source Code . . . 2

2 Problem Setting and Definitions **3**
- 2.1 Partition-based k-ANNS Framework . . . 4

3 Index Structures **7**
- 3.1 Locality Sensitive Hashing . . . 7
  - 3.1.1 Hashing Scheme . . . 8
  - 3.1.2 LSH for k-ANNS on the Unit Sphere . . . 10
  - 3.1.3 LSH Implementation Details . . . 11
  - 3.1.4 LSH Build Parameters . . . 11
  - 3.1.5 Asymptotic Analysis of the LSH Index Structure . . . 12
- 3.2 Collision Counting Locality Sensitive Hashing . . . 14
  - 3.2.1 Virtual Rehashing . . . 14
  - 3.2.2 Querying C2LSH . . . 16
  - 3.2.3 C2LSH Implementation Details . . . 16
  - 3.2.4 C2LSH Build Parameters . . . 19
  - 3.2.5 Asymptotic Analysis of the C2LSH Index Structure . . . 20
- 3.3 Random k-Dimensional Forest . . . 20
  - 3.3.1 RKD-Tree Build Parameters . . . 21
  - 3.3.2 Asymptotic Analysis of the RKD-Forest Index Structure . . . 22
- 3.4 Random Projection Forest . . . 23
  - 3.4.1 RP-Tree Build Parameters . . . 24
  - 3.4.2 Asymptotic Analysis of the RP-Forest Index Structure . . . 25

4 Search Strategies **26**
- 4.1 Lookup Search . . . 26
- 4.2 Voting Search . . . 27
  - 4.2.1 Voting Search Implementation Details . . . 28
  - 4.2.2 Setting Parameters for Voting Search . . . 29
- 4.3 Natural Classifier Search . . . 30
  - 4.3.1 Natural Classifier Search Implementation Details . . . 32
  - 4.3.2 Setting Parameters for Natural Classifier Search . . . 32
- 4.4 Quick-select Natural Classifier Search . . . 34
  - 4.4.1 Setting Parameters for Quick-Select Natural Classifier Search 35
  - 4.4.2 Quick-select Natural Classifier Search Implementation Details . . . 37

5 Build Parameter Assessment Study **39**

6 Experiments **43**
6.1 Algorithm Parameters . . . 43
6.2 Performance Metrics . . . 44
6.3 Input Instances . . . 45
6.4 Experiment Environment and Parallelization . . . 46
6.5 Experiment Design: Round 1 . . . 47
6.5.1 Algorithm Parameters - Fashion-MNIST . . . 47
6.5.2 Algorithm Parameters - SIFT . . . 47
6.6 Experiment Design: Round 2 . . . 48

7 Experiment Results **49**
7.1 Fashion-MNIST Results . . . 49
7.1.1 Parameter Evaluation - Fashion-MNIST . . . 50
7.2 SIFT Results . . . 52
7.2.1 Parameter Evaluation - SIFT . . . 55
7.3 GloVe Results . . . 56
7.4 Algorithm Recommendations . . . 59

8 Conclusion **61**
8.1 Future Work . . . 62

A Implementation Appendix **66**

B Results Appendix **67**

1 Introduction

The k-Nearest Neighbor Search (k-NNS) problem is the proximity problem of finding the k points nearest to a query point q in a given corpus of points. The corpus of points can be thought of as a database of objects, in which each object is represented as a d-dimensional vector. The query point q is then as an external object, also represented as a d-dimensional vector, for which the goal is to identify the k most similar objects within the database. This fundamental problem has important applications in many domains such as computer vision, machine learning, multimedia database querying, contextual advertising and many more. Given the fact that these use cases typically rely on large datasets of high dimensionality, it is often necessary to sacrifice the exact nature of the solution due to the phenomenon called the "Curse of Dimensionality" [KE11]. This approximate variant of the problem is called k-Approximate Nearest Neighbor Search (k-ANNS). Partition-based algorithmic solutions to k-ANNS rely on searching one or more index structures which induce a partition of the elements in the corpus, in order to determine a smaller candidate set of points on which to carry out an exact nearest neighbor search. Two common categories of partition-based index structures for algorithms that solve k-ANNS are partition-tree index structures and locality-sensitive hashing-based index structures [JAL23].

The way in which an index structure is utilized to build the candidate set is referred to as a *search strategy*, while the combination of an index structure and a search strategy is referred to as a k-ANNS algorithm. In [HJR22a] the authors propose a novel search strategy called the Natural Classifier Search. The authors experimentally demonstrate that the Natural Classifier Search strategy outperforms the two previously proposed strategies, Lookup Search and Voting Search, for tree-based index structures across multiple datasets [HJR22a]. The scientific literature does not yet present any evaluation of the Voting or Natural Classifier Search strategies being coupled with locality-sensitive hashing-based index structures for k-ANNS. It is therefore undocumented if the performance of algorithms based on locality-sensitive hashing-based index structures could benefit from being coupled with these search strategies.

1.1 Research Question

To uncover whether locality-sensitive hashing-based index structures for k-ANNS benefit from these novel search strategies, and to identify an overall best performing algorithm for k-ANNS, the following research question is posed:

> *What are the performance characteristics of classic and novel search strategies when coupled with both tree-based and locality-sensitive hashing-based index structures to produce an algorithm for k-Approximate Nearest Neighbor Search?*

1.2 Thesis Structure

This thesis project aims to answer the research question by experimentally evaluating all algorithms corresponding to the combinations of four different index structures, each coupled with four search strategies. In order to carry out the experimental evaluation to conclude on the research question, the thesis will follow the structure:

- In section 2 define the problem setting of the k-Approximate Nearest Neighbor problem and a framework for describing algorithmic solutions to the problem.
- In section 3 implement and in detail describe the theoretical underpinnings of two common locality-sensitive hashing-based index structures for k-ANNS; LSH and Collision Counting LSH as well as the two tree-based index structures; RP-Forest and RKD-Forest.
- In section 4 implement and in detail describe the search strategies Lookup Search, Voting Search, Natural Classifier Search and introduce a novel search strategy; Quick-select Natural Classifier Search.
- Throughout section 4 and in section 5 describe high-level strategies and techniques for configuring index structures and search strategies for k-ANNS.
- In section 6 describe the experimental design of a suite of experiments which aim to evaluate the performance characteristics of each k-ANNS algorithm corresponding to an index structure and search strategy combination.
- In section 7 present, analyze and discuss the results of said experiments and provide recommendations for the choice of algorithms for k-ANNS.
- In section 8 based on the experimental findings conclude on the research question.

1.3 Source Code

A substantial part of the work of this thesis lies in producing a clear implementation of each index structure and search strategy in a well-organized code base. The reader is encouraged to supplement the reading of this thesis with the relevant source code, as it can provide an additional tool for understanding each index structure and search strategy. All source code is implemented in Java 11 using the Gradle build tool. All source code related to the discussed algorithms and experimental evaluation is available at: https://github.com/KarateMogens/ANNSearch. The reader is also encouraged to run local experiments of the described algorithms on their own device using the compiled version of the source code with relevant datasets available at: https://github.com/KarateMogens/ANNSearchBuild.

2 Problem Setting and Definitions

The k-Nearest Neighbor Search problem (abbreviated kNNS from this point forward) is the proximity problem of finding the set of k points closest (or most similar) to a query point q in a given corpus of points. Formally, let $\mathbf{X} \subseteq \mathbb{R}^d$ be a corpus of data objects where each object is represented as a point $x \in \mathbb{R}^d$. Given a query vector, $q \in \mathbb{R}^d$, let $\text{dist}(q, x)$ denote some distance function calculating a distance between q and x. That is, $\text{dist} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. Given an integer k where $1 \leq k \leq |\mathbf{X}|$, the set of points in $\mathbf{X}$ closest to q is denoted $k\text{NN}(q)$ and is defined:

$$k\text{NN}(q) := \underset{x \in \mathbf{X}}{k\text{-argmin}}(\text{dist}(q, x))$$

where k-argmin$(f(x))$ is defined as the set of k minimizing arguments to the function $f(x)$ [HJR22a]. Note that this definition assumes that ties are broken evenly at random and thus $|k\text{NN}(q)| = k$.

The naïve solution to solving kNNS involves calculating $\text{dist}(q, x)$ for all $x \in \mathbf{X}$, sorting the corpus points by distance to q, and selecting the k closest points. This is typically achieved by calculating $\text{dist}(q, x)$ for all $x \in \mathbf{X}$ one at a time, adding the key-value pair $(\text{dist}(q, x), x)$ to a minimum-oriented priority queue and subsequently retrieving k points from the queue [GFFN12]. This approach will be referred to as the *brute-force approach* throughout the paper. A common approach for efficient solutions to kNNS involves partitioning the points in $\mathbf{X}$ according to some datastructure and selectively searching partition elements to build a set of candidate points $\mathbf{C}(q) \subseteq \mathbf{X}$ on which to carry out a brute-force search [Ben75, SAH08, DS15, AC23]. This is done to minimize the number of distance computations, which entails making $\mathbf{C}(q)$ as small as possible while still retaining accuracy. The efficiency of these solutions is known to suffer greatly in higher dimensionality due to the phenomenon called the "Curse of Dimensionality" [KE11]. Under high-dimensionality conditions, $\mathbf{C}(q)$ typically becomes so large that the brute-force search is carried out on a set of points that is almost as large as $\mathbf{X}$, but with the additional, non-trivial computation of searching a partitioning datastructure and building $\mathbf{C}(q)$. It has been shown that the performance of kNNS for many such datastructures degrades to that of a brute-force solution with a dimensionality as low as $d = 10$ [WSB98].

Therefore, in many applications, it is preferable to solve an approximate variant of the k-NNS problem called k-Approximate Nearest Neighbor Search (abbreviated k-ANNS), which will be the focus of this paper. For k-ANNS the goal is to find an approximate solution to k-NNS to increase computational efficiency. The solution to k-ANNS is equally given by a set of points $k\text{ANN}(q) \subseteq \mathbf{X}$, where each point may or may not belong to $k\text{NN}(q)$ and where $|k\text{ANN}(q)| \leq k$. The quality of a given solution, $k\text{ANN}(q)$, is typically measured by a recall value that ranges from 0 to 1, inclusive [ABF18]. For this paper, the recall of a solution $k\text{ANN}(q)$, is defined as:

$$\frac{|\{x \in k\text{ANN}(q) \mid \text{dist}(q, x) \leq \text{dist}(q, x_k)\}|}{k}$$

where x_k denotes the point which is the k-th most distant neighbor to q in $k\text{NN}(q)$:

$$x_k = \underset{x \in k\text{NN}(q)}{\text{argmax}} \left(\text{dist}(q, x)\right)$$

While the quality of a $k\text{ANN}(q)$ solution is typically measured by the recall value, the performance of an algorithm for solving the k-ANNS problem is typically evaluated by its average recall-query time trade-off [ABF18, LZS+16]. This and other performance metrics will be covered in greater detail in section 6.

Finally, to clear up any potential confusion, it is important to note that this paper will not focus on the related similarity search problem called the c-approximate Nearest Neighbor problem, in which the solution is a point $x \in \mathbf{X}$ for which $\text{dist}(q, x)$ is at most c times the distance to the exact nearest neighbor of q, or its' k-variant in which the solution is a set of k points where each point is a c-approximation of its respective member in $k\text{NN}(q)$ [GFFN12, DIIM04].

2.1 Partition-based k-ANNS Framework

A common category of approaches to solving k-ANNS is the so-called partition-based approaches, which rely on searching multiple partitions of $\mathbb{R}^d$, where $\mathbf{X} \subseteq \mathbb{R}^d$. A partition P is defined as a set $\{P_1, P_2, ...\}$ where each partition element P_i is a mutually disjoint subset of $\mathbb{R}^d$, such that $P_i \cap P_j = \emptyset$ whenever $i \neq j$. Furthermore it must also hold that $\bigcup_{P_i \in P} = \mathbb{R}^d$. Standard partition-based solutions to k-ANNS and specifically the solutions explored in this paper, follow a common multi-step approach which is described in [AC23]. At the highest level of abstraction, these approaches can be divided into a *build-phase* and a *search-phase*. In the build phase, an indexing structure is constructed based on a set of build parameters. The index structure is typically an ensemble of datastructures $L = \{D_1, ..., D_{|L|}\}$, where each datastructure D_i, induces a partition on $\mathbb{R}^d$. Once built, a single datastructure $D_i \in L$ supports the query operation $\text{query}(D_i, q)$, for a point $q \in \mathbb{R}^d$, which returns a set of points $Q_i \subseteq \mathbf{X}$ belonging to the same partition element as q.

Once the build phase is completed, the index structure L can be used in a search phase. In the search phase, given a set of search parameters and a query vector $q \in \mathbb{R}^d$, the index structure is searched to return a solution, $k\text{-ANN}(q)$. In this phase, each datastructure in the index is queried and the returned sets of points are used to construct a preliminary candidate set $\mathbf{C}'(q)$. More formally:

$$\mathbf{C}'(q) = \bigcup_{D_i \in L} \text{query}(D_i, q)$$

Based on a score function, which assigns a score to each point in $\mathbf{C}'(q)$, and a set of supplied search parameters, in the candidate refinement process $\mathbf{C}'(q)$ is trimmed to a set of candidate points $\mathbf{C}(q) \subseteq \mathbf{C}'(q)$. In the final re-ranking step, a brute-force search is carried out considering only points in $\mathbf{C}(q)$ rather

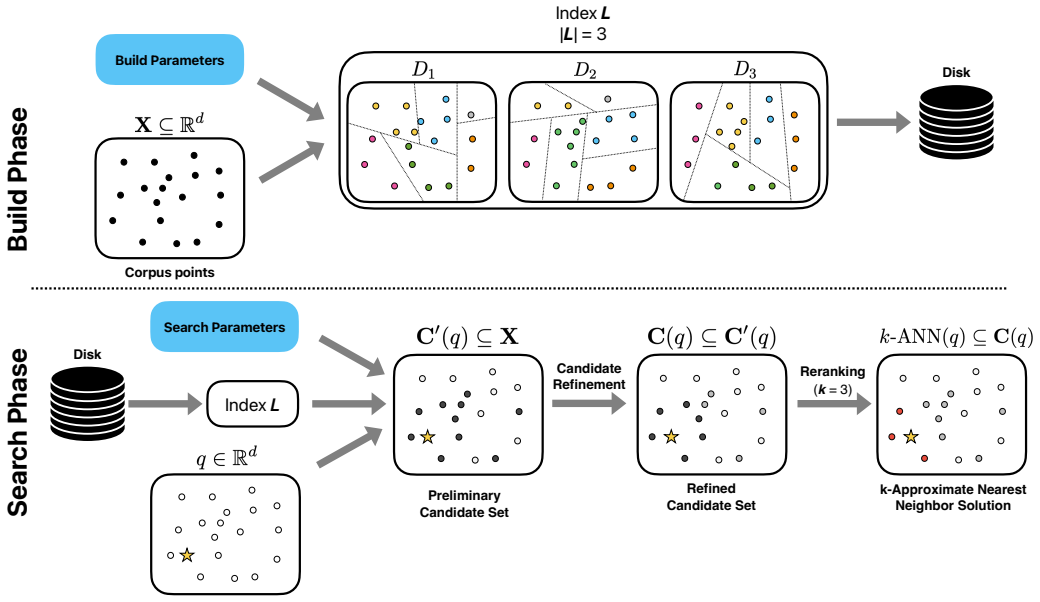

Figure 1: Overview of partition-based k-ANNS. Partitions and candidate sets are random and not indicative of any specific index structure or search strategy. The figure is modified from [AC23].

than the full set of corpus points, to determine the final set $k\text{ANN}(q)$. Therefore any point $x \in \mathbf{X}$ belongs to $k\text{ANN}(q)$ iff. $x \in \mathbf{C}(q)$. The combination of score function and refinement condition will be referred to as a *search strategy* from this point. The combination of an index structure and a search strategy will be referred to as a *k-ANNS algorithm*. See figure 1 for an overview of the partition-based k-ANNS framework.

The build phase is often computationally expensive and is not dependent on a particular set of query vectors. Therefore, in practice, it is typically not feasible to construct an index structure when a search result is needed. Instead, the index structure is computed prior to the search phase and is stored on persistent storage to be fetched into memory when needed. The feasibility and relevance of an index structure is dependent on multiple factors, most importantly the index construction time as well as the spatial requirement, as it is vital that the structure fits onto main memory during the search phase [AC23]. Furthermore, an index structure should support efficient query operations during the search phase, meaning that the query operation is both fast to execute and returns corpus points which are relevant to the solution.

The most computationally expensive aspect of the search phase for a high-dimensional dataset will typically be the distance computations involved in the

re-ranking step. Therefore it is imperative that $\mathbf{C}(q)$ does not exceed a size that makes the re-ranking step infeasibly slow in relation to the required recall quality. However, it is also possible that the process of building such a candidate set carries a disproportionately large computational expense compared to re-ranking a larger $\mathbf{C}(q)$. Thus, the overarching goal of the search strategy is to strike a balance between these two desired properties in order to efficiently create a qualified candidate set $\mathbf{C}(q)$, which minimizes the number of points that are likely to be distant from and maximizes the number of points which are likely to be close to q.

3 Index Structures

In the section that follows, I will account for the four different index structures that will be subject to experimental analysis in section 6. The four indxwex structures to be examined are:

- Locality Sensitive Hashing (LSH)
- Collision-counting Locality Sensitive Hashing (C2LSH)
- Random Projection Forest (RP-Forest)
- Random k-dimensional Forest (RKD-Forest)

For each index structure, I will first provide an overview of the index structure and its' relevant theoretical background, followed by select implementation details. Afterward, I will provide an overview of the related build parameters and a high-level discussion of their effect on the characteristics of the partitions induced by the index structure. Finally, I will provide a short analysis of the asymptotic running time and space complexity associated with each index structure.

Note that the RKD-Forest and RP-Forest index structure have been largely covered in [JAL23], but the framework in which they are described has been adapted to fit the framework for describing k-ANNS algorithms presented above.

3.1 Locality Sensitive Hashing

Relevant source code: `HashTable.java, HashFunction.java, AngHashTable.java, BinaryHash.java`

The first index structure that will be detailed was proposed in the seminal work [DIIM04]. The index structure is an ensemble structure in which each datastructure indexes $\mathbf{X}$ based on locality-sensitive hashing. At the risk of causing confusion, this index structure will simply be referred to as LSH throughout this thesis. A single datastructure D_i in such an LSH index structure L consists of a hash table T_i, and a compound hash function g_i which is used to index points to buckets in T_i. The compound hash function g_i consists of a K-tuple (NB: $k \neq K$) of hash functions $g_i = (h_1, ..., h_K)$, where each h_j is drawn from a family of locality-sensitive hash functions, which is described in greater detail in section 3.1.1. Hash functions can be locality-sensitive in the sense, that each hash function $h_j \in g_i$ is more likely to hash near points to the same value than distant points. To build each datastructure $D_i \in L$, all points $x \in \mathbf{X}$ are indexed according to the compound hash function, such that $g_i(x) = (h_1(x), ..., h_K(x))$, and are inserted in the corresponding bucket in the hash table T_i. Note that the value $g_i(x)$ corresponds to the concatenation of the K hash functions. A datastructure $D_i \in L$ is subsequently queried by hashing a query vector q according to the same compound hash function ($g_i(q)$) and returning the set of

corpus points Q_i, which belong to the same bucket as q in the table T_i. That is:

$$Q_i = \{x \in \mathbf{X} \mid g_i(x) = g_i(q)\}$$

The complete index structure is finally comprised of $|L|$ such datastructures - that is $L = \{D_1, ..., D_{|L|}\}$ - where each D_i consists of a hash table T_i and a compound hash function g_i.

3.1.1 Hashing Scheme

The first family of locality-sensitive hash functions for c-ANNS was described in [IM98] for the special case where $\mathbf{X}$ lives in the Hamming Space $\{0,1\}^d$. It is possible to extend the algorithm proposed in this work to support points which live in the Euclidean space, however, it complicates the algorithm considerably and comes with a high toll to both accuracy and query time as it relies on embedding the Euclidean space into the Hamming Space [DIIM04].

The family of locality-sensitive hash functions used in the LSH index structure described in this thesis was introduced in [DIIM04]. Rather than relying on embedding, it instead works directly in the Euclidean space and builds on the properties of so-called p-stable distributions. Although the authors primarily focus their efforts on LSH for the sake of solving the c-ANNS problem, the approach is still highly relevant for solutions to k-ANNS. The authors define the family of locality-sensitive hash functions from which each hash function is drawn:

$$h(x) = \left\lfloor \frac{\vec{a} \cdot x + b}{r} \right\rfloor$$

where $x, \vec{a} \in \mathbb{R}^d$, the components of $\vec{a}$ are i.i.d values drawn from the normal distribution $\mathcal{N}(0,1)$, $r \in \mathbb{R}$ and b is uniformly drawn from the range $[0, r]$ [DIIM04]. The intuition behind this family of locality-sensitive hash functions is that through the operation $\vec{a} \cdot x$, any vector x is projected onto the real line and shifted by a value of b. By the division of the value r combined with the floor function, the real line is cut into equi-width segments which each map to a value in $\mathcal{Z}$, corresponding to a bucket-ID. A vector x can then be hashed to an integer value according to the hash function, to identify the bucket which it belongs to. In this family of hash functions, r is a free tuning parameter that determines the bucket width. Geometrically, such a hash function can be interpreted as a partitioning of $\mathbb{R}^d$ that is based on slicing $\mathbb{R}^d$ with evenly spaced apart hyper-planes, which are orthogonal to the random vector $\vec{a}$. In this interpretation, each partition element maps to a bucket value. A visualization of such a partition in two dimensions is illustrated in figure 2.

By indexing points according to the concatenation of multiple such hash functions, a much more complex partition is achieved where each partition element corresponds to the intersection of K different partition elements of $\mathbb{R}^d$ (see figure 3).

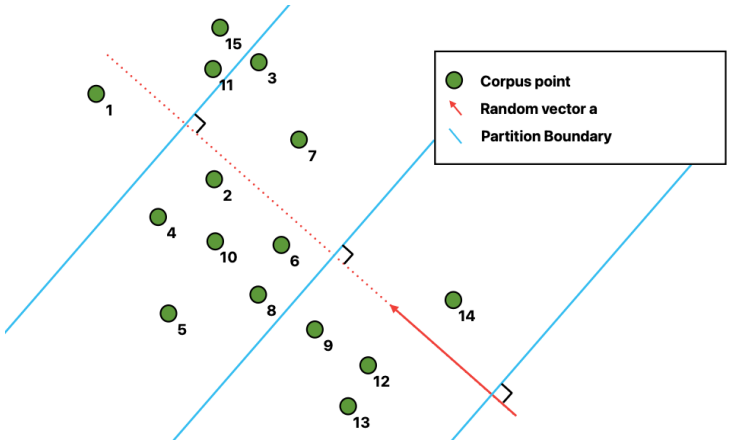

Figure 2: A single locality-sensitive hash function, h

The reason that this family of hash functions is so relevant for the LSH index structure is due to the relationship between Gaussian distributions and the Euclidean distance detailed in lemma 3.1 [DIIM04].

Lemma 3.1 *Let $[x]_j$ denote the j-th component of the vector x. Given two points $x, y \in \mathbb{R}^d$, their Euclidean distance $||x-y||_2$ and a random vector $\vec{a}$ where each component $[a]_j \sim \mathcal{N}(0,1)$. It follows that $(\vec{a}\cdot x - \vec{a}\cdot y) \sim \mathcal{N}(0, (||x-y||_2)^2)$.*

Lemma 3.1 detailing the distribution of the value $\vec{a}\cdot x - \vec{a}\cdot y$ can be deducted in the following way:

$$\vec{a}\cdot x - \vec{a}\cdot y = \sum_{j=1}^{d}([x]_j - [y]_j)\cdot[\vec{a}]_j$$

Given that $[a]_j$ is a Gaussian distributed as $\mathcal{N}(0,1)$, then $([x]_j - [y]_j)\cdot[\vec{a}]_j$ is a Gaussian distributed as $\mathcal{N}(0, ([x]_j - [y]_j)^2)$. Finally since each $([x]_j - [y]_j)\cdot[\vec{a}]_j$ is a Gaussian, the sum of these is also a Gaussian which is distributed as:

$$\mathcal{N}(0, \sum_{j=1}^{d}([x]_j - [y]_j)^2) = \mathcal{N}(0, (||x-y||_2)^2)$$

Therefore the difference between $\vec{a}\cdot x$ and $\vec{a}\cdot y$ is a Gaussian which has a mean 0 and a variance equal to the Euclidean square distance between the two points [CG]. In summation, two points x and y are more likely to have similar values of their dot product with $\vec{a}$ than distant points are.

Therefore, given lemma 3.1 we find that $\vec{a}\cdot x$ for some $x \in \mathbf{X}$ is a locality-sensitive projection onto the real line. The final step which leverages this fact, is the remaining operators of the hash function $\lfloor \frac{\vec{a}\cdot x + b}{r} \rfloor$, which divide $\mathbb{R}$ into equi-width buckets. It then becomes clear that the described LSH family can

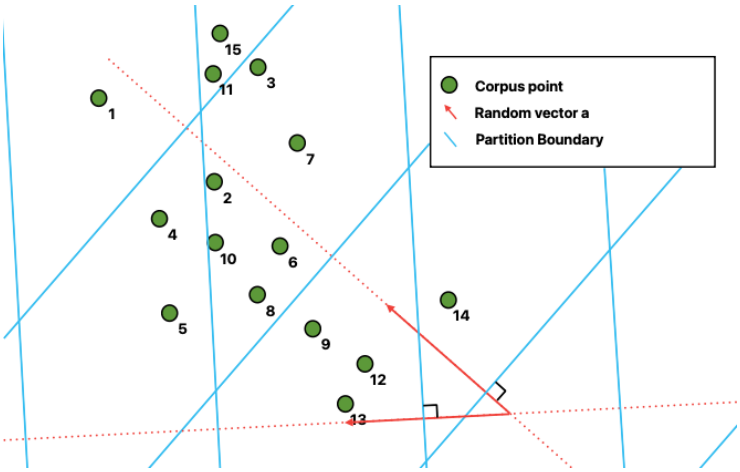

Figure 3: Partition induced by a compound hash function g where $K = 2$

be used to create partitions of $\mathbf{X}$ where close points are more likely to belong to the same partition element than distant points [DIIM04].

3.1.2 LSH for k-ANNS on the Unit Sphere

A common and important special case of k-ANNS that has many practical applications, is k-ANNS under the angular distance metric. That is, k-ANNS where $\text{dist}(q, x)$ for $q, x \in \mathbf{X}$ is given by:

$$\text{dist}(q, x) := \frac{\arccos\left(\frac{q \cdot x}{|q||x|}\right)}{\pi}$$

and where $|x|$ denotes the magnitude of x. A typical way to solve this special case is to solve a proxy problem under the Euclidean distance metric, where all points $x \in \mathbf{X}$ and q are normalized to unit length. Normalizing all points to unit length entails that all points lie on the unit sphere which has its center in the origin. This transformation is carried out because the ordering of Euclidean distances on the unit sphere has the same ordering as the angular distances of the non-normalized $\mathbf{X}$ and q. Given this transformation, it is advantageous to draw each $h_j \in g_i$, from the following simplified family of hash functions, which was proposed in [Cha02]:

$$h(x) = \begin{cases} 1 & \text{if } x \cdot \vec{a} \geq 0 \\ 0 & \text{if } x \cdot \vec{a} < 0 \end{cases}$$

where $\vec{a}$ is a random vector constructed in the way described above. Thus each hash function $h \in g$ becomes a binary hash function which splits the unit sphere into two equal-sized partitions according to a hyper-plane that intersects

the origin. The above-described transformation and family of hash functions are used in the experiments which follow the angular distance metric.

3.1.3 LSH Implementation Details

Given that most of the implementation of the LSH datastructure is relatively close to the description provided above, I will highlight a specific implementation detail regarding a hashing scheme used to hash values given by the compound hash function g_i and therefore also the bucket-IDs in each hash table T_i. A naïve approach would be to use a standard implementation of a symbol table for T_i (such as the `HashMap` class), using keys simply corresponding to an `array` of `int`s with size K , where the value stored at index $j-1$ is computed from $h_j \in g_i$. However, this approach presents a problem, which is that two identical arrays of `int`, may be considered two separate buckets due to Java's `HashMap` implementation relying on both calls to the object methods `.hashCode()` and subsequent calls to `.equals()` or `==`, the latter of which relies on the memory address of an object (see appendix A, listing 1).

To avoid this issue, the implementation of the LSH datastructure which is used throughout this thesis for the Euclidean distance metric, utilizes reference hashing as described in [PP16] to hash each array of K `int` values into a single `long` value. In practice, this is done by using an associated universal hash function. Given some prime P, an K-tuple of `int` values: $(h_1(x), .., h_K(x))$, and a K-tuple of random integers $(b_1, ..., b_k)$ where each b_i is drawn uniformly at random from $[0, P-1]$, the universal hash function used is:

$$p = b_1 \cdot h_1(x) \ mod \ P + b_2 \cdot h_2(x) \ mod \ P +, ..., b_K \cdot h_K(x) \ mod \ P$$

where p is the `long` value used for indexing some point x. The prime P must be sufficiently large, to avoid unwanted collisions of partition elements corresponding to different K-tuples. It is recommended by [PP16] to use a value of $P > |\mathbf{X}|^2$. In the implementation of the LSH index structure of this thesis, the value P is chosen as the Mersenne prime $2^{61} - 1$. This value can safely be used for datasets as large as 10^9 corpus points with minimal probability of unwanted collisions. Thus, the final datastructure which implements each hash table T_i is a `HashMap` of the key-value pair `<Long, LinkedList<Integer>>`, where the `LinkedList` datastructure is used to store the indices of all corpus points belonging to the bucket identified by its `long` value. For LSH under the angular distance metric in which the hash value computed from a compound hash function g_i is a K-tuple of binary values, such a reference hashing scheme can be completely avoided. For this compound hash function, the K-tuple can simply be transformed to a `long` value corresponding to the bits of the K-tuple using bit operations.

3.1.4 LSH Build Parameters

The LSH-based index structure under the Euclidean distance metric requires three build parameters, $K, r, |L|$ which each impact the likelihood that a cor-

pus point will belong to Q_i for a given q and datastructure D_i. It is vital to understand the impact of each build parameter, to construct a relevant index structure for k-ANNS over some dataset $\mathbf{X}$. [CG] provides an excellent overview of the three build parameters, which the following discussion is based on. First of all, as r is increased the collision probability of any point $x \in \mathbf{X}$ and q under a single hash function $h \in g_i$ increases. Intuitively this is because the bucket width prescribed by each h is increased. Given some specific value of r and the fact that the probability of collision is distance dependent, let this collision probability be denoted $p(\text{dist}(x, q))$. As the value of K is increased, the probability of x and q being hashed to the same bucket under a compound hash g_i function monotonically decreases, because the two points have to collide under all hash functions $(h_1, ..., h_K)$ in order to collide under g_i. Therefore the collision probability depends on K in the following way: $p(\text{dist}(x, q))^K$. This also means that as K increases, for a low value $p(\text{dist}(x, q))$ the collision probability decreases at a much faster rate than for closer points where $p(\text{dist}(x, q))$ is closer to 1. With this in mind, it becomes clear that K is an efficient parameter for the task of pruning distant outliers from q while retaining a lower risk of pruning closer candidates.

Finally the parameter $|L|$ increases the probability that x and q will collide under some g_i belonging to a datastructure $D_i \in L$. This is of course equivalent to x belonging to the preliminary candidate set $\mathbf{C}'(q)$, due to the fact that $\mathbf{C}'(q)$ is the union of points which collide with q under some $D_i \in L$:

$$\mathbf{C}'(q) = \bigcup_{i \in \{1,...,|L|\}} Q_i$$

That is, for a point to be included in $\mathbf{C}'(q)$, it simply has to collide with q in a single of the $|L|$ compound hash functions. Therefore, the final probability of x and q colliding under at least one datastructure and thus $x \in \mathbf{C}'(q)$ becomes:

$$1 - (1 - p(\text{dist}(x, q))^K)^L$$

Figure 4 showcases a small study of the impact of both K and L on the collision probability of two corpus points x_i and x_j for some distance to the query vector q and a fixed value of r. We have that $p(dist(x_i, q)) = 0.4$ and $p(dist(x_j, q)) = 0.8$ under a single hash function h. Note that the left plot showcases the collision probability of two points considering a single compound hash function g_i for increasing values of K, while the right plot showcases the collision probability under at least one g_i for various values of $|L|$. The right plot considers $|L|$ compound hash functions for which $K = 10$. The plots effectively highlight how the parameters K and L can be configured and combined to reduce the collision probability of distant points to q (illustrated by the point x_i) while retaining high collision probability for nearer points (x_j).

3.1.5 Asymptotic Analysis of the LSH Index Structure

In the following analysis, let n denote the size of the corpus set ($|\mathbf{X}|$). The LSH index structure contains $|L|$ datastructures, which each are composed of a hash

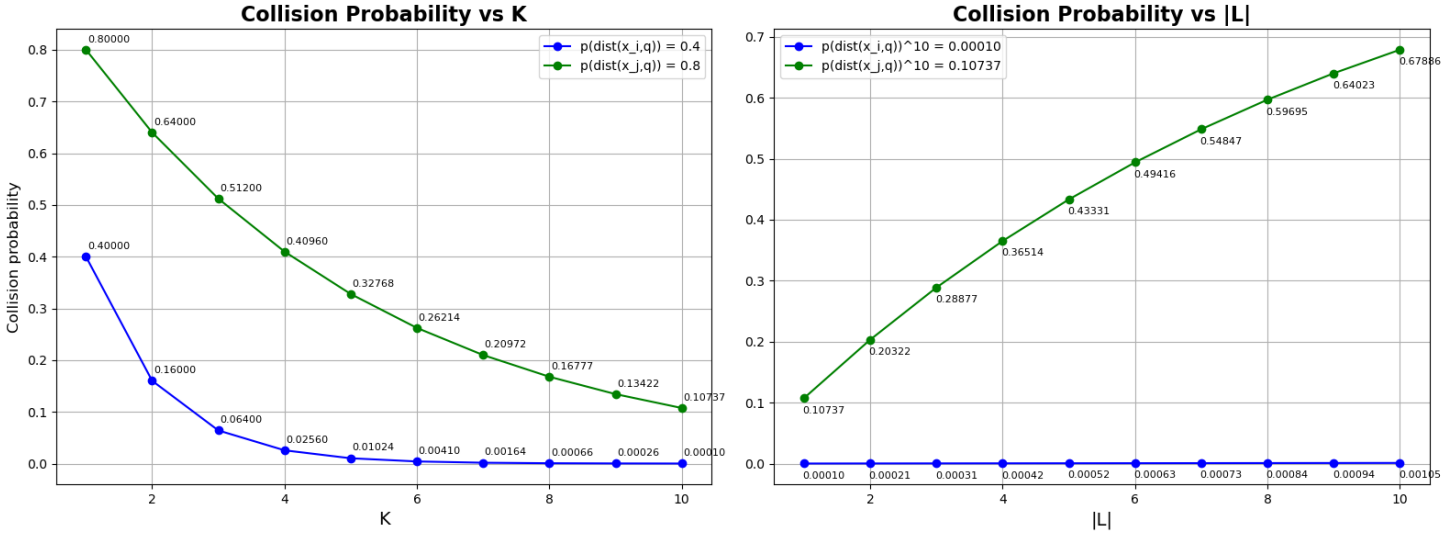

Figure 4: The collision probability of two points x_i and x_j for which $p(\text{dist}(x_i, q)) = 0.4$ and $p(\text{dist}(x_j, q)) = 0.8$. Left: The collision probability under a single g_i as a function of K. Right: The collision probability under at least one g_i for $K = 10$ as a function of $|L|$.

table (implemented as a `HashMap`) and a compound hash function consisting K hash functions which each store a d-dimensional vector ($\vec{a}$) and two `float` values (b and r). Given that the hash table stores just the indices of each corpus point, a single hash table has a space complexity of $\mathcal{O}(n)$ and the compound hash function has a space complexity of $\mathcal{O}(K \cdot d)$. Therefore each $D_i \in L$ has a space complexity of $\mathcal{O}(n + K \cdot d)$ and the LSH index structure as a whole has a space complexity of $\mathcal{O}(|L| \cdot (n + K \cdot d))$.

Constructing the K vectors used in a compound hash function g_i takes time proportional to their dimensionality d and the number of vectors K. Thus constructing the hash function g_i takes time $\mathcal{O}(K \cdot d)$. Inserting a single corpus point into a hash table T_i requires first computing the K independent hash values each requiring $\mathcal{O}(K \cdot d)$ time, then computing a subsequent reference hash which takes time $\mathcal{O}(K)$ and finally appending the point index to the `LinkedList` identified by the reference hash which takes time $\mathcal{O}(1)$. This leads to a total insertion time complexity of $\mathcal{O}(K \cdot d + K + 1) = \mathcal{O}(K \cdot d)$. This complexity rests on assumptions of the implementation of the `HashMap` class, namely uniform hashing leading to good load balancing and clever choices of the underlying array size leading to constant time insertion in the amortized sense. Thus, indexing all corpus points to build a single datastructure D_i has a time complexity of $\mathcal{O}(n \cdot K \cdot d)$. The full construction of the datastructure therefore has a time complexity of $\mathcal{O}(K \cdot d + n \cdot K \cdot d) = \mathcal{O}(n \cdot K \cdot d)$. The full index structure therefore has a build time complexity of $\mathcal{O}(|L| \cdot n \cdot K \cdot d)$

Finally, a single datastructure $D_i \in L$ supports the query operation query(D_i, q) with a time complexity of $\mathcal{O}(K \cdot d + K) = \mathcal{O}(K \cdot d)$ since to identify the bucket of q, q is hashed according to K hash functions each computing a dot-product in d dimensions and a reference hash value is subsequently computed. Again, this time complexity rests on an assumption of constant time retrieval from T_i.

3.2 Collision Counting Locality Sensitive Hashing

Relevant source code: `C2LSH.java`, `HashFunction.java`

The second index structure to be examined is called Collision Counting Locality Sensitive Hashing (abbreviated C2LSH) and was proposed in [GFFN12]. C2LSH differs from the other three index structures, mainly because it is not based on an ensemble of datastructures, but rather uses a single dynamic datastructure. That is for C2LSH, $L = \{D\}$. The datastructure used for C2LSH, just like the LSH index structure described above, also utilizes hash functions drawn from the locality-sensitive family:

$$h(x) = \left\lfloor \frac{\vec{a} \cdot x + b}{r} \right\rfloor$$

The datastructure for C2LSH differs from the datastructures used in LSH by instead of consisting of a compound hash function g and corresponding hash table T, the C2LSH datastructure maintains a set of K hash tables $\{T_1, ..., T_K\}$, which each use a single corresponding locality sensitive hash function from the set $\{h_1, ..., h_k\}$ to index corpus points. To construct the C2LSH datastructure, each corpus point $x \in \mathbf{X}$ is hashed one-by-one according to each hash function h_i and stored in the corresponding bucket in table T_i.

When the datastructure is built, it is supplied with two build parameters besides the value K, which are *partitionSize* and a threshold value t, where $1 \leq t \leq K$ and $1 \leq \textit{partitionSize} \leq |\mathbf{X}|$. When the datastructure is queried with some $q \in \mathbb{R}^d$ through the operation query(D, q), a set of points Q is returned which satisfies the two conditions; each $x \in Q$ collides with q under at least t *virtually rehashed* hash functions (detailed later) and $|Q| \geq \textit{partitionSize}$. In the sections that follow, it will be detailed how these conditions are satisfied in the C2LSH index structure.

3.2.1 Virtual Rehashing

Virtual rehashing is a technique where given a hash function h and a query point q, the datastructure is able to consider neighboring partition elements to the partition element corresponding to the bucket $h(q)$. This is achieved by rehashing the value $h(q)$ according to the following hash function:

$$H^R(q) = \left\lfloor \frac{h(q)}{R} \right\rfloor$$

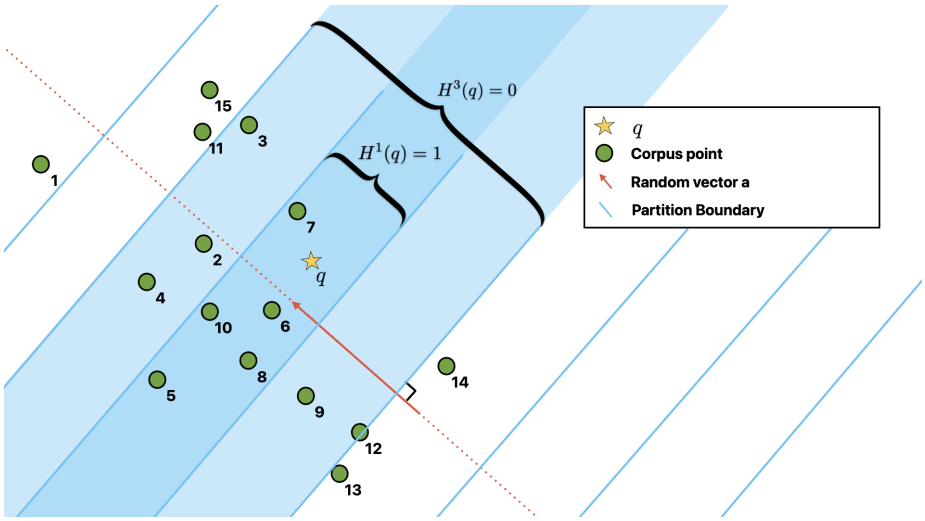

Figure 5: Partition elements corresponding to the level-1 bid $H^1(q)$ and the level-3 bid $H^3(q)$ for some point $q \in \mathbb{R}^d$

where R is some integer value, H^R is called the level-R hash function and $H^R(q)$ is called the level-R bid of q.

Theoretically, virtual rehashing is based on the fact that H^R preserves the locality-sensitive properties of the hash family h and at the same time that the partition elements corresponding to the buckets of H^R are effectively the union of neighboring partition elements defined by h. The second observation can be motivated by the two lemmas retaining to the level-R hash function H^R which are detailed by [GFFN12]:

Lemma 3.2 *For a given bid $y = h(q)$, it holds that $y = H^1(q)$. That is, the so-called level-1 hash function H defines the same buckets as the hash function h, and thus the bid $h(q) = H^1(q)$.*

Lemma 3.3 *The R consecutive level-1 bids $zR, zR+1, ..., zR+(R-1)$ all map to the same level-R bid, z.*

Lemma 3.3 is exactly the reason that each bucket defined by a level-R hash function corresponds to R consecutive buckets prescribed by level-1 hash functions. Figure 5, demonstrates this fact in the geometric space corresponding to the partition elements induced by a level-1 function H^1 and its' corresponding level-3 function H^3.

Finally, the C2LSH datastructure builds on an important property of virtual rehashing, which is that any bid $H^{R'}(q)$ will be included in any higher level bid, $H^R(q)$, where $R \geq R'$ [GFFN12]. That is:

Lemma 3.4 *For some level-R' bid $H^{R'}(q)$, where $R \geq R'$ and $R, R' \in \mathbb{Z}^+$:*

$$\left\lfloor \frac{H^{R'}(q)}{R} \right\rfloor \cdot R \leq H^{R'}(q) \leq \left\lfloor \frac{H^{R'}(q)}{R} \right\rfloor \cdot R + (R-1)$$

3.2.2 Querying C2LSH

In [GFFN12] the authors present the C2LSH index structure for c-Approximate Nearest Neighbor Search. Therefore, the following section describes a slightly adapted index structure for k-ANNS, which retains the general structure and ideas presented by the authors.

When the C2LSH datastructure is queried, it maintains a variable R which is initially set to 1, a constant c which is typically set to 2 or 3, and a set $Q \subseteq \mathbf{X}$ which contains the points that collide with q under at least t level-R hash functions. More formally Q is defined:

$$Q = \{x \in \mathbf{X} \mid \text{collisions}(q, x) \geq t\}$$

where the function collisions(q, x) counts the number of level-R hash functions that q and x collide under:

$$\text{collisions}(q, x) := |\{H_i^R \in \{H_1^R, ...H_K^R\} \mid H_i^R(x) = H_i^R(q)\}|$$

Thus, first q is hashed according to the level-1 hash functions $H_1^R, ..., H_K^R$. All points $x \in \mathbf{X}$ belonging to the level-1 buckets of q are retrieved from the hash tables $T_1, ..., T_k$ and are evaluated according to collisions(q, x). A point from a level-1 bucket is added to Q if the number of collisions is equal to or larger than the threshold t. If afterward, the size of Q is still smaller than $partitionSize$, the value of R is increased by a factor of c, and a new iteration of bucket retrieval and collision counting is initiated, now considering the level-cR hash functions. This process continues until $|Q|$ is as large or larger than the build parameter $partitionSize$. It is important to note that for any level-cR bid, the datastructure only searches the tables $T_1, ..., T_K$ for the level-1 buckets included in the bid, which have not been retrieved by the previous level-R bid. This is done to avoid searching the same buckets multiple times and is possible due to lemma 3.4. The query operation for the C2LSH index structure is described in pseudo-code in algorithm 1.

It is important to note that the level-1 bucket B is not chosen at random from the remaining unchecked buckets in $H_i^R(q)$ but is instead chosen such that it always neighbors a previously checked bucket and such that it alternates between searching the left and right side when possible.

3.2.3 C2LSH Implementation Details

It is important to note that in their implementation, the authors of [GFFN12] maintain each hash table T_i as a sorted list of non-empty buckets. This is essentially done by offsetting all level-1 bucket IDs by a constant value, such that all non-empty bucket IDs become non-negative. This detail is essential to

Algorithm 1 C2LSH Query

Variables:
t = threshold
$partitionSize$ = minimum size of Q
c = expansion rate ($c = 2$ or $c = 3$)

procedure QUERY(D, q)
  $Q \leftarrow \emptyset$
  $prevR \leftarrow 0$
  $R \leftarrow 1$
  while True **do**
    for $1 \leq i \leq K$ **do**
      $minBid_i \leftarrow \lfloor \frac{H_i^1(q)}{R} \rfloor \cdot R$
      $maxBid_i \leftarrow \lfloor \frac{H_i^1(q)}{R} \rfloor \cdot R + (R - 1)$
      $nextBid_i \leftarrow$ unchecked neighboring bucket-ID,
      st. $minBid_i \leq nextBid_i \leq maxBid_i$
    for $prevR \leq R$ **do**
      for $1 \leq i \leq K$ **do**
        $B \leftarrow$ bucket $nextBid_i$ in T_i
        $Q \cup \{x \in B \mid \text{collisions}(q, x) \geq t\}$
        $nextBid_i \leftarrow$ unchecked neighboring bucket-ID,
        st. $minBid_i \leq nextBid_i \leq maxBid_i$
      if $|Q| \geq partitionSize$ **then**
        return Q
    $prevR \leftarrow R$
    $R \leftarrow Rc$

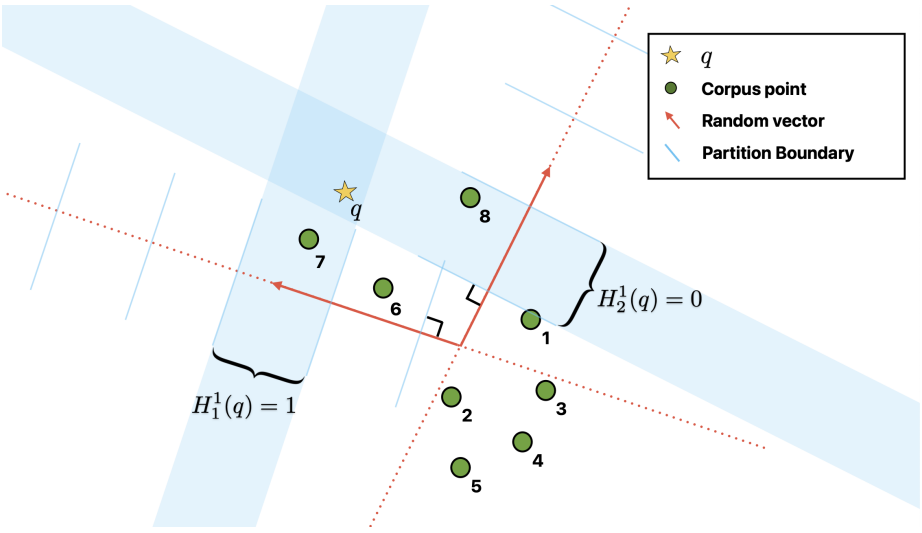

Figure 6: Partition elements corresponding to the level-1 bid $H^1(x)$ and the level-3 bid $H^3(x)$ for some point $q \in \mathbb{R}^d$

avoid a potential issue with the virtual rehashing technique described above, which is that it is not able to virtually rehash to include level-1 bids below 0 or above -1 if the initial level-1 bid $H^1(q)$ is positive or negative respectively. This fact can be demonstrated by the following small example, where the level-1 bid is $H^1(q) = 4$. When R is enlarged from 1 to 3, we have that $minBid = \lfloor \frac{4}{3} \rfloor \cdot 3 = 3$ and $maxBid = \lfloor \frac{4}{3} \rfloor \cdot 3 + (3 - 1) = 5$. When R is enlarged to 9, we have that $minBid = \lfloor \frac{4}{9} \rfloor \cdot 9 = 0$ and $maxBid = \lfloor \frac{4}{9} \rfloor \cdot 9 + (9 - 1) = 8$. Finally, when R is enlarged to 27, we have that $minBid = \lfloor \frac{4}{27} \rfloor \cdot 27 = 0$ and $maxBid = \lfloor \frac{4}{27} \rfloor \cdot 27 + (27 - 1) = 26$. At this point, it becomes clear that $minBid$ will always have a value of 0 for any subsequent rehashing, no matter how much R is increased. That is, whenever $R > H^1(q)$, we have that $minBid = 0$.

During the query of C2LSH, this can lead to an infinite search in larger and larger radii, where the stopping condition for the query is never met. An example instance of $\mathbf{X}$ and a C2LSH index structure for which an infinite query would occur is presented in figure 6.

The index structure uses the two hash functions h_1 and h_2, for which $H_1^1(q)$ and $H_2^1(q)$ hash to the values 1 and 0. Given these two hash hash functions and a set of build parameters; $t = 2$, $partitionSize = 5$, and $K = 2$, the stopping condition $|Q| \geq partitionSize$ will never be met. This is because increasing R will only lead to examining level-1 buckets of higher and higher positive values and therefore $|Q|$ is bound to a maximum size of 3, as the datastructure is unable to rehash the initial bids to include the corpus points 1 through 5.

In the implementation that is tested in this thesis, rather than offsetting all non-empty bucket IDs to be positive, a different and arguably simpler bucket ex-

pansion strategy is used. The implemented strategy simply assigns $minBid_i \leftarrow H_i^1(q) - (R-1) \operatorname{div} 2$ and $maxBid_i \leftarrow H_i^1(q) + (R-1) \operatorname{div} 2$ for every virtual rehashing iteration (Line 7-8, Algorithm 1). That is, the expansion radius R is distributed evenly to both directions of the previous bid. With this implementation, it is no longer necessary to sort and store buckets in a list, nor to offset bucket IDs whenever a bucket needs to be located. Rather, any symbol table can be used in which to store the level-1 buckets. The implementation of this thesis simply uses the standard class `HashMap` from the Java standard library, where the key-value pair types are `<Integer, LinkedList<Integer>>`. Finally, this virtual rehashing technique also allows for a more even expansion of the buckets to be searched, in respectively the positive and negative direction of the level-1 bid, $H^1(q)$.

3.2.4 C2LSH Build Parameters

In the previous sections, a consideration that has not been touched upon is the bucket-width of the level-1 hash functions in $\{H_1^1, ..., H_K^1\}$. For a large bucket width, corresponding to a high value of r relative to the distribution of points in $\mathbf{X}$, the level-1 iteration of the query operation can retrieve a set of points, Q which is much larger than the parameter $partitionSize$. Therefore, the authors recommend that each level-1 hash function utilizes a small value of r relative to the distance between points in $\mathbf{X}$. In the following experiments, r is treated as a fixed parameter and is set to the value 1 as recommended by the authors, unless otherwise mentioned. Furthermore, the authors experimentally evaluate differing values of the parameter c and find that $c = 3$ performs best for multiple datasets. Thus, the evaluated implementation also treats c as a fixed parameter, using the recommended value of 3.

Given that R and c are fixed, the C2LSH datastructure is constructed according to the three build parameters t, K, and $partitionSize$. The parameter t is tightly connected to the build parameter K. For an assignment of build parameters where $t > 1$ and $t = K$, C2LSH can somewhat be understood as a variant of the LSH datastructure described above, where a set of points Q is identified corresponding to an optimal value of r for a given K and some target value of $|Q| = partitionSize$. Under this interpretation of C2LSH, r effectively becomes a dynamic variable that is set automatically for each instance of q to accommodate a target size of the returned set of points Q.

For some assignment of values where $t < K$, the difference $K - t$ can be interpreted as the number of hash functions $h_i \in \{h_1, ..., h_K\}$ which can be considered irrelevant. The motivation for having a collision threshold t which is smaller than K, is to provide an error margin for 'unlucky' hash functions, in which close points are not hashed to the same bucket as q.

Finally, while it is theoretically possible to utilize C2LSH as an ensemble index structure such that $L = \{D_1, ..., D_{|L|}\}$, this is not an approach that will be covered in the experiments conducted in this thesis. Limited testing showed little promise in the performance of this approach while increasing the complexity of setting relevant build parameters for the datastructure significantly.

3.2.5 Asymptotic Analysis of the C2LSH Index Structure

Much of the relevant asymptotic analysis of the C2LSH index structure is similar to the analysis of the LSH index structure. Therefore the following section will only provide a superficial analysis. The C2LSH index structure is composed of K hash tables, which each store the indices of all points in $\mathbf{X}$. The index structure also contains K hash functions which each require a vector in d dimension. Therefore, with the K hash tables being implemented with the `HashMap` class, the space complexity is $\mathcal{O}(K \cdot n + K \cdot d) = \mathcal{O}(K \cdot (n + d))$.

Building the C2LSH index structure requires first of all determining K hash functions, which each are constructed in time proportional to d (the time it takes to construct $\vec{a}$). Subsequently, indexing n points in the K hash tables takes time $\mathcal{O}(n \cdot K \cdot d)$ assuming an asymptotic insertion time of $\mathcal{O}(1)$, which leads to a total asymptotic index construction time of $\mathcal{O}(k \cdot d + n \cdot K \cdot d) = \mathcal{O}(n \cdot K \cdot d)$

As opposed to the other three index structures covered in this thesis, it is not possible to provide guarantees for the query time of C2LSH. Each iteration of both the first for loop (line 6, algorithm 1) and the second for loop (line 11, algorithm 1) finishes in time proportional to K. However, it is not possible to determine how many iterations of each loop are conducted, as it depends entirely on the build parameters $partitionSize$, t, and not least the distribution of corpus points.

3.3 Random k-Dimensional Forest

Relevant source code: `RKDTree.java, Tree.java`

The first tree-based index structure for k-ANNS which will be detailed, is the index structure $L = \{D_1, ..., D_{|L|}\}$ where any $D_i \in L$ is a Random k-Dimensional Tree (abbreviated RKD-Tree). Therefore the ensemble index structure will be referred to as the RKD-Forest. The RKD-Tree is a datastructure first proposed in [SAH08] and is a stochastic variant of the k-d Tree, which was initially proposed for solving the k-NNS problem in [Ben75]. In essence, an RKD-Tree is a binary search tree that recursively splits a subspace of $\mathbb{R}^d$ into two mutually disjoint subspaces based on the corpus points in $\mathbf{X}$. The root node thus contains the entire space $\mathbb{R}^d$ and therefore also the full set $\mathbf{X}$. At the root node, a splitting hyper-plane is determined such that half of the points in $\mathbf{X}$ are positioned on one side, and the other half is positioned on the other side of the hyper-plane. The two spaces determined by the hyper-plane are then recursively divided according to the same process, considering only the subset of $\mathbf{X}$ belonging to the respective subspaces. Depending on the implementation, the recursion stops and a leaf node is created, when either a certain tree depth is reached or the subspace defined by a node contains fewer corpus points than a given threshold. Thus, the leaf nodes of a tree induce a partition on $\mathbb{R}^d$ and therefore also $\mathbf{X}$. At each leaf node, the corpus points belonging to the partition element defined by the node are stored. Partition trees typically vary in the way that the splitting hyper-plane is constructed and therefore how $\mathbb{R}^d$ is

partitioned. The RKD-Tree creates axis-aligned splitting hyper-planes which are based on a median value of vector components in some dimension i. For any node in the RKD-Tree, the splitting dimension i is drawn uniformly at random from the set of dimension indices $O \subseteq \{1, ..., d\}$ containing the o dimensions of highest variance, calculated from the corpus points belonging to the subspace defined by the node. More formally, let $N \subseteq \mathbf{X}$ be the set of corpus points belonging to a specific node, let $[x]_j$ denote the j-th component of $x \in N$ and finally let $[N]_j$ denote the multiset of the j-th components of points in N. Then:

$$O = \underset{j \in \{1,...,d\}}{o\text{-}\operatorname{argmax}}(\text{variance}([N]_j))$$

During the construction of the RKD-Tree, a corpus point $x \in N$ is assigned to either the left child-node N_L or the right child node N_R according to the following recursion, provided that the stopping condition has not been met:

$$x \in \begin{cases} N_L & \text{if } [x]_i < \text{median}([N]_i) \\ N_R & \text{if } [x]_i \geq \text{median}([N]_i) \end{cases}$$

Thus the splitting hyper-plane is orthogonal to the i-th dimension axis, which it intersects at the point $\mathbf{e}_i \cdot \text{median}([N]_i)$, where $\mathbf{e}_i$ is the i-th standard-basis vector. In the unlikely scenario that points lie on the hyper-plane, these are consistently assigned to the left child node.

An RKD-Tree $D_i \in L$ is queried much in the same way that it is built. Given a query vector q, the tree is recursively traversed starting from the root node until a leaf node is encountered. At each internal node, the query is routed to either the left or right child node by determining which side of the splitting hyper-plane q belongs to. Once a leaf node is encountered the set of corpus points Q_i, belonging to the leaf node partition element are returned. The traversal is described in pseudo-code in algorithm 2.

Algorithm 2 RKD-Tree Query

```
1: procedure QUERY(N, q)
2:     while N is not leaf node do
3:         i ← splitting dimension of N
4:         if [q]_i < median([N]_i) then
5:             N ← N_L
6:         else
7:             N ← N_R
8:     return points in N
```

In the above pseudo-code, it is important to note that both i and $\text{median}([N]_i)$ are pre-computed values that are computed during the build phase.

3.3.1 RKD-Tree Build Parameters

The RKD-Tree-based index structure has a total of three build parameters. As stated earlier the recursion used to construct the RKD-Tree can either use

a maximal depth or a maximum leaf-size threshold as a stopping condition. The variant of the RKD-Tree which is tested throughout this thesis implements the maximum leaf-size build parameter, $leafSize$. The smaller the value of $leafSize$, clearly the smaller Q_i returned by the query(D_i, q) as $|Q_i|$ is upper bounded by $leafSize$. Therefore, $leafSize$ is an effective parameter at controlling $|\mathbf{C}'(q)|$. Furthermore, a smaller value of $leafSize$ will also trim a greater number of irrelevant (distant) candidates. This however also presents a qualitative drawback, in that a smaller $leafSize$ increases the likelihood that a corpus point belonging to $x \in k\text{NN}(q)$ is assigned to a neighboring subspace and therefore not returned by query(D_i, q). This issue is especially relevant if the points in $\mathbf{X}$ are clustered around subspace boundaries [DS15]. To remedy this fact the second build parameter $|L|$, which dictates the number of partitioning trees, can be increased. As $|L|$ is increased the likelihood that a point $x \in k\text{NN}(q)$ is returned by query(D_i, q) for at least one $D_i \in L$ is also increased.

Finally, the RKD-Tree-based index structure also takes the build-parameter o, which determines for each internal node the number of dimensions that the splitting dimension i is drawn from. Following suggestions from [ML14] o is not treated as a configurable build parameter in the following experiments, but is instead fixed to a value of 5. Therefore the two free build parameters for the tested implementation are $leafSize$ and $|L|$.

3.3.2 Asymptotic Analysis of the RKD-Forest Index Structure

To understand the query- and build-time of the RKD-Forest, I will start by detailing the space complexity. Once again, let n denote $|\mathbf{X}|$. Assuming that an RKD-Tree always splits subspaces such that an equally large number of points belonging to $\mathbf{X}$ are assigned to the left and right subspaces, the RKD-Tree has a height of $\log_2 n$ given that $leafSize = 1$. For some value of $leafSize > 1$, and under the assumption that nodes are always split evenly, a tree has a max height of $\log_2 \frac{n}{leafSize}$. For the sake of simplicity, the maximum value height of $\log_2 n$ will be used in the following analysis. At each level l, the RKD-Tree contains 2^l nodes, where each node contains a constant number of pointers (left node, right node, splitting dimension and a splitting value). Finally, a total of n corpus point indices are stored at the leaf level, regardless of $leafSize$. From the geometric sum: $2^0 + 2^1 + 2^2 + ... + 2^{log_2 n}$, we have that a single tree contains $2n - 1$ nodes which each occupy constant space, as well as n corpus point indices, leading to a spatial complexity of $\mathcal{O}(2n-1+n) = \mathcal{O}(n)$. Therefore, an RKD-Forest of size $|L|$ has a total space complexity of $\mathcal{O}(|L| \cdot n)$.

The building of an RKD-Tree requires at every node to first calculate the variance across d dimensions for a subset of $\mathbf{X}$, then select the splitting dimension, calculate the splitting value (median component in the chosen dimension), and finally assign points to the left or right child node comparing a single component against the splitting value.

Given that each node carries out calculations for a mutually disjoint subset of $\mathbf{X}$ for each level, then all variances at each level can be calculated in time $\mathcal{O}(n \cdot d)$. The o dimensions of highest variance can be determined for each node

in time linear to d, using the Quick-select algorithm. Thus, at every level of the RKD-Tree picking the splitting dimensions takes $\mathcal{O}(n \cdot d + 2^l \cdot d)$. Since $2^l \leq n$, this gives $\mathcal{O}(n \cdot d)$. Following the selection of the splitting dimension, the splitting values for an entire level can be determined in $\mathcal{O}(n)$ using an efficient implementation of the median calculation. Assigning all points to their respective child-node likewise is done in time $\mathcal{O}(n)$. Thus, each level is created in time $\mathcal{O}(n \cdot d + n + n) = \mathcal{O}(n \cdot d)$. Given the max height of the tree being $log_2 n$, this leads to a total asymptotic build time of $\mathcal{O}(log_2 n \cdot n \cdot d)$ for a single RKD-Tree and $\mathcal{O}(|L| \cdot log_2 n \cdot n \cdot d)$ for the entire forest.

Finally, querying a single RKD-Tree with a vector q requires at every level to compare exactly one component of q against a splitting value. Therefore querying a single RKD-Tree has a time complexity of $\mathcal{O}(\log_2 n)$ while querying an entire RKD-Forest has a time complexity of $\mathcal{O}(|L| \cdot \log_2 n)$.

3.4 Random Projection Forest

Relevant source code: `RPTree.java, Tree.java`

The second tree-based index structure uses an ensemble of Random Projection Trees (abbreviated RP-Tree) and is therefore called the RP-Forest. The RP-Tree is a space-partitioning tree datastructure that was first introduced in [DS15]. As previously stated, space-partitioning trees typically vary in the way that the splitting hyper-plane for any internal node is constructed. Just like the RKD-Tree, the RP-Tree also relies on recursively splitting subspaces of $\mathbb{R}^d$ based on the points in $\mathbf{X}$ belonging to the subspace. Rather than an axis-aligned splitting hyper-plane, RP-Trees determine splitting hyper-planes which are orthogonal to the direction of random vectors sampled from the unit sphere [HPT+16].

During the build-phase, at each internal node, a corpus point $x \in N$ is assigned to either of the child nodes N_L or N_R according to the median projected value of corpus points in N onto the chosen random vector $\vec{v} \in \mathbb{R}^d$. More formally:

$$x \in \begin{cases} N_L & \text{if } x \cdot \vec{v} < \text{median}(\text{proj}(N, \vec{v})) \\ N_R & \text{if } x \cdot \vec{v} \geq \text{median}(\text{proj}(N, \vec{v})) \end{cases}$$

where $\text{proj}(N) = [x \cdot \vec{v} \mid x \in N]$ and where $[\ ...\ ]$ denotes a multiset. In the original RP-tree proposed in [DS15] a random vector is created for each internal node N and the random vector $\vec{v}$ is 'dense' in the sense that it is a fully d-dimensional vector.

In [HPT+16] the authors propose a datastructure called a sparse RP-Tree which showcases a series of space and time-saving improvements to the RP-Tree. Most noteworthy is that the sparse RP-Tree uses sparse random vectors to determine the splitting hyper-planes, which reduces both the time complexity of building and querying the tree. Such a sparse random vector $\vec{v}$ is constructed by assigning each vector component $[\vec{v}]_i$ in the following way:

$$[\vec{v}]_i = \begin{cases} r & \text{with probability } a \\ 0 & \text{with probability } 1 - a \end{cases}$$

where r is an i.i.d value drawn from $\mathcal{N}(0, 1)$ and a is a parameter that can be tuned to determine the sparsity of the resulting vector [HPT+16]. Furthermore, the authors propose a datastructure that instead of computing and storing a random vector for each internal node, reuses the same random vector across all internal nodes of the same level in the sparse RP-Tree.

Just like the RKD-Tree, the sparse RP-Tree is queried with some query vector q by traversing from the root node until a leaf node is encountered, at which point the set of points belonging to the defined subspace is returned as the set Q. Algorithmically, the query operation is carried out as detailed in Algorithm 3 where median(proj$(N, \vec{v})$) is a pre-computed value stored to each node and randomVectors is the collection which holds the sparse random vector used at each level of the tree:

Algorithm 3 Sparse RP-Tree Query

```
1:  procedure QUERY(N, q)
2:      level ← 0
3:      while N is not leaf node do
4:          v ← randomVectors[level]
5:          if q · v < median(proj(N, v)) then
6:              N ← N_L
7:          else
8:              N ← N_R
9:          d ← d + 1
10:     return points in N
```

3.4.1 RP-Tree Build Parameters

The ensemble index structure based on sparse RP-Trees takes a total of three build parameters. In the implementation tested throughout this thesis, the sparse RP-tree is implemented with the $leafSize$ parameter as a stopping condition for the build recursion. Besides the $leafSize$ the building of the sparse-RPTree also takes the parameter $|L|$ which determines the number of trees in the final index structure. Both the parameters $leafSize$ and $|L|$ have similar effects to the corresponding build parameters of the RKD-Tree-based index structure. Finally, the index structure also takes the build parameter a, where $0 < a \leq 1$. This parameter determines the probabilistic sparsity of each random vector used to build and query each tree $D \in L$. The higher the value of a the more dense the random vector. Naturally, the more dense the random vectors used, the higher the query time since at each level of a tree q is expected to be projected onto an $a \cdot d$-dimensional vector. It has been shown experimentally

and theoretically that $a = \frac{1}{\sqrt{d}}$ provides a good balance of sparsity and robustness [HPT+16, LHC06]. Therefore, throughout this thesis, a is simply treated as a fixed build-parameter for index structures based on sparse RP-Trees. This leaves the final configurable build parameters: $leafSize$ and $|L|$.

3.4.2 Asymptotic Analysis of the RP-Forest Index Structure

Detailing the query and build time complexity of the RP-Forest once again requires to first examine the spatial complexity. The RP-Tree has a maximal height of $\log_2 n$ and each node contains a constant number of pointers. Furthermore, the RP-Tree stores the indices of n points at the leaf level. Thus the RP-Tree itself has a size of $\mathcal{O}(n)$. For each level of the tree, a single sparse vector is stored, which (given that $a = \frac{1}{\sqrt{d}}$) has a probabilistic size of $\sqrt{d}$. Therefore the total space complexity of a single RP-Tree is $\mathcal{O}(n + \log_2 n \cdot \sqrt{d})$ and of a RP-Forest is $\mathcal{O}(|L| \cdot (n + \log_2 n \cdot \sqrt{d}))$[JAL23].

Constructing an RP-Tree requires creating a random sparse vector at each level, where each vector construction takes time proportional to d. Furthermore, at every level, a total of n points are projected onto a sparse vector where each projection takes time proportional to the dimensionality of the vector. Again, the median projection value can be calculated in time $\mathcal{O}(n)$ for all nodes at each level of the tree. Thus each level of the tree is constructed in $\mathcal{O}(d+n\cdot\sqrt{d}+n) = \mathcal{O}(d + n \cdot \sqrt{d})$[i] and a single tree is constructed in $\mathcal{O}(log_2 n \cdot (d + n \cdot \sqrt{d}))$. The complete RP-Forest can therefore be constructed in time $\mathcal{O}(|L| \cdot log_2 n \cdot (d + n \cdot \sqrt{d}))$.

Finally, to query an RP-Tree, the query vector is projected onto a sparse vector at each level. Thus, to query a single RP-Tree has an asymptotic running time of $\mathcal{O}(\log_2 n \cdot \sqrt{d})$ and querying the full RP-Forest has an asymptotic running time of $\mathcal{O}(|L| \cdot \log_2 n \cdot \sqrt{d})$[DS15].

[i] although it is quite unlikely that $d \geq n \cdot \sqrt{d}$

4 Search Strategies

Relevant source code: `ANNSearcher.java`

Having examined the four different index structures which will be subject to experimental analysis, I will now turn to the four search strategies that each index structure can be coupled with in order to construct a complete k-ANNS algorithm. The four search strategies to be examined are:

- Lookup Search
- Voting Search
- Natural Classifier Search
- Quick-select Natural Classifier Search

For each search strategy, I will first provide an overview of each search strategy and how it couples with an index structure to provide a k-ANNS solution. Essentially, this means detailing how each search strategy approaches the task of building a qualified candidate set $\mathbf{C}(q)$ from some index structure. I will also provide a short discussion on setting search parameters for each search strategy, as well as provide an overview of select implementation choices.

Note that for the Lookup Search, Voting Search and Natural Classifier Search strategy, the search strategy overview section has been largely covered in [JAL23], but has been modified to fit the algorithmic framework which is based on index structure and search strategy combinations. The discussions on setting parameters and implementation details are not covered in [JAL23].

4.1 Lookup Search

The simplest of the four search strategies is Lookup Search, which will be considered a baseline strategy for partition-based algorithms for k-ANNS. Despite not being mentioned explicitly as a search strategy, Lookup Search is used in many algorithms such as those presented in [DIIM04, DS15, GFFN12]. The Lookup Search strategy is characterized by not having any candidate refinement process, but rather the candidate set $\mathbf{C}(q)$ is exactly the same as $\mathbf{C}'(q)$. That is, given an index structure L, in Lookup Search the candidate set $\mathbf{C}(q)$ is constructed directly from querying each datastructure $D \in L$:

$$\mathbf{C}(q) = \bigcup_{D_i \in L} \text{query}(D_i, q)$$

The candidate set $\mathbf{C}(q)$ is subsequently brute-force searched in the re-ranking step.

Therefore the Lookup Search strategy is also completely without search parameters and relies entirely on the configuration of the index structure L to produce results in a desired recall range. This also makes the search strategy

rather inflexible, as supporting a range of recall values requires multiple index structures built according to different parameters. Furthermore, Lookup Search also typically requires rather large index structures for high recall ranges, as build parameters are modified (typically $|L|$ is increased) in order to increase the likelihood that points from $k\text{NN}(q)$ are included in $\mathbf{C}(q)$. This in turn also affects the search time negatively as both the time spent building $\mathbf{C}(q)$ increases and the size of $\mathbf{C}(q)$ itself can grow to a size that is infeasible to brute-force search.

4.2 Voting Search

The next search strategy was proposed in [HPT+16] as a search strategy designed specifically for tree-based index structures. The search strategy aims to mitigate some of the issues present with Lookup Search and produce a more qualified candidate set, by introducing a candidate refinement condition and an accompanying search parameter, τ for which $\tau \in \mathbb{Z}^+$. In essence, Voting Search works on ensemble-based index structures (RP-Forest, RKD-Forest and LSH) by counting the number of times that a corpus point $x \in \mathbf{X}$ appears in partition elements Q_i retrieved from querying all datastructures $D_i \in L$. Corpus points are then only added to $\mathbf{C}(q)$ if they appear in at least τ such partition elements. More formally, the search strategy relies on a score function, $\text{Frequency}(L, x)$, which is defined as:

$$\text{Frequency}(L, x) := \sum_{i \in \{1,\ldots,|L|\}} \begin{cases} 1 & \text{if } x \in Q_i \\ 0 & \text{else} \end{cases}$$

This score-function is used together with the search parameter τ in order to create the candidate set $\mathbf{C}(q)$ from the preliminary candidate set $\mathbf{C}'(q)$, according to the following candidate refinement:

$$\mathbf{C}(q) = \{x \in \mathbf{C}'(q) \mid \text{frequency}(L, x) \geq \tau\}$$

After candidate refinement, $\mathbf{C}(q)$ is subject to the re-ranking step consisting of a brute-force search over the points in $\mathbf{C}(q)$ in order to return a final solution $k\text{ANN}(q)$. Note that $1 \leq \tau \leq |L|$ and that for the special case of $\tau = 1$, Lookup Search and Voting Search return an identical solution, $k\text{ANN}(q)$, given the same index structure.

Voting Search mitigates some of the issues present in the Lookup Search strategy by first of all providing a larger degree of flexibility through the introduction of a candidate refinement process. This allows for a single index structure to support queries in a range of recall values. The candidate refinement is motivated by the fact that points with a higher frequency value are more likely to be close to q and thus more relevant to include in $\mathbf{C}(q)$. On the other hand, less frequent points are more likely to be outliers and therefore not relevant to include [HPT+16]. This also means that the less relevant points are more likely to be trimmed first when τ is increased and therefore that, at least

theoretically, the candidate refinement process can improve the search time over the Lookup Search strategy for a given L by reducing the number of distance computations in the brute-force search, while only making minor reductions to the search quality.

4.2.1 Voting Search Implementation Details

Voting Search naturally is marginally more complex than Lookup Search, which makes it necessary to discuss the implementation choices related to Voting Search. In essence, the strategy iterates over all datastructures $D_i \in L$, querying them with vector the query vector $q \in \mathbb{R}^d$ one at a time. For each query operation, the strategy iterates over the elements in the set of retrieved points Q_i, for each point counting the frequency for which it has been encountered during the search and adding a point to $\mathbf{C}(q)$ when it has been encountered τ or more times. In pseudo-code, this can be described in the following way, where the variable count(x) counts the number of times that x has been encountered during iteration:

Algorithm 4 Voting Search

```
1: procedure VOTINGSEARCH(L, q, τ)
2:     C(q) ← ∅
3:     for D_i ∈ L do
4:         Q_i ← query(D_i, q)
5:         for x ∈ Q_i do
6:             Increment count(x) by 1
7:             if count(x) = τ then
8:                 Add x to C(q)
9:     return bruteForceSearch(C(q), q)
```

This of course begs the question of which datastructure is used to store the frequency count of all encountered points. Two options were considered during implementation. The first option is a `HashMap` from the standard Java library with the key-value pair $< Integer, Integer >$ where the entry with key i represents the i-th point in $\mathbf{X}$ and the value counts the frequency. The second is a simple array of `int` of size $|\mathbf{X}|$ where the value stored at index $i-1$ counts the frequency for the $i-1$-th corpus point.

Consider first the option using a `HashMap` implementation to count frequencies. The `HashMap` class in Java 11 is implemented using so-called separate chaining with a `LinkedList` holding colliding entries. Let n denote the number of entries and m denote the capacity of the array holding m `LinkedList`s. Then this implementation provides constant time insertion and retrieval $\mathcal{O}(1)$[ii]. However, this is only true in the amortized sense, due to dynamic resizing. If the datastructure reaches a predetermined load factor threshold, the underlying

[ii] Technically the insertion and retrieval is $\mathcal{O}(1 + m/n)$, but assuming a clever choice of n this is $\mathcal{O}(1)$

array is resized to double its original size ($2m$), and all entries are rehashed according to a new hash function, which takes time $\mathcal{O}(n)$ [Ora]. An approach to avoiding frequent resizing during the search, is to provide a large initial capacity for the `HashMap`, for example $|\mathbf{X}|/100$.

The second option of using a simple array of size $|\mathbf{X}|$ will provide constant time insertion and retrieval for all cases. However, this method also comes with an added overhead of having to allocate in memory a potentially very large array of size $|\mathbf{X}|$, which can potentially be a heavy operation. To avoid such an allocation taking place for each search, the array is constructed before the first search is carried out. This requires a simple method to reset all values stored in the array for each search - either before or after a search is carried out.

Both methods were experimentally evaluated for a single dataset. In the limited testing carried out, the second option provided significantly faster searching for a wide range of recall values. Furthermore, it is a markedly simpler solution given that it does not require configuring the initial capacity to fit a specific recall range. Therefore, the second option, that is using an array of size $|\mathbf{X}|$ to count frequencies, was chosen for the implementation of Voting Search.

4.2.2 Setting Parameters for Voting Search

In relation to the baseline search strategy Lookup Search, there are multiple overall strategies for parameterizing an index structure to be used with Voting Search. The first approach is based on the realization that for a given index structure L, Voting Search is able to produce a $k\text{ANN}(q)$ solution which is upper-bounded in its recall value by the solution provided by searching L with Lookup Search. In the best case, Voting Search will provide a slightly worse recall but a considerable search time speed-up over Lookup Search. Thus for Voting Search to achieve a similar recall to Lookup Search for some index structure L, a possible approach could be to increase $|L|$ while maintaining all other build parameters fixed. This could potentially reduce the search time for an equal recall for some value of $\tau > 1$.

The second strategy is to consider a set of build parameters for the index structure which induces larger partition elements on $\mathbb{R}^d$, which are therefore more likely to return a larger Q_i for each $D_i \in L$. This is done under the assumption that datastructures that induce partition elements containing a larger number of corpus points, will still include distant points to q less frequently than closer points, and that distant points can therefore be expected to be discarded during the candidate refinement process. Simultaneously, larger partition elements are likely to include more points belonging to $k\text{NN}(q)$, than smaller partition elements are. For Lookup Search, this could mean a blow-up in search time as all encountered corpus points are included in $\mathbf{C}(q)$. For Voting Search, however, many irrelevant corpus points could be pruned in the candidate refinement process for some value of $\tau > 1$. This approach could allow for a smaller value of $|L|$ for a similar recall, which depending on the index structure, could potentially provide a considerably faster search time.

It is evident from figure 4 that τ is an efficient parameter at reducing the

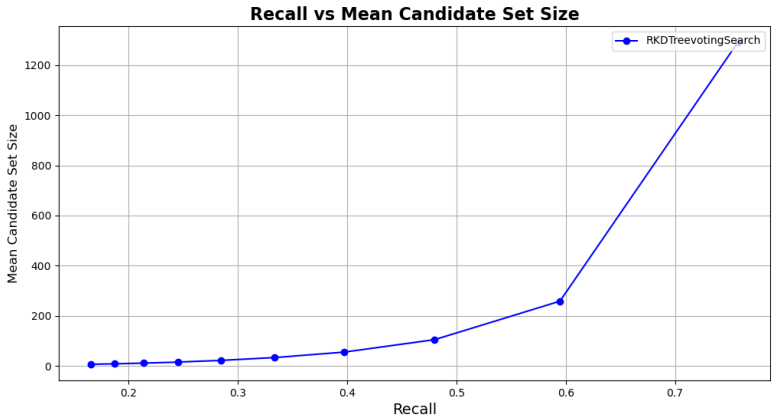

Figure 7: Plot of recall/candidate set size trade-off for the RKDTreeVotingSearch algorithm with $leafSize = 8$, $|L| = 350$ and τ ranging from 1 (right-most point) to 10 (left-most point) on the SIFT dataset. Note the difference in recall for neighboring τ values.

size of $\mathbf{C}(q)$, and thus decreasing search time (at the cost of accuracy) for some fixed index structure, L. However, an important aspect of Voting Search is that it is only possible to set τ as integer values. Therefore, it is not possible to perform any fine-grained tuning of the search strategy between some values of $\tau = s$ and $\tau = s + 1$. This is further complicated by the fact that the recall difference between two values of τ can be quite significant (see figure 7). In order to perform fine-grained tuning, it is necessary to modify the build parameters of the associated index structure, such that the recall for each value of τ is shifted upwards or downwards. Naturally, increasing the value of $|L|$ for an ensemble-based index structure monotonically increases the recall for a fixed value of τ. This tuning strategy can be necessary in scenarios in which the goal is to minimize the search time for a specific recall value or range.

4.3 Natural Classifier Search

The third search strategy to be examined is the Natural Classifier Search which was recently proposed in [HJR22a]. The search strategy aims to mitigate a general problem of the above-described search strategies and index structures, which is that for any query operation, query(D_i, q), the search strategies presented so far are unable to consider corpus points that may belong to kNN(q) but reside in neighboring partition elements. As stated above, this fact is typically remedied by increasing $|L|$ or increasing the size of the partition elements created by each datastructure $D_i \in L$, which can present a considerable degradation in search time. Therefore, the Natural Classifier Search strategy introduces a supplementary datastructure which is essentially a lookup table containing the

set $k\text{NN}(x)$ for all $x \in \mathbf{X}$. That is, the lookup table index structure is defined as an $|\mathbf{X}|$-tuple of sets: $(k\text{NN}(x_1), ..., k\text{NN}(x_{|\mathbf{X}|}))$ where x_i denotes the i-th point in $\mathbf{X}$. It is important to note that for any $x \in \mathbf{X}$, x is its' own nearest neighbor - that is $\text{NN}(x) = x$. This lookup table is created during the build phase where each $k\text{NN}(x)$ is calculated by an exact k-NNS algorithm.

With this lookup table, the search strategy is able to consider corpus points outside the subspaces defined by the partitioning datastructures. In order to do so, during the search phase, a corpus pint $y \in \mathbf{X}$ is added to $\mathbf{C}'(q)$ iff. it belongs to the set of k exact nearest neighbors of a corpus point in some set of retrieved points, $x \in Q_i$. This can in the worst case lead to an incredibly large preliminary candidate set $\mathbf{C}'(q)$, as $|\mathbf{C}'(q)| \leq |\{Q_i\}_{i \in \{1,...,|L|\}}| \cdot k$. Therefore an efficient candidate refinement is needed to avoid a large number of distance computations in the re-ranking step. In this context, the Natural Classifier Search strategy introduces a search parameter τ where $0 \leq \tau \leq 1$. In the candidate refinement process, any point $y \in \mathbf{C}'(q)$ is scored according to how frequently it appears as a nearest neighbor to corpus points in Q_i for all $D_i \in L$. In order to account for varying sizes of Q_i, this score is weighted according to $|Q_i|$ and thus becomes a relative frequency score. Finally, an average of the relative frequency scores across all Q_i retrieved from iterating over $D_i \in L$ is calculated, and a point $x \in \mathbf{C}'(q)$ is added to $\mathbf{C}(q)$ only if the average score is greater than or equal to the threshold value τ.

The search strategy can be formally defined in the following way: Given Q_i retrieved from querying the datastructure $D_i \in L$ with the query point $q \in \mathbb{R}^d$ ($\text{query}(D_i, q)$), the lookup table is used in order to create a derived superset of corpus points Q_i' based on the following construction:

$$Q_i' = \{x \in Q_i \mid y \in k\text{NN}(x)\}$$

Thus it follows that for the Natural Classifier Search strategy the preliminary candidate set $\mathbf{C}'(q)$ is defined in the following way:

$$\mathbf{C}'(q) = \bigcup_{i \in \{1,...,|L|\}} Q_i'$$

Given a set of corpus points retrieved corpus point Q_i, and some corpus point $y \in Q_i'$, the score function $\text{RelativeFrequency}(Q_i, y)$ is defined in the following way:

$$\text{RelativeFrequency}(Q_i, y) := \frac{1}{|Q_i|} \cdot \sum_{x \in Q_i} \begin{cases} 1 & \text{if } y \in k\text{NN}(x) \\ 0 & \text{else} \end{cases}$$

Given the scoring function above, the average relative frequency of a corpus point $y \in \mathbf{C}'(q)$, given an index structure L and a query vector q is defined:

$$\text{AvgRelativeFrequency}(L, q, y) = \frac{1}{|L|} \cdot \sum_{i \in \{1,...,|L|\}} \text{RelativeFrequency}(Q_i, y)$$

Finally, the candidate set $\mathbf{C}(q)$ is then defined:

$$\mathbf{C}(q) = \{y \in \mathbf{C}'(q) \mid \text{AvgRelativeFrequency}(L, q, y) \geq \tau\}$$

4.3.1 Natural Classifier Search Implementation Details

The Natural Classifier Search is very similar in its implementation structure to the Voting Search strategy described above. It iterates over all $D_i \in L$, querying them one at a time. For all corpus points $x \in Q_i$ retrieved from the query, it iterates of the set of nearest neighbors $k\text{NN}(x)$, for each point $y \in k\text{NN}(x)$ incrementing the variable avgRelativeFrequency(y) by $\frac{1}{|Q_i|\cdot|L|}$. If avgRelativeFrequency(y) exceeds τ, it is added to the candidate set $\mathbf{C}(q)$. Finally, the candidate set $\mathbf{C}(q)$ is searched according to the brute-force approach and the top k candidates are returned. In pseudo-code, this process can be described in the following way:

Algorithm 5 Natural Classifier Search

```
1:  procedure NATURALCLASSIFIERSEARCH(L, q, τ)
2:      C(q) ← ∅
3:      for D_i ∈ L do
4:          Q_i ← query(D_i, q)
5:          voteWeight ← 1 / (|Q_i|·|L|)
6:          for x ∈ Q_i do
7:              for y ∈ KNN(x) do
8:                  Increment avgRelativeFrequency(y) by voteWeight
9:                  if avgRelativeFrequency(y) ≥ τ then
10:                     Add y to C(q)
11:     return bruteForceSearch(C(q), q)
```

Thus, the Natural Classifier Search strategy encounters a very similar task to that of Voting Search of having to maintain a vote value for each point. However, rather than counting a `int` value, instead a `float` value has to be maintained. Following the same line of reasoning as for the Voting Search, the datastructure that maintains the votes of each corpus point is implemented simply as an array of `float` which is constructed before the search and is reset at the beginning of any subsequent search. It is also worthwhile to mention that the candidate set $\mathbf{C}(q)$ is implemented as a `HashSet` from the Java standard library, which is necessary to avoid duplicate points in $\mathbf{C}(q)$ as a point y can be added multiple times.

The described lookup table is simply implemented as a 2-dimensional array of int (`int[][]`) which ensures constant time retrieval of any $k\text{NN}(x)$ for $x \in \mathbf{X}$. This search strategy therefore also requires $\mathcal{O}(n \cdot k)$ extra space over the Lookup Search and Voting Search strategies.

4.3.2 Setting Parameters for Natural Classifier Search

Given some ensemble index structure L which provides a given recall value for Lookup Search, we have that coupling the Natural Classifier Search strategy with L, the recall value of the Natural Classifier Search is no longer bounded by the Lookup Search recall, as was the case for Voting Search. In fact, if expanding

the preliminary candidate set $\mathbf{C}'(q)$ as described above is effective, potentially Natural Classifier Search could achieve a significantly higher recall value searching L for some value of τ. Therefore a general strategy for determining build parameters for an ensemble index structure Natural Classifier Search is to fix build parameters that work well for Lookup Search but reducing $|L|$. Subsequently, the search parameter τ can be used to trim points in $\mathbf{C}'(q)$ with lower values of AvgRelativeFrequency(L, q, x) and which are therefore intuitively less likely to belong to kNN(q). For the C2LSH index structure, the corresponding strategy could be to reduce $partitionSize$, while possibly increasing values of K and t.

When building index structures for Natural Classifier Search, it is however also possible that datastructures that induce large partition elements of $\mathbb{R}^d$ and therefore could contain whole clusters of corpus points, could be less suited for Natural Classifier Search. This is due to the fact, that the score function would score corpus points higher which belong to high-density areas, despite them not necessarily being in close proximity to q, and therefore would effectively be identifying clusters within a large partition element. This effect is studied in greater detail in section 4.4.1.

Settings algorithm parameters for the Natural Classifier Search is complicated by certain characteristics of the two tree-based index structures; RP-Forest and RKD-Forest. The threshold value τ used in the Natural Classifier Search being configured as a `float` value between 0 and 1 inclusive, can lead one to believe that it provides more fine-grained tuning than what is possible for Voting Search. However, the Natural Classifier Search exhibits very similar characteristics to the Voting Search threshold value described in section 4.2.2, but only for specific index structures. Namely, it is not necessarily possible to configure a specific recall value for a fixed index structure, L. This is true for the tree-based index structures, for which each partition element has certain properties about the number of corpus points contained within the partition element. For both the RKD-Tree and RP-Tree datastructures, any leaf node is expected to contain between $leafSize$ and $\frac{leafSize}{2}$ corpus points. This essentially means that for any Q_i retrieved from querying $D_i \in L$ during the Natural Classifier Search strategy execution, the value $voteWeight$ (see algorithm 11, line 5) with high probability is the same. This gives the same ordering of the values avgRelativeFrequency(y) for each point in $y in \mathbf{C}'(q)$ as if the $voteWeight$ was always set to 1. The result of this tendency can be seen in figure 8, in which the value τ is incremented in values of 0.00025 from 0.0 to 0.006, for an RP-Forest index structure which is configured with the parameters $leafSize = 4$ and $|L| = 200$. Therefore, the expected $voteWeight$ for all Q_i is $\frac{1}{4 \cdot 200} = 0.00125$[iii] and all values of τ in the range 0.00125 to 0.002499 produce identical $\mathbf{C}(q)$, equivalent to all the points which are encountered at least once during the search. Retaining only the points which are encountered at least twice during the search, results in a massive shift in recall value of 0.2. To

[iii] It is also possible the expected $voteWeight$ is $\frac{1}{3 \cdot 200} = 0.001667$, however this is not important for the overall argument

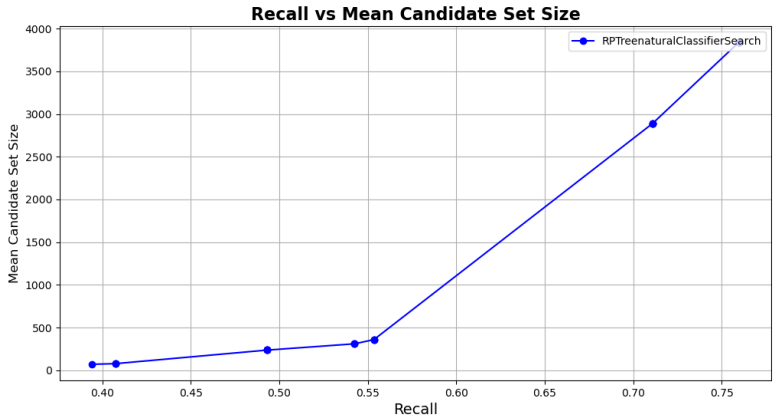

Figure 8: Plot of recall/candidate set size trade-off for the RPForest with Natural Classifier Search algorithm with $leafSize = 4$, $|L| = 200$ and τ ranging from 0.0 (right-most point) to 0.006 (left-most point) in increments of 0.00025 on the SIFT dataset. Note that multiple points are stacked on top of each other, as well as the large jump in recall for neighboring points stacks.

configure the algorithm to produce results in the range between these two recall values therefore entails modifying the underlying index structure. For the LSH index structure, this tendency is much less pronounced, as partition elements can provide a much larger variation in the number of contained corpus points and therefore also vary $voteWeight$ for each Q_i retrieved.

4.4 Quick-select Natural Classifier Search

The final search strategy to be detailed is a novel contribution from this thesis. The search strategy is derived from the Natural Classifier Search described above and can therefore be understood not as a completely novel search strategy, but as a variation of the Natural Classifier Search. The search strategy makes use of the same $\mathbf{C}'(q)$ construction approach and scoring function, $\text{AvgRelativeFrequency}(L, q, y)$, but utilizes a different search parameter, ν which provides fine-grained tuning to achieve a desired recall range for a fixed index structure L. The search strategy is motivated by the fact that it can be relatively difficult to set the algorithm parameters for the Natural Classifier Search, as has been described immediately above (see figure 8). The Quick-select Natural Classifier Search aims to remedy this difficulty as well as to provide a guarantee on the maximum number of distance computations carried out in the re-ranking step. This guarantee is achieved by selecting the ν points in the preliminary candidate set, $\mathbf{C}'(q)$, which have the highest score of $\text{AvgRelativeFrequency}(L, q, y)$. This refinement condition is motivated by the same assumption which is also the basis for the refinement condition in Natural Classifier Search, namely that

a higher value of AvgRelativeFrequency(L, q, y) for some point $y \in \mathbf{X}$, correlates to a higher likelihood that $y \in k\text{NN}(q)$. Therefore, it follows that increasing ν monotonically increases the recall quality while guaranteeing a maximum number of distance computations in the re-ranking step.

The selection of the ν points to be included in $\mathbf{C}(q)$ is, as the name suggests, achieved by utilizing the Quick-select algorithm. Quick-select is an algorithm used to identify the m-th greatest element in an array. The m-th greatest element is identified by partially sorting the array in a manner similar to the Quicksort algorithm. The Quick-select Natural Classifier Search strategy leverages two invariants of the Quick-select algorithm. The first invariant states that when the Quick-select algorithm has finished execution on an array, the m-th largest element is positioned at index $m-1$ and becomes the so-called pivot element. The second invariant states that all elements positioned at indices higher than the pivot element are lesser than or equal to the pivot element. This also means that in the case of multiple elements being equal to the pivot element in size, tie-breaks are broken evenly at random.

Quick-select Natural Classifier Search initially builds $\mathbf{C}'(q)$, in exactly the same way as the Natural Classifier Search. It subsequently invokes Quick-select on the preliminary candidate $\mathbf{C}'(q)$, with each element being scored according to the score function AvgRelativeFrequency(L, q, y), to identify the ν-th greatest element. The search strategy then leverages the two invariants described above, in order to construct a $\mathbf{C}(q)$, composed of all elements positioned at indices less than ν.

Formally, both the candidate set $\mathbf{C}'(q)$ and the scoring function AvgRelativeFrequency(L, q, y) are defined identically to the definitions in Natural Classifier Search (Section 4.3). Let $\mathbf{C}'(q)_{quick-select}$ denote the tuple corresponding to the preliminary candidate set $\mathbf{C}'(q)$, which has been modified by the Quick-select algorithm. Let $\mathbf{C}'(q)_{quick-select}[i]$ denote the corpus point at index i of $\mathbf{C}'(q)_{quick-select}$. The candidate set $\mathbf{C}(q)$ for the Quick-select Natural Classifier Search strategy can then be defined:

$$\mathbf{C}(q) = \{\mathbf{C}'(q)_{quick-select}[i] \mid 0 \leq i \leq \nu - 1\}$$

4.4.1 Setting Parameters for Quick-Select Natural Classifier Search

Given that the Quick-Select Natural Classifier Search strategy generates the exact same preliminary candidate set $\mathbf{C}'(q)$ and also uses the scoring function AvgRelativeFrequency(L, q, y) as the Natural Classifier Search, the approach to setting build parameters will most likely be identical between the two search strategies. However, it is worthwhile investigating the claimed fine-grained tuning that the quick-select candidate refinement provides. Figure 9 demonstrates an identical index structure to the one used to produce figure 8. There is a stark difference between the two plots that highlights the degree of fine-tuning that the Quick-select Natural Classifier provides for the tree-based index structures. Namely, for the Natural Classifier, it is not possible to produce searches for the recalls in the range between 0.56 and 0.77, while the Quick-select Natural Clas-

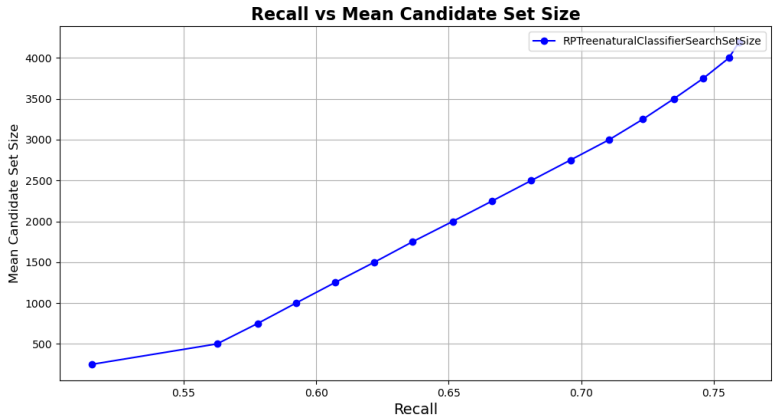

Figure 9: Plot of recall/candidate set size trade-off for the RPForest with Quick-select Natural Classifier Search algorithm with $leafSize = 4$, $|L| = 200$ and ν ranging from 4250 (right-most point) to 250 (left-most point) in increments of 250 on the SIFT dataset.

sifier is able to be configured to provide a large number of target recall values in the range and more importantly a wide range of candidate set sizes for a fixed index structure. Note that the fixed step size skews the plot somewhat for figure 9, as the step size should decrease as the recall approaches 0.0 in order to produce a plot more similar to figure 8.

The Quick-select Natural Classifier also makes it possible to perform a more in-depth analysis of how both itself and the Natural Classifier Search are impacted by index structures that generate larger partition elements. In section 4.3.2 it was argued that larger partition elements could lead to reduced precision. This fact is demonstrated in figure 10, which details the recall achieved by a value of ν of 200, for the RP-Forest index structures in which $|L|$ is fixed to 100 and $leafSize$ is increased from 2 to 4096. The figure clearly demonstrates that the ranking created by the AvgRelativeFrequency(L, q, y) scoring function, becomes less and less attuned to finding nearest neighbors as the partition element size is increased. Thus, the scoring function increasingly scores elements in high-density areas higher than elements that are likely neighbors to q as the partition element size is increased. It is important to note that as $leafSize$ is increased the computation of creating $\mathbf{C}(q)$ is also increased as the search strategy iterates over and scores an increasing number of points. Thus, in the example provided, above a $leafSize$ of 16 both the amount of computation is increased while the recall decreases for a fixed ν. This analysis indicates that both the Natural Classifier Search and the Quick-select Natural Classifier Search achieve the best performance on index structures which result in smaller partition elements that contain only the most 'relevant' corpus points, over index structures which result in larger partition elements as these contain corpus points which

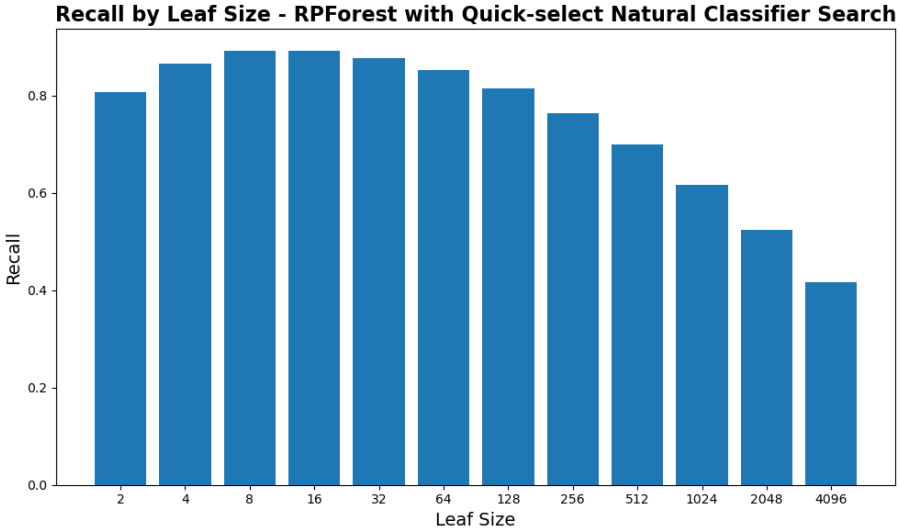

Figure 10: Plot of recall and $leafSize$ for the RP-Forest with Quick-select Natural Classifier Search algorithm on the Fashion-MNIST dataset. For all searches $|L| = 100$ and $\nu = 200$.

create a 'noise' effect on the ranking of examined points.

4.4.2 Quick-select Natural Classifier Search Implementation Details

The Quick-select Natural Classifier Search is quite similar in its implementation structure to the Natural Classifier Search strategy. It iterates over all $D_i \in L$, querying them one at a time. For all corpus points $x \in Q_i$ retrieved from the query, it iterates of the set of nearest neighbors $k\text{NN}(x)$, for each point $y \in k\text{NN}(x)$ incrementing the variable avgRelativeFrequency(y) by $\frac{1}{|Q_i| \cdot |L|}$. From this point, the search strategy differs from the Natural Classifier Search. It maintains a supplementary datastructure, which simply stores the encountered points and adds points to this datastructure the first time they are encountered during the search. When all iteration has finished, the supplementary datastructure is used to build an array of corpus points which the Quick-select algorithm is executed on, selecting the ν points of highest avgRelativeFrequency to build $\mathbf{C}(q)$. $\mathbf{C}(q)$ is then subjected to the brute-force search and the top k candidates are returned. The search strategy is described in pseudo-code in algorithm 15.

Again, an array of `float` is used to maintain the avgRelativeFrequency value of all points $x \in \mathbf{X}$, which is reset at the beginning of every search. In the implementation, the tuple B is maintained as a `LinkedList` which stores just the indices of encountered points. After the tallying of avgRelativeFrequency for all encountered points, an array of simple objects is built, where each object corresponds to an encountered point index and its' avgRelativeFrequency. For

Algorithm 6 Quick-select Natural Classifier Search

```
1:  procedure QUICKSELECTNATURALCLASSIFIERSEARCH(L, q, ν)
2:      B ← ()
3:      for D_i ∈ L do
4:          Q_i ← query(D_i, q)
5:          voteWeight ← 1 / (|Q_i| · |L|)
6:          for x ∈ Q_i do
7:              for y ∈ KNN(x) do
8:                  if avgRelativeFrequency(y) = 0.0 then
9:                      Add y to B
10:                 Increment avgRelativeFrequency(y) by voteWeight
11:     C(q) ← ∅
12:     Select ν-th largest element in B, selecting for avgRelativeFrequency
13:     for i ← 1 to ν do
14:         add C'(q)_{quick−select}[i] to C(q)
15:     return bruteForceSearch(C(q), q)
```

this process, the `LinkedList` B, is used to create the array in time which is linear in $|B|$ ($\mathbf{C}'(q)$), as each avgRelativeFrequency is retrieved in constant time. This array is then subjected to the Quick-select selection for the ν-th greatest value of avgRelativeFrequency and the first ν points are subjected to the brute-force search.

An important implementation detail of the Quick-select algorithm is the choice of pivot element. A naïve approach is to select either the first or last element in the sub-array as the pivot element. This approach leads to a worst-case performance of $\mathcal{O}(N^2)$ but an average running time of $\mathcal{O}(N)$, where N denotes the size of the array. To reduce the probability of worst-case executions, the pivot element in the sub-array is chosen uniformly at random [Dev01].

5 Build Parameter Assessment Study

Having detailed each search strategy it is possible to conduct a small study of the influence on both the recall, candidate set size, and the search time of the most important build parameters for the index structures that a search strategy can be coupled with. Given that a high-level discussion of each build parameter has already been covered in each relevant section, the following study covers only select parameters in more detail. Configuring index structures for a desired outcome is a non-trivial task, which is complicated further by interactions that occur with the search strategies that they are coupled with, and the fact that the configuration is for the most part also dataset-dependent. Therefore, the following study is vital to motivating the algorithm configurations in the following experiments.

In the following section, all build parameters are assessed under the base search strategy Lookup Search, as this search strategy relies entirely on the configuration of the underlying index structure, and therefore highlights its characteristics.

The first parameter to detail is one of the most central parameters for configuring the recall value of all the examined ensemble index structures. This is the parameter $|L|$ which for all ensemble index structures dictates the number of datastructures within an ensemble index structure L. Figure 11 shows a plot covering a range of values of $|L|$ for a fixed $leafSize$ of an RKD-Forest index structure. It is clear from the plot that increasing $|L|$ results in a higher recall value, at the cost of the search time, as a larger number of datastructures are queried and therefore also more corpus points are retrieved. It is worth noting that increasing $|L|$ also comes with increased memory consumption as more datastructures are created.

Typically, this increase in search time as $|L|$ is increased is dominated by the increase in candidate set size on which to carry out the re-ranking step, even though there is also an added computation of querying a larger number of datastructures. Figure 12 showcases the mean value of $|\mathbf{C}(q)|$ for the same range of $|L|$ and for the same index structure which is described in figure 11. It is relatively clear from these two plots, that the search time and the candidate set size are tightly correlated. The tendency showcased in figure 11 and 12 is similar for the two other ensemble-based index structures; the RP-Forest and LSH, where the RP-Forest the parameter $leafSize$ is fixed and for LSH the values of K and r are fixed while $|L|$ is increased.

Specifically for the LSH index structure, increasing r for fixed values of L and K results in a similar plot as figure 12. Figure 13 showcases a variation of the r parameter on an LSH index structure. It is relatively intuitive, that as the bucket size of each hash function $h_j \in g_i$ increases, the size of the partition elements increases as well, leading to a larger $|\mathbf{C}(q)|$. Furthermore, decreasing K for fixed values of L and r results in a similar tendency, which is also caused by increasing partition element sizes. This fact can also be deducted from figure 4, as it follows that if the collision probability of any point $x \in \mathbf{X}$ is increased with a reduction to K, $|\mathbf{C}(q)|$ can also be expected to increase. Despite the

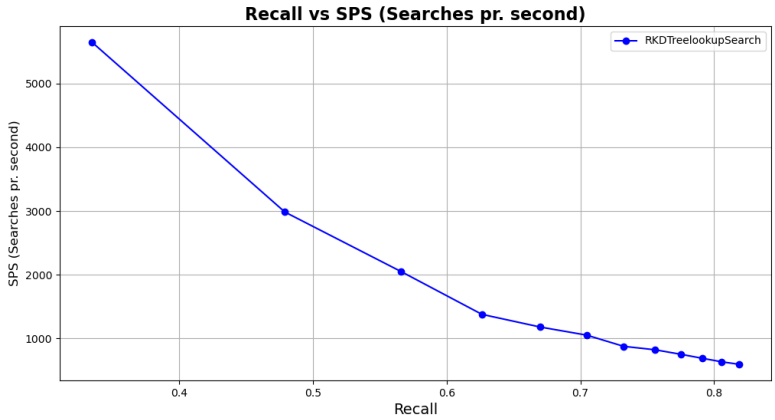

Figure 11: Plot of recall/query-time trade-off for the RKDTreeLookupSearch algorithm with $leafSize = 4$ and $|L|$ ranging from 25 to 300 in increments of 25 on the SIFT dataset.

authors in [DIIM04] focusing on c-ANNS, they still present recommendations for setting the values of K and r which are useful under the k-ANNS problem. The authors recommend a value of r which is in the neighborhood of four times the mean nearest neighbor distance and a value of K in the neighborhood of $K = \log_2 |\mathbf{X}|$. In the following experiments, these recommendations are used as a starting point for algorithm configuration under the Lookup Search strategy.

The final index structure to detail is the C2LSH index structure and specifically the build parameter t, which determines the collision threshold for points included in $\mathbf{C}(q)$. Figure 14 demonstrates certain properties of t, namely that as t is increased in relation to a fixed $partitionSize$ and K, the obtained recall is increased under the Lookup Search strategy. Naturally, this can be explained by the higher threshold including only more and more 'qualified' corpus points in $\mathbf{C}(q)$. As with other index structures, this higher recall comes at the cost of an increased search time. However, for previous index structures, the increased search time is mostly attributed to an increased $|\mathbf{C}(q)|$ which is not true for C2LSH. Figure 15 shows the same range of searches as figure 14 plotted against the candidate set size. It is clear that smaller and smaller candidate sets are used to achieve higher recalls - a tendency that is opposite in other index structures. Depending on where the stopping condition (line 17, algorithm 1) is checked, the value of $|\mathbf{C}(q)|$ can either be expected to decrease for a higher recall value if the check is made seldom, or be fixed to $partitionSize$ if checked every time a point is added to $\mathbf{C}(q)$.

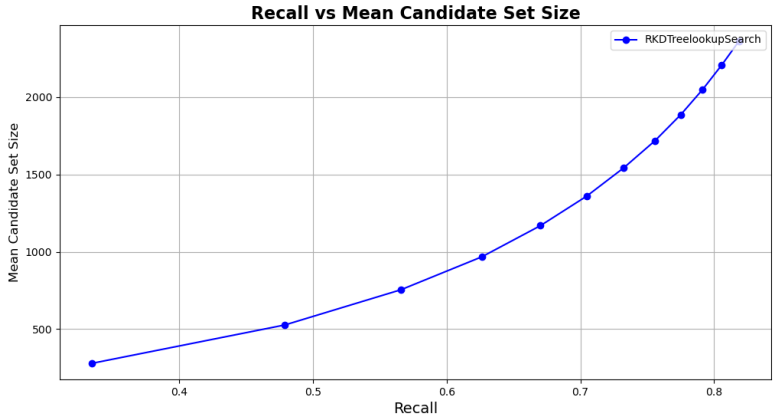

Figure 12: Plot of recall/candidate set size trade-off for the RKDForest with Lookup Search algorithm with $leafSize = 4$ and $|L|$ ranging from 25 to 300 in increments of 25 on the SIFT dataset.

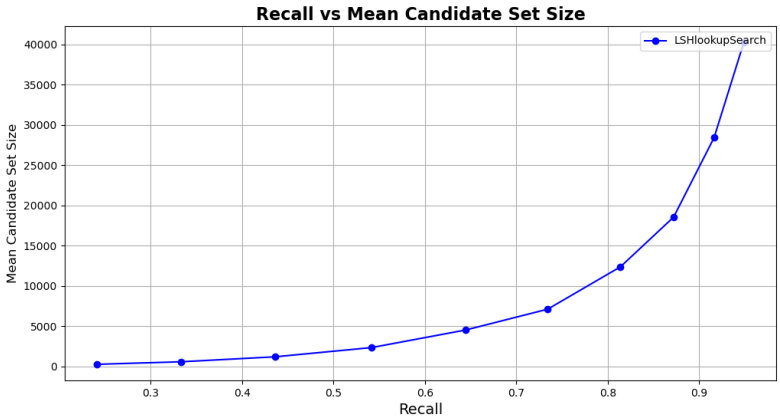

Figure 13: Plot of recall/candidate set size trade-off for the LSH with Lookup Search algorithm with $K = 20$, $|L| = 200$ and r ranging from 550 (left-most point) to 1000 (right-most point) in increments of 50 on the SIFT dataset.

Figure 14: Plot of recall/search-time trade-off for the C2LSH with Lookup algorithm with $K = 20$, $partitionSize = 200$ and t ranging from 3 (left-most point) to 18 (right-most point) in increments of 1 on the Fashion-MNIST dataset.

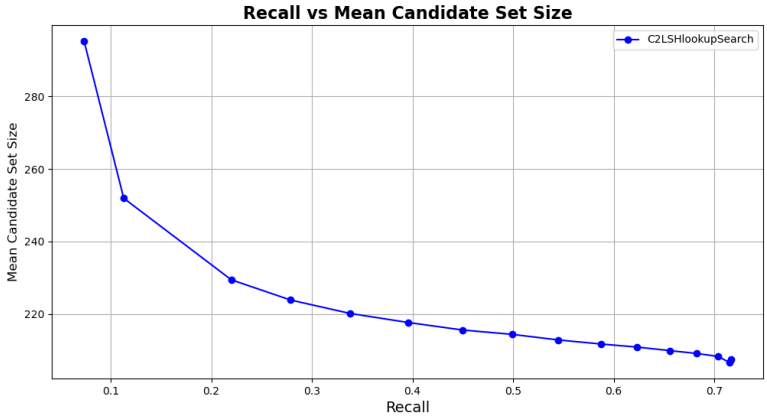

Figure 15: Plot of recall/candidate set size trade-off for the C2LSHLookup algorithm with $K = 20$, $partitionSize = 200$ and t ranging from 3 (left-most point) to 18 (right-most point) in increments of 1 on the Fashion-MNIST dataset.

k-ANNS Algorithm Overview

Index Structure	Search Strategy
RP-Forest	Lookup Search
RP-Forest	Voting Search
RP-Forest	Natural Classifier Search
RP-Forest	Quick-select Natural Classifier Search
RKD-Forest	Lookup Search
RKD-Forest	Voting Search
RKD-Forest	Natural Classifier Search
RKD-Forest	Quick-select Natural Classifier Search
LSH	Lookup Search
LSH	Voting Search
LSH	Natural Classifier Search
LSH	Quick-select Natural Classifier Search
C2LSH	Lookup Search
C2LSH	Natural Classifier Search
C2LSH	Quick-select Natural Classifier Search

Table 1: Overview of algorithms to experimentally evaluate. Note that the C2LSH index structure is not compatible with the Voting Search strategy.

6 Experiments

The following section will detail the experimental design and environment under which the experiments for the evaluation of each algorithm for k-ANNS take place. The terminology and general framework used to describe the experimental setup stems from [McG12].

The experiments consist of two rounds of experiments on a total of three datasets. In the first round of experiments, all algorithms corresponding to the 15 combinations of index structures and search strategies (see table 1) are evaluated across two datasets. The second round of experiments takes the best-performing algorithm for each index structure from round 1 and a so-called horse-race experiment is conducted on a hold-out dataset, in order to determine an overall best-performing algorithm for k-ANNS among the evaluated algorithms.

6.1 Algorithm Parameters

All algorithm parameters have previously been detailed in the sections above, but for the sake of clarity, an overview of these can be seen in table 2. Please note the distinction between those parameters which are treated as fixed parameters within the scope of these experiments and are therefore not configured from trial to trial and those which are treated as configurable parameters (experiment factors).

Build Parameters				
Index Structure	Experiment Factors			Fixed Parameters
RKD-Forest	$leafSize$	$\|L\|$		$o = 5$
RP-Forest	$leafSize$	$\|L\|$		$a = \sqrt{d}$
LSH	K	r	$\|L\|$	
C2LSH	K	$partitionSize$	t	$r = 1$ $c = 3$
Search Parameters				
Search Strategy	Experiment Factors			Fixed Parameters
Lookup Search				
Voting Search	τ			
Natural Classifier Search	τ			
Quick-select Natural Classifier Search	ν			

Table 2: Overview of algorithm parameters relating to each index structure and search strategy.

6.2 Performance Metrics

The algorithms[iv] in both rounds of experiments will primarily be evaluated according to the recall/search time trade-off performance metric, where recall is defined as described in section 2 and where the search time is given as the number of k-ANN searches executed pr. second (SPS)[v]. The time is measured in elapsed (wall clock) time as opposed to CPU time. This performance metric is chosen as the primary metric, as it provides a relevant evaluation of each algorithm for real-life purposes. Furthermore, the recall/search-time trade-off metric is commonly used to quantify the performance of k-ANNS algorithms [ABF18] making the results of this thesis comparable to other related studies. Another commonly used performance metric is the recall/distance computations trade-off metric, where the number of distance computations is defined as $|\mathbf{C}(q)|$, which will also be used for select analysis of experiment results. This performance metric has the benefit of being agnostic towards potential sub-optimal implementation choices made for both index structures and search strategies and can provide a deeper insight into the performance characteristics of each algorithm. Furthermore, the performance metric is also robust against uncontrollable noise parameters related to the Java Virtual Machine or background processes, which may impact the search-time performance indicator [Ses13]. The distance computations performance indicator is not used for the primary eval-

[iv] It is important to state that while the following sections refer to the evaluation of algorithms, more specifically what is being evaluated is the *implementation* of each algorithm. For the sake of simplicity each algorithm implementation will be referred to as 'algorithm'.

[v] Note that [ABF18] refers to this performance metric as *queries* pr. second (QPS). The term *searches* is used to avoid confusion with how the term query is used in this thesis.

Name	Dimensions	Corpus Size	Test size	Distance metric	Experiment
Fashion-MNIST	784	60,000	10,000	Euclidean	Round 1
SIFT	128	1,000,000	10,000	Euclidean	Round 1
GloVe-100	100	1,183,514	10,000	Angular	Round 2

Table 3: Overview of dataset instances.

uation of algorithms, as it doesn't take into account the cost of generating the preliminary candidate set $\mathbf{C}'(q)$ and the subsequent candidate refinement process, which can be computationally expensive operations. All performance and quality indicators are calculated as the average values obtained from executing searches of the entire test set for the specified input instance. For example, for the Fashion-MNIST dataset, each algorithm configuration is used to search 10,000 query points. The experiment results are calculated as the mean recall, search time and number of distance computations over all the 10,000 searches. Both the recall and SPS values presented for all experiments are calculated by the ANNBenchmarks tool, based on neighbor indices and search times which are reported in a compatible HDF5 file format produced by the benchmarking implementation of this thesis. Finally, it is important to note that all searches are for a k-value of 10, the reason being primarily that [HJR22a] and [ABF18] note similar results for differing values of k, meaning that the performance of the evaluated algorithms can reasonably be expected to generalize somewhat to other values of $k \neq 10$. Secondarily, experiments are only conducted for a single value of k to limit the scope of the required experiment trials and thereby increase the experimental efficiency.

6.3 Input Instances

The input instances for both rounds of experiments are from the public testbed used in the benchmarking suite, ANN-Benchmarks [ABF18]. Public testbed instances are used for the experiments, to maximize the reproducibility and relevance of the experiments by using datasets that are commonly used to evaluate algorithms for k-ANNS [McG12]. An overview of the input instances used in each round of experiments can be seen in table 3.

Considering the results of a suite of evaluated algorithms for k-ANNS presented in [ABF18] as well as on the website related to the benchmarking tool [ABF], these datasets can be considered to be ranked in order of least to greatest 'difficulty' in the following way:

- Fashion-MNIST
- SIFT
- GloVe

Name	CPU	Total Cores	Architecture	Memory
Node 1	2x Intel Xeon Gold 6242	64	x86	384 gigabytes
Node 2	2x Intel Xeon E5-2667 v4	32	x86	512 gigabytes

Table 4: Overview of HPC cluster nodes used for executing experiments

Thus, the notion of dataset 'difficulty' within the scope of this thesis, is simply based on the consensus of performance results across the algorithms for k-ANNS which are represented on [ABF]. While in [AC21] it is shown that it is possible to evaluate the difficulty of datasets according to other metrics, as well as to differentiate the difficulty of queries for each dataset, this is considered outside the scope of this thesis. Each dataset is split into a training set and a test set, where the training set is equivalent to the set of corpus points ($\mathbf{X}$), and the test set is a set of queries for which to report a kANN(q) solution given $\mathbf{X}$. Furthermore, each dataset also contains the 100 actual nearest neighbors of each query point, from which the recall value is calculated according to the definition in section 2.

6.4 Experiment Environment and Parallelization

All experiments were carried out in April and May of 2024 on the High-performance Computing (HPC) cluster of the IT-University of Copenhagen[vi]. The HPC cluster uses the Slurm Workload Manager to schedule jobs on the cluster, with the possibility of multiple jobs executing on the same compute node simultaneously. The experiments were executed on two separate compute nodes, with all experiments involving the Fashion-MNIST dataset being run on Node 1 and all experiments involving the SIFT and GloVe-100 dataset being executed on Node 2. See figure 4 for an overview of the technical specifications of each node.

Note that all experiment trials were restricted to a maximum of 96 gigabytes of Java heap memory, regardless of the specifications of the compute node. It was not possible to enforce single-threaded execution through containerization with Docker, as the software is incompatible with the Slurm Workload Manager. Therefore, single-threaded execution during the search phase is enforced through the omission of explicit parallelization and concurrency in the implementation. Furthermore, to enforce a greater certainty of single-threaded execution during the search phase, no external libraries are used for vector operations or statistical calculations, why some operations may be computed in a sub-optimal manner. Given that the experiments are not focused on index construction times, all index construction for ensemble index structures is parallelized through the use of the `ForkJoinPool` class from the Java standard library, such that the task of constructing the index structure is shared among all threads available to the program, but ensuring that every single datastructure D_i is constructed

[vi] More information on the ITU HPC cluster can be found at: http://hpc.itu.dk/

entirely by a single `Thread`. This provides a significant speed-up of the index construction, which is necessary for increasing the efficiency of running the experiments.

6.5 Experiment Design: Round 1

In order to evaluate the performance characteristics of each algorithm a suite of horse race experiments will be conducted, in which each index structure is paired with all four search strategies to determine the best-performing search strategy for each index structure. Each algorithm will be evaluated according to the fastest measured mean SPS above a threshold recall value of 0.8. This threshold is chosen to both limit the scope of the experiments and also because it is a common practice to evaluate algorithms in a competitive setting according to some recall threshold value [Sim]. The 0.8 recall threshold is a somewhat common threshold used to compare k-ANNS algorithm performance and is used amongst other places in select analysis in both [HJR22b] [HPT+16]. It is important to note that the results presented for a specific recall threshold do not necessarily generalize across all recall thresholds between 0 and 1. An overview of the combinations of index structures and search strategies which will be experimentally evaluated can be seen in table 1.

6.5.1 Algorithm Parameters - Fashion-MNIST

For the Fashion-MNIST dataset, the algorithm parameters tested are based on a full-factorial grid search of algorithm parameter combinations for some range of values for each parameter. For example, for evaluating the algorithm composed of the RP-Forest and Voting Search on the Fashion-MNIST dataset, all combinations of values for the following parameters were tested:

- $leafSize$ = 4, 8, 16, 32, 64, 128, 256
- $|L|$ = 195, 185, 175, 165, 155, 145, 135, 125, 115, 105, 100, 95, 90, 85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35, 30, 29, 27, 25, 23, 21, 19, 17, 15, 13, 11, 9, 7, 5, 3, 1
- τ = 20, 18, 16, 14, 12, 10, 9, 8, 7, 6, 5, 4, 3, 2

This results in a total of $3,920$ experiment trials, with each trial consisting of an execution of $10,000$ searches, each search executed for a different query vector q. The parameter ranges and interval sizes were defined based on limited preliminary testing to identify relevant minimum and maximum values. Furthermore, the tested ranges and interval sizes also reflect a balance of providing wide coverage and efficiency of running the grid search.

6.5.2 Algorithm Parameters - SIFT

The grid-search approach to setting parameters is only feasible within the scope of this thesis for the 'easiest' dataset Fashion-MNIST, due to both the increased

index construction and search times related to the larger datasets. Therefore, for the SIFT dataset, an experimental set-up typically used in competitions for k-ANNS was adopted. For each of the algorithms described above, a maximum of 30 algorithm configurations were tested, where a single configuration is comprised of a set of build and search parameters. The tested configurations for each algorithm were based on knowledge acquired from the Fashion-MNIST experiments (see section 7.1) as well as the discussion in section 5. The results obtained from the grid-search parameterization conducted on the Fashion-MNIST dataset provided valuable insight into the characteristics of the best-performing configurations of the search strategies Voting, Natural Classifier and Quick-select Natural Classifier Search, relative to the best-performing configuration of the baseline strategy of Lookup Search. The configurations were not set blindly in the sense that during experimentation, information related to the performance of all previously executed searches as well as the distribution of corpus points was used to determine algorithm parameters. The results obtained from this experimental setup for the SIFT dataset, are therefore also to a higher degree than for the Fashion-MNIST dataset, impacted by my ability to configure each algorithm optimally for the given recall threshold.

6.6 Experiment Design: Round 2

Following the first round of experiments a second round will be conducted on the GloVe dataset. This second round of experiments will be based on picking the best-performing search strategy for each index structure from the first round of experiments. The chosen k-ANNS algorithms will then be evaluated against the most difficult of the three datasets, GloVe. The GloVe dataset has been kept as a hold-out dataset for the entire development and experimentation, meaning no previous searches were carried out for the dataset. Based on the performance of each algorithm on this dataset, as well as the two previous datasets (Fashion-MNIST and SIFT), recommendations for the overall best-performing algorithm can be identified. For the second round of experiments, algorithm parameters are set in the same way as for the SIFT dataset, such that a maximum of 30 sets of algorithm parameters are tested for each algorithm. The algorithm parameters are set once again with access to information on results from previous experiments and searches as well as statistics about the distribution of $\mathbf{X}$. Please note that for the second round of experiments, the LSH index structure follows the scheme for LSH under the angular distance metric presented in section 3.1.2.

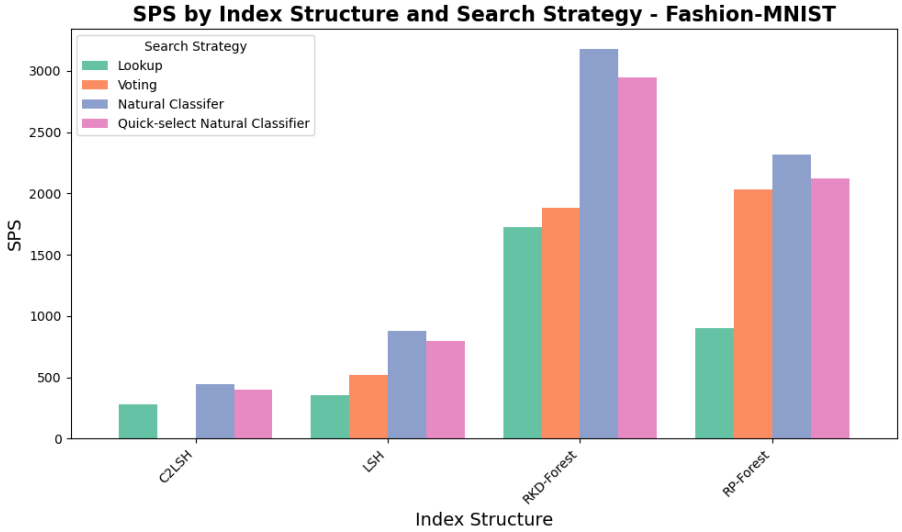

Figure 16: Overview best-achieved SPS by index structure and search strategy for a recall of 0.8 or greater obtained from the Fashion-MNIST dataset. Raw results are detailed in appendix B, table 9. Higher is better.

7 Experiment Results

7.1 Fashion-MNIST Results

In the first round of experiments, for the Fashion-MNIST dataset, the Natural Classifier Search strategy achieved the best performance across all four index structures, while Lookup Search achieved the worst performance (see figure 16). The Quick-select Natural Classifier strategy consistently performed second-best for all index structures with Voting Search performing third-best. Thus, the ordering of the performance of each search strategy is consistent for all four index structures. The overall best-performing algorithm for the Fashion-MNIST dataset was the RKD-Forest with Natural Classifier Search.

Considering again Lookup Search as a baseline strategy, the largest percentwise increase achieved by the Natural Classifier Search for any index structure was for the RP-Forest, where the Natural Classifier Search provided a 157.27% increase in SPS over Lookup Search (see table 5). The smallest increase achieved by the Natural Classifier Search strategy was for the C2LSH index structure, where it provided an increase in SPS of 60.70%. For the RP-Forest index structure, an almost comparable increase in SPS to the one achieved by the Natural Classifier Search strategy was achieved for Voting Search. The Voting Search strategy increased the SPS by 125.2% over the Lookup Search strategy, which compared to the 157.27% increase achieved by Natural Classifier Search, is just a 32.07 percentage point difference. In [HJR22b], the authors likewise find that

SPS relative to Lookup Search - Fashion-MNIST

	Lookup	Voting	Natural Classifier	Quick-select Natural Classifier
RP-Forest	100%	225.2%	**257.27%**	235.74%
RKD-Forest	100%	109.1%	**184.46%**	170.78%
LSH	100%	145.94%	**245.66%**	222.13%
C2LSH	100%	-	**160.79%**	143.17%

Table 5: Best-achieved SPS relative to Lookup Search for each index structure, for a recall of 0.8 or greater obtained from the Fashion-MNIST dataset.

the Voting Search and Natural Classifier Search strategies achieve similar performance on the RP-Forest when tested against the Fashion-MNIST dataset, with the Natural Classifier only slightly outperforming Voting Search (see appendix B, figure 21). For the RKD-Forest, the Voting Search strategy provided only a very slight performance increase over the Lookup Search strategy, with only a 9.1% increase in SPS. This is a difference in SPS which can arguably be considered statistically insignificant in the context of k-ANNS. This particular result on the Fashion-MNIST dataset is not immediately consistent with the results documented in [HJR22b], in which a much larger relative improvement to search time is reported.

Among the reported results, it is especially relevant to highlight that the second-largest increase achieved by the Natural Classifier Search strategy was for the LSH index structure, for which it achieved a 145.66% increase over the Lookup Search strategy. Similarly, for the C2LSH index structure, the Natural Classifier Search increased the performance over the Lookup Search strategy by 60.79%. For the LSH index structure, the Voting Search strategy also provides a lesser, but significant performance increase over the Lookup Search strategy of 45.94%. Thus, the results from the Fashion-MNIST dataset indicate that the locality-sensitive hashing-based index structures benefit rather considerably from the Natural Classifier Search strategy compared to the Lookup Search strategy, which is typically used for algorithms using these index structures. To a lesser degree, the LSH index structure also benefits from the Voting Search strategy on the Fashion-MNIST dataset.

7.1.1 Parameter Evaluation - Fashion-MNIST

Other than just the results showcased above, it is also worthwhile investigating the algorithm configurations that produce the best performance, to motivate the algorithm parameters used in following experiments. In appendix B, table 10 the configurations used to achieve the best performance above the 0.8 recall threshold are detailed. As a first point, it is necessary to highlight the fact that the recommendations of values for K and r for the LSH index structure under the Lookup Search strategy outlined in [DIIM04], were indeed shown to produce

the best performance on the Fashion-MNIST dataset. Among the corpus points in the dataset, the mean k-nearest neighbor distance is roughly 1000.0, and the value $\log_2 |\mathbf{X}| = 15.87$. Thus, the best performing configuration of LSH with Lookup Search of $r = 4000.0$ and $K = 15$ matches the recommendations of r being set to four times the nearest neighbor distance and K being set to roughly $\log_2 |\mathbf{X}|$.

Compared to the Lookup Search strategy, all the best results achieved with the Voting Search strategy are based on an index structure configured to induce larger partition elements. For example, for the LSH index structure, the best configuration for Lookup Search was $K = 15$, $r = 4000.0$, $|L| = 75$ while the best configuration for Voting Search was $K = 15$, $r = 6000.0$, $|L| = 50$. Given that r is increased by 50%, the set Q_i returned from querying each datastructure $D_i \in L$ can be expected to be much larger for the second configuration. Thus, the results obtained from the Fashion-MNIST dataset support the second strategy hypothesized in 4.2.2 for setting build parameters for Voting Search.

In appendix B, table 10 it is also clear that there is support for the strategy to constructing an ensemble index structure for the Natural Classifier Search described in section 4.3.2. Namely, the same recall is achievable for a similarly configured index structure but with a significantly smaller value of $|L|$ than is necessary for Lookup Search. For example, the best configuration of the RP-Forest with Natural Classifier Search uses an identical value of $leafSize$ but a value of $|L|$, which is 50% less than what is used for Lookup Search. Furthermore, while not visible from the table, for LSH the configuration $K = 15$, $r = 4000.0$, $|L| = 35$ performed very similarly to the best-performing configuration (805 SPS), which is an identical index structure to the best-performing index structure for Lookup Search, but with a more than 50% reduction to $|L|$.

For the C2LSH index structure, it is clear that the best-performing configuration for the Natural Classifier Search strategy is very similar to the best-performing index structure under Lookup-search, but for a much smaller value of $partitionSize$[vii]. It is also worth noting, that for the C2LSH index structure coupled with the Natural Classifier or Quick-select Natural Classifier Search strategies, the best-performing algorithm configurations were all for values of τ and ν where effectively no candidate refinement took place. This is unlike any of the other three index structures, where when coupled with the Natural Classifier Search, algorithm configurations with no candidate refinement consistently performed worse than configurations with candidate refinement.

Finally, for all four index structures a very similar set of build parameters as those used for the regular Natural Classifier Search was used to achieve the best performance for Quick-select Natural Classifier Search. This is expected, as the search strategies are essentially selecting the candidate set $\mathbf{C}(q)$ according to the same score function and ranking of points in $\mathbf{C}'(q)$, and only differ in the way the selection is carried out.

[vii] While not visible in appendix B, table 10, the configuration $K = 14$, $partitionSize = 1200$, $t = 10$ performed identically to the best performing configuration.

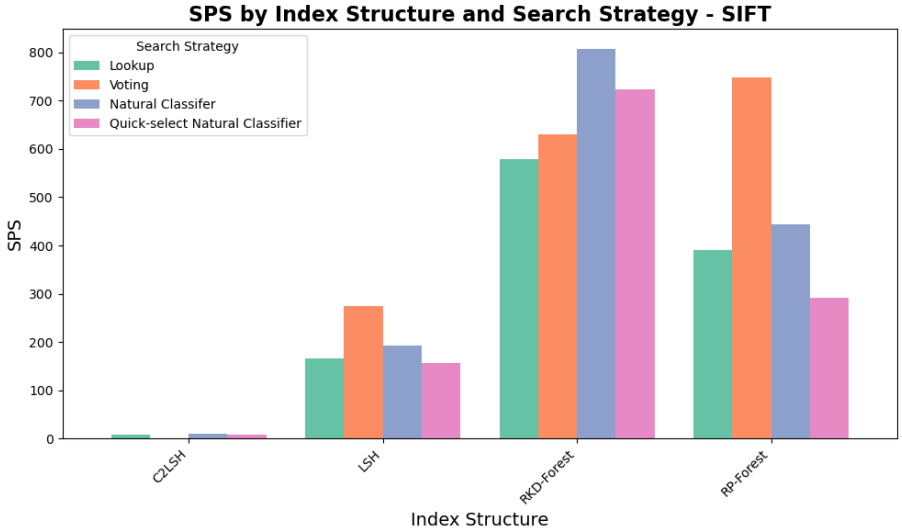

Figure 17: Overview best-achieved SPS by index structure and search strategy for a recall of 0.8 or greater obtained from the SIFT dataset. Raw results are detailed in appendix B, table 11. Higher is better.

7.2 SIFT Results

For the results obtained from the experiments that were executed on the SIFT dataset, the tendency of the best-performing search strategy for each index structure is much less consistent than for the Fashion-MNIST dataset. Figure 17 details the best achieved SPS for each tested k-ANNS algorithm. For the RKD-Forest, the best performance was achieved by the Natural Classifier Search, while the second best was achieved by the Quick-Select Natural Classifier Search, followed by Voting Search. Thus, for the RKD-Forest, the ranking of search strategies was consistent between the two datasesets. For the C2LSH index structure, the Natural Classifier was also the best-performing search strategy, while the Quick-select Natural Classifier Search achieved a worse performance than Lookup Search. It is worth noting, however, that the C2LSH achieved extremely poor performance for all search strategies when compared to other index structures. Perhaps most interestingly, for both the LSH and RP-Forest index structures, the Voting Search strategy achieved the best performance while the Natural Classifier Search strategy performed second best. For both these index structures, the Quick-select Natural Classifier performed worse than the Lookup Search strategy. Thus, for these two index structures, the ranking of search strategies differs completely from the Fashion-MNIST dataset. The overall best-performing algorithm for the SIFT dataset was once again the RKD-Forest with Natural Classifier Search, however, the RP-Forest with Voting Search algorithm achieved very similar performance.

The largest percent-wise increase in performance over the Lookup Search strategy was achieved by Voting Search for the RP-Forest index structure, where it achieved a 91.56% increase in SPS (see table 6). The search strategy that achieved the worst performance relative to the Lookup Search strategy was the Quick-Select Natural Classifier, for which the performance decreased by 25.32% and 6.02% for the RP-Forest and LSH index structures respectively.

Considering the RP-Forest and LSH index structures for which the ranking of search strategies is inconsistent with the results obtained from the Fashion-MNIST dataset, the fact that the Voting Search outperforms the Natural Classifier Search (and Quick-select Natural Classifier Search) can be attributed mainly to a decrease in performance of the Natural Classifier Search rather than an increase by the Voting Search. The Natural Classifier Search an increase in performance over the Lookup Search of just 13.55% and 16.27% for the RP-Forest and LSH index structures respectively, while for the Fashion-MNIST dataset, the increases were 157.27% and 145.66%. On the Fashion-MNIST dataset, the Voting Search strategy saw increases of 125.2% and 45.94% over the Lookup Search for the aforementioned index structures, while for the SIFT dataset, the increase was 91.56% and 65.06% respectively. For the RKD-Forest with Natural Classifier Search, the increase in performance over the Lookup Search also fell from 84.46% to 39.55% from the Fashion-MNIST dataset to the SIFT dataset, while the same number for RKD-Forest with Voting Search was virtually unchanged. Thus, in general, the relative performance of the Natural Classifier fell drastically, while for the Voting Search, the relative performance remained relatively unchanged between the two datasets. These findings indicate that regardless of the index structure, the Voting Search is more robust in its performance against the increased difficulty of the dataset than the Natural Classifier Search strategy is.

Even though the authors do not evaluate their implementation directly against the GloVe dataset, the ranking of the search strategies on the RP-Forest is not immediately consistent with the results presented in [HJR22b]. Here, the authors find that the Natural Classifier Search performs better than Voting Search when coupled with the RP-Forest for most datasets and recall values (see appendix B, section 21). However, it is worth noting that on two of the evaluated datasets (GIST and Trevi), the Voting Search strategy outperforms the Natural Classifier Search in high recall ranges (above 0.9 for GIST) when coupled with the RP-Forest. Therefore, it is not possible to exclude the possibility that had the algorithm been evaluated against the GloVe dataset or a dataset of similar difficulty, the authors would have found similar results to those presented in this thesis. This theory is especially relevant considering that in [HJR22b], the GIST dataset on which the authors carry out their experiments, is a reduced version of the dataset consisting of 100,000 corpus points and is therefore considerably 'easier' than the version of the dataset used in [ABF18], which consists of 1,000,000 corpus points in the same dimensionality (960).

A more nuanced picture of the performance of the four search strategies arises when considering their performance according to the recall/candidate set

SPS relative to Lookup Search - SIFT

	Lookup	Voting	Natural Classifier	Quick-select Natural Classifier
RP-Forest	100%	**191.56%**	113.55%	74.68%
RKD-Forest	100%	108.81%	**139.55%**	125.04%
LSH	100%	**165.06%**	116.27%	93.98%
C2LSH	100%	-	122.22%	88.89%

Table 6: Best-achieved SPS relative to Lookup Search for each index structure, for a recall of 0.8 or greater obtained from the SIFT dataset.

size trade-off performance metric. The best-achieved candidate set size $|\mathbf{C}(q)|$ for a recall above 0.8 for each tested algorithm is detailed in figure 18. The figure shows that for the LSH index structure, the Natural Classifier Search achieved a candidate set size that is almost identical to the Voting Search, while the Quick-select Natural Classifier succeeded in achieving a smaller candidate set size, despite both search strategies being considerably slower for the specified recall. Intuitively it could be expected that they would also perform similarly under the recall / SPS metric, which has clearly been demonstrated to not be the case. This indicates that the Natural Classifier Search and Quick-select Natural Classifier Search perform worse than Voting Search for the LSH index structure, not because they don't succeed in building a qualified candidate set, but because the task of building it requires a disproportionate amount of computation. This underlines the importance of balancing the complex task of building a qualified candidate set, but doing so without significant computational overhead - a point which was also discussed in section 2.1. This balance is also pertinent to discuss in relation to the fact that the Quick-select Natural Classifier consistently achieves a smaller candidate set, despite also achieving worse SPS than the Natural Classifier. This is true for all index structures on both the Fashion-MNIST and SIFT datasets. This illustrate the fact that the candidate refinement process to a higher degree succeeds in building a minimal qualified candidate set, but that the computational overhead of performing the selection (algorithm 15, line 12) is too great to justify the reduction to the candidate set size which it achieves. For the RP-Forest index structure, however, the Voting Search strategy also succeeds in building a candidate set for the recall threshold, which is significantly smaller than the best achieved by the Natural Classifier Search. This clearly indicates that on the more difficult SIFT dataset, the RP-Forest index structure with the Voting Search strategy is much better suited for the task of k-ANNS, even when considering both performance metrics.

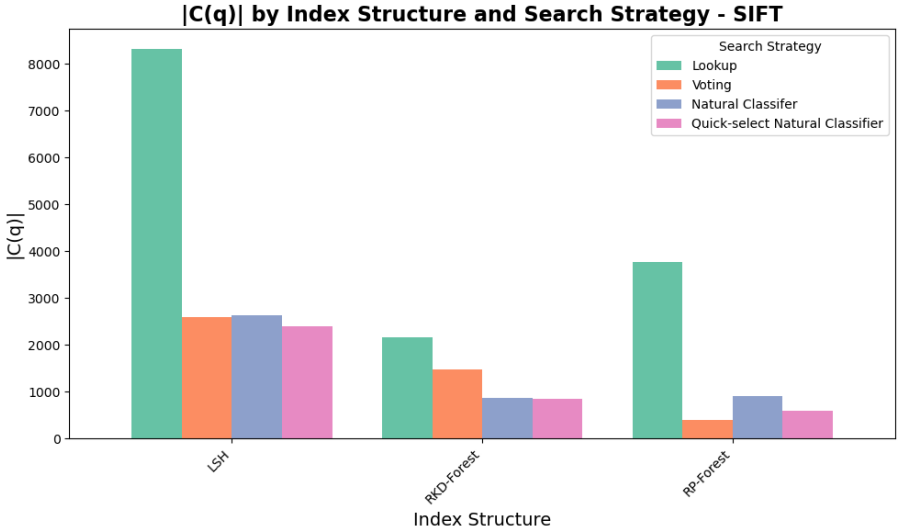

Figure 18: Overview best-achieved $|\mathbf{C}(q)|$ by index structure and search strategy for a recall of 0.8 or greater obtained from the SIFT dataset. Lower is better. For the sake of readability, the C2LSH index structure is not included in this figure. For a figure including C2LSH see appendix B, figure 22

7.2.1 Parameter Evaluation - SIFT

Given the central difference in the experimental design between the two sets of experiments in the first round - namely that for the Fashion-MNIST dataset a grid-search approach to algorithm configurations was taken, while for the SIFT dataset, the algorithms were tested for a limited number of configurations - it is highly pertinent to discuss the algorithm parameters used to obtain the best-performing results for the SIFT dataset. An overview of these is found in appendix 1, table 12.

Considering first the RKD-Forest for which the ranking was identical for both datasets, we also find that the approach to configuring the index structures for the Natural Classifier Search discussed in section 4.3.2 and section 7.1, produced the best performance. Namely, compared to the best performing Lookup Search configuration ($leafSize = 16$, $|L| = 310$), to fix the build parameters which dictate the partition element size (in this case $leafSize$), reduce$|L|$ and use τ to reach the desired recall. While this is not immediately clear, given the best index configuration used with the Natural Classifier was $leafSize = 4$, $|L| = 274$ (thus a lower value of $leafSize$), it has to be considered that the best performing RKD-Forest for Lookup Search, would most likely be a configuration of $leafSize = 4$ and a very high value $|L|$ and that a second-best configuration of $leafSize = 8$ and a lower value of $|L|$. However, both of these configurations were not possible with the given memory restrictions.

Turning to LSH, the above strategy for building the index structure for the Natural Classifier Search again produced the best performance for the specific algorithm but did not succeed in achieving better performance than LSH coupled with Voting Search. We have that LSH coupled with Lookup Search configured as $K = 20$, $r = 755.0$, $|L| = 400$ achieved an SPS of 166, while coupled with Natural Classifier Search configured as $K = 20$, $r = 750.0$, $|L| = 225$ and $\tau = 0.0003$ achieved just 193 SPS, despite a much lower value of $|L|$. Naturally, other configurations of the LSH with Natural Classifier Search were also tested, the most relevant of which are detailed in B, table 13 along with their achieved performance. As is clear from table 13, configurations that also do not follow the described strategy for setting parameters were tested but none succeeded in achieving any better performance. Similarly, for the RP-Forest with Natural Classifier Search, no other configurations than those following the above-described strategy were able to produce any better performance. Even so, the RP-Forest with Natural Classifier Search was not able to outperform the RP-Forest with Voting Search algorithm. Thus, with relative confidence it can be concluded that the experiment results are likely to reflect inherent properties of the evaluated search strategies and are less likely to reflect sub-optimal algorithm configuration.

7.3 GloVe Results

Before covering the results obtained from the experiments on the GloVe dataset, it is necessary to first detail the basis for which each algorithm in this set of experiments was chosen. An overview of the chosen algorithms can be seen in table 7. For the RKD-Forest index structure, across both the Fashion-MNIST and the SIFT dataset, the Natural Classifier Search strategy achieved the best performance. Therefore the RP-Forest coupled with Natural Search Classifier algorithm is selected for this final experiment. For the RP-Forest, the Natural Classifier Search strategy performed best on the Fashion-MNIST dataset, while the Voting Search strategy performed best on the SIFT dataset. Therefore both the algorithm consisting of the RP-Forest coupled with Voting Search and Natural Classifier Search are tested against the GloVe dataset. For the LSH index structure, the Natural Classifier Search likewise performed best on Fashion-MNIST and Voting Search performed best on SIFT. Therefore, both algorithms are selected for the GloVe dataset. For the C2LSH index structure, across both the Fashion-MNIST and SIFT datasets, the Natural Classifier Search achieved the best performance and is therefore included in the final set of experiments.

For the C2LSH index structure, the otherwise fixed parameter value of $r = 1$ created extremely large partition elements on the GloVe dataset, due to the much lower mean nearest neighbor distance compared to the two other datasets. Therefore, for the experiments on the GloVe dataset, the build parameter r was set to 0.001, which induced partition elements containing roughly the same number of corpus points as for $r = 1$ on the SIFT dataset. Even so, it was not possible to configure the C2LSH with Natural Classifier Search algorithm to produce a recall above the 0.8 threshold within a time limit of 1 hour total

Algorithm	
Index Structure	**Search Strategy**
RKD-Forest	Natural Classifier Search
RP-Forest	Voting Search
RP-Forest	Natural Classifier Search
LSH	Voting Search
LSH	Natural Classifier Search
C2LSH	Natural Classifier Search

Table 7: Overview of the algorithms evaluated against the GloVe dataset.

running time for a single experiment trial. Thus, the algorithm is not considered feasible under this experimental design and is deemed highly unlikely to outperform the other five evaluated algorithms. The best-achieved result was a recall of 0.54764 for an SPS of 3.412. Therefore, the C2LSH with Natural Classifier algorithm is not represented in the following results.

After experiments on the GloVe dataset were concluded it was discovered that the implemented benchmarking tool that was used during experimentation consistently reported a recall value that was 0.02-0.03 lower than the one calculated by the ANNBenchmarks tool, due to rounding differences. This difference only occurred for the GloVe dataset, due to the much lower distance values than the previous datasets. Thus, each algorithm was evaluated for a minimum recall threshold slightly higher than 0.8. However, since the reporting is consistent for all algorithms, the relative performance is still preserved and the difference is not considered to have a significant impact on the achieved results.

The results of the experiments on the GloVe dataset are shown in figure 19, as well as in table 8. As can be seen from figure 19, for both the RP-Forest and LSH index structures the Voting Search strategy outperformed the Natural Classifier Search strategy. While these results are consistent with the findings on the SIFT dataset, the difference in performance for the two search strategies is greatly exacerbated on the GloVe dataset, in which the Voting Search strategy achieved a 151,49% and 358,27% performance gain over the Natural Classifier Search for the RP-Forest and LSH index structure respectively. On the SIFT dataset, these performance increases were just 41.97% and 68.96%. Furthermore, on the GloVe dataset, the RKD-Forest with the Natural Classifier Search algorithm performed the worst of all five evaluated algorithms, despite achieving the best performance overall for both the Fashion-MNIST and SIFT datasets. The best-performing algorithms on the GloVe dataset were the LSH with Voting Search and RP-Forest with Voting Search algorithms, which achieved almost identical performance. The fact that these two algorithms perform identically, is somewhat surprising, given that the RP-Forest with Voting Search outperformed the LSH with Voting Search algorithm by a factor of 3.89 and 2.73 on the Fashion-MNIST and SIFT datasets respectively. However, it must be stated

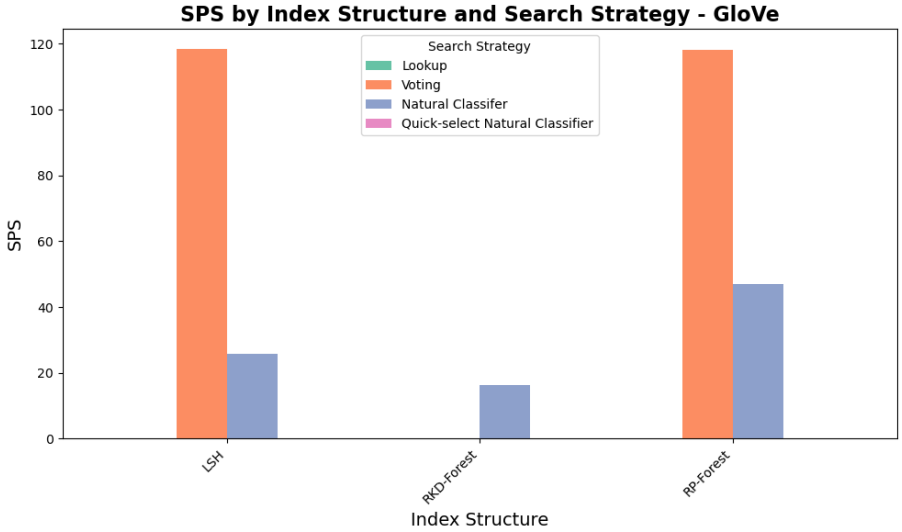

Figure 19: Overview best-achieved SPS by index structure and search strategy for a recall of 0.8 or greater obtained from the GloVe dataset. Higher is better.

again2 that the LSH index structure for the GloVe dataset, utilizes the simplified hashing scheme described in section 3.1.2 and that this fact may also contribute to the increased performance of the LSH with Voting Search algorithm.

One likely factor that can, at least partly, explain the overall poor performance of the Natural Classifier Search strategy, is the fact that the difficulty of the GloVe dataset seemingly necessitates index structure configurations which are not possible under the experimental environment, in which a maximum of 96GB of Java heap memory was allocated. For both the RP-Forest and RKD-Forest, the best-performing configurations for the Natural Classifier Search were the minimal values of $leafSize$ which were possible under these memory restrictions. For both index structures, these values were considerably higher than the best-performing $leafSize$ for the two previous datasets (see appendix B, table 14). Reiterating on the study and discussion presented in section 4.4.1, for larger than optimal partition element sizes the Natural Classifier Search strategy becomes worse at ranking the preliminary candidate set according to distance to q, while requiring significantly more computation. Considering this, it is very likely that memory is a limiting factor to the performance of the tree-based index structures with the Natural Classifier Search on more difficult datasets, as it is only possible to configure index structures that are sub-optimal to the performance of the Natural Classifier Search. This is a situation that was most likely not encountered in [HJR22b], as the largest dataset consisted of only 100,000 corpus points. Furthermore, the authors have implemented their algorithms in the C++ programming language which is likely to use significantly less memory than the Java implementation tested in this thesis.

Algorithm		
Index Structure	**Search Strategy**	**SPS**
RKD-Forest	Natural Classifier Search	16.22
RP-Forest	Voting Search	**118.05**
RP-Forest	Natural Classifier Search	46.94
LSH	Voting Search	**118.57**
LSH	Natural Classifier Search	25.76
C2LSH	Natural Classifier Search	-

Table 8: Overview of the algorithms evaluated against the GloVe dataset.

However, given that on the SIFT dataset, no such memory limitations were encountered for the tested configurations and that the Voting Search strategy still outperformed the Natural Classifier Search strategy for the RP-Forest and LSH index structures, it is likely that the results obtained from the GloVe dataset are, to a higher degree, a reflection of general properties of the two search strategies. Namely, that inherently the Voting Search strategy is more robust against increasingly difficult dataset instances than the Natural Classifier Search strategy is. This tendency is especially pronounced for the RP-Forest and LSH index structures.

This hypothesis is further substantiated in figure 20, which showcases the candidate set size of the best-performing algorithm configurations. It is evident from this plot, that the RKD-Forest with Natural Classifier Search fails dramatically at producing a minimal, qualified candidate set. In fact, the candidate set for the best-performing configuration consists of more than 8% of the full corpus. Similarly, the best-performing configurations of the Natural Classifier Search for both the LSH and RKD-Forest index structures, construct a candidate set that is many times larger than for the Voting Search strategy.

7.4 Algorithm Recommendations

Based on the results from the experiments and the accompanying analysis and discussion, among the 15 evaluated algorithms, the following section will provide recommendations for the optimal choice of algorithms for k-ANNS. Considering the above-described variation of both the ranking of search strategies for each index structure as well as the best-performing algorithm for the three evaluated datasets, it is not possible to recommend a single algorithm for all datasets. There are strong indications in the presented results, that the optimal algorithm depends on the general 'difficulty' of the dataset for which it is applied.

First of all, the RKD-Forest with Natural Classifier Search achieved the best performance of any evaluated algorithm on both the Fashion-MNIST and SIFT datasets. Therefore, this study recommends the algorithm as the optimal choice for datasets of easy to medium difficulty. However, it cannot be recommended

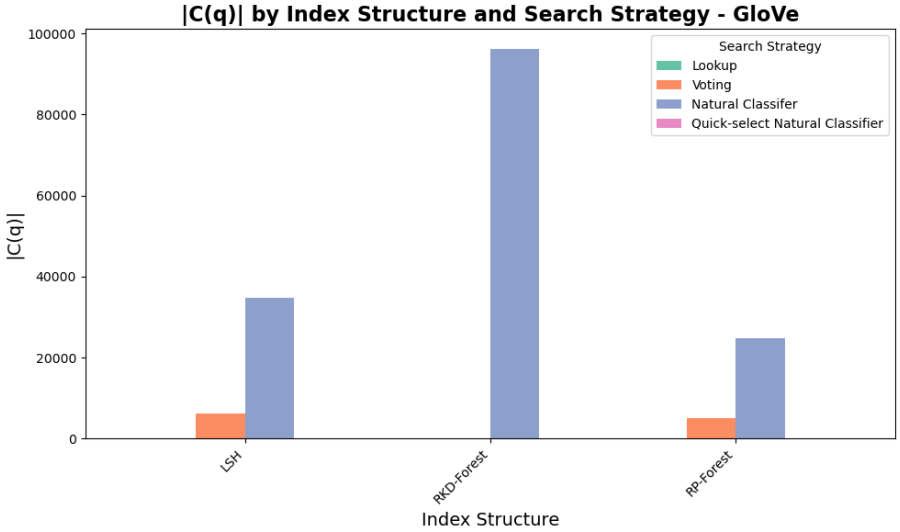

Figure 20: Overview of $|\mathbf{C}(q)|$ for the best-performing algorithm configuration under the SPS / recall trade-off metric from the GloVe dataset. Lower is better.

for datasets that are more difficult than the SIFT dataset.

The RP-Forest coupled with the Voting Search strategy performs very well for both the SIFT and GloVe-100 datasets, indicating that generally, it is the best choice of algorithm for difficult datasets. While the RKD-Forest with Natural Classifier Search slightly outperforms it for the SIFT dataset, and the LSH with Voting Search algorithm performs equally well on the GloVe-100 dataset, the RP-Forest with Voting Search algorithm performed very well across both, while the two other algorithms were only able to perform well on a single of the two datasets. Furthermore, it performed reasonably well on the easiest of the three datasets, Fashion-MNIST. Therefore, among the evaluated algorithms, it is recommended for datasets of medium to high difficulty as well as a good choice of a general-purpose algorithm for k-ANNS on datasets of unknown difficulty.

While it is unclear which algorithms are likely to achieve the best performance for datasets with a difficulty beyond the GloVe-100 dataset, the suggested choices to evaluate first would be both the LSH and RP-Forest coupled with Voting Search as these performed best on the most difficult dataset which was tested.

Finally, when considering more qualitative aspects of the evaluated algorithms, the RKD-Forest coupled with the Quick-select Natural Classifier Search strategy performs almost as well as the RKD-Forest with Natural Classifier Search across all datasets. Therefore, it is recommended in scenarios where a single index structure is needed to support a larger range of recall quality, however, only for datasets of low to medium difficulty.

8 Conclusion

This thesis has detailed and implemented the following four index structures for k-Approximate Nearest Neighbor Search; LSH, C2LSH, RKD-Forest, and RP-Forest, as well as the three search strategies; Lookup Search, Voting Search, and Natural Classifier Search. Furthermore, a novel search strategy, Quick-select Natural Classifier Search was introduced. A limited study and accompanying discussion were presented to motivate high-level strategies for setting parameters for both the above-mentioned index structures and search strategies.

An experimental evaluation of 15 algorithms, corresponding to all feasible combinations of index structures and search strategies, was conducted across three datasets of varying difficulty. Considering for all experiments the Lookup Search as a baseline search strategy, on the easiest of the three datasets, it was demonstrated that the LSH index structure benefits considerably from all three novel search strategies; Voting Search, Natural Classifier Search, and Quick-select Natural Classifier Search. The best-performing search strategy for the index structure was the Natural Classifier Search, which presented a performance increase of 145.66% over the Lookup Search. For the medium difficulty dataset, the LSH index structure saw a considerable benefit from the Voting Search, which demonstrated a performance increase of 65.06% over the Lookup Search and a lesser performance increase of 16.27% for the Natural Classifier Search strategy. Thus, it is concluded that under the experimental setting of this thesis, the performance of the locality-sensitive hashing-based index structure LSH benefits from novel search strategies.

The experimental evaluation also demonstrated that the performance of the C2LSH index structure improved under the Natural Classifier Search strategy for the easy and medium datasets, where it demonstrated performance increases of 60.79% and 22.22% respectively over the Lookup Search strategy. Thus, it can be concluded that the performance of both index structures based on locality-sensitive hashing, benefits from novel search strategies. Compared to other index structures, the C2LSH index structure in general performed considerably worse, with the performance gap widening for more difficult datasets.

The experimental evaluation of each algorithm demonstrated that on the easiest of the three datasets, for each index structure, the Natural Classifier Search strategy performed the best among all search strategies. On the other hand, for the datasets of medium and hard difficulty, the Voting Search strategy is demonstrated to show markedly better performance than the Natural Classifier Search strategy for the two index structures LSH and RP-Forest. Furthermore, for the most difficult of the three datasets, the two best-performing algorithms utilize the Voting Search strategy. These algorithms greatly outperform all three algorithms utilizing the Natural Classifier Search strategy. Thus, it is theorized that the Voting Search strategy demonstrates a greater degree of robustness against increasingly difficult datasets. This is a theory that nuances the conclusion of [HJR22b], in which the Natural Classifier Search strategy is initially proposed.

The Quick-select Natural Classifier Search strategy, which is a novel contribution of this thesis, is experimentally shown to perform well under similar

conditions as the Natural Classifier Search. The search strategy consistently performs somewhat worse but demonstrates a qualitative improvement in the ease of algorithm parameter configuration over the Natural Classifier Search.

Finally, on the basis of the experimental evaluation, the thesis establishes recommendations for algorithms for k-Approximate Nearest Neighbor search in the high-recall range for datasets of various difficulties. The RKD-Forest coupled with Natural Classifier Search is recommended for datasets of easy to medium difficulty, while the RP-Forest coupled with Voting Search is recommended for datasets of medium to hard difficulty, as well as for a general-purpose algorithm for datasets of unknown difficulty. For datasets with a difficulty beyond the GloVe-100 dataset, both the RP-Forest and LSH with Voting Search are hypothesized to be good candidates.

8.1 Future Work

This thesis has shown that the performance of locality-sensitive hashing-based index structures for k-ANNS can be improved by novel search strategies, such as Natural Classifier Search and Voting Search, for the evaluated recall range. The LSH index structure with Voting Search performed best of any evaluated algorithms on the most difficult of the three datasets - therefore this thesis recommends further experimental evaluation of the algorithm for datasets of higher difficulty than GloVe-100 to investigate its performance in this setting.

Given the exploration of the high-level strategies for setting algorithm parameters for each algorithm, this thesis can also be used as a starting point for research on automatic algorithm configuration of the evaluated algorithms.

Finally, it is opportune to implement performance-optimized versions of the algorithms recommended by this thesis, including the LSH with Voting Search algorithm, in a more performant language than Java, to evaluate against current state of the art algorithms for a wider recall range.

References

[ABF] Martin Aumüller, Erik Bernhardsson, and Alex Faithfull. Benchmarking results. https://ann-benchmarks.com/index.html#datasets. [Accessed 20-05-2024].

[ABF18] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, 2018.

[AC21] Martin Aumüller and Matteo Ceccarello. The role of local dimensionality measures in benchmarking nearest neighbor search. *Information Systems*, 101:101807, 2021.

[AC23] Martin Aumüller and Matteo Ceccarello. Recent approaches and trends in approximate nearest neighbor search. *IEEE Data Engineering Bulletin*, 2023.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975.

[CG] Matteo Ceccarello and Johann Gamper. https://cecca.github.io/attimo/VLDB-supplemental/. [Accessed 05-05-2024].

[Cha02] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[Dev01] Luc Devroye. On the probablistic worst-case time of"find". *Algorithmica*, 31:291–303, 2001.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, page 253–262, New York, NY, USA, 2004. Association for Computing Machinery.

[DS15] Sanjoy Dasgupta and Kaushik Sinha. Randomized partition trees for nearest neighbor search. *Algorithmica*, 72:237–263, 5 2015.

[GFFN12] Junhao Gan, Jianlin Feng, Qiong Fang, and Wilfred Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 541–552, New York, NY, USA, 2012. Association for Computing Machinery.

[HJR22a] Ville Hyvönen, Elias Jääsaari, and Teemu Roos. A multilabel classification framework for approximate nearest neighbor search. volume 35, pages 35741–35754. Curran Associates, Inc., 2022.

[HJR22b] Ville Hyvönen, Elias Jääsaari, and Teemu Roos. A multilabel classification framework for approximate nearest neighbor search, 2022.

[HPT+16] Ville Hyvonen, Teemu Pitkanen, Sotiris Tasoulis, Elias Jaasaari, Risto Tuomainen, Liang Wang, Jukka Corander, and Teemu Roos. Fast nearest neighbor search through sparse random projections and voting. pages 881–888. Institute of Electrical and Electronics Engineers Inc., 2016.

[IM98] Piotr Indyk and Rajeev 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, 1998.

[JAL23] Malte Johnsen, Andreas Plenge Andreasen, and Guan-Ting Lin. Research project. https://github.itu.dk/apla/CuteVectorNextDoor, 2023.

[KE11] Nikolaos Kouiroukidis and Georgios Evangelidis. The effects of dimensionality curse in high dimensional knn search. In *2011 15th Panhellenic Conference on Informatics*, pages 41–45. IEEE, 2011.

[LHC06] Ping Li, Trevor J Hastie, and Kenneth W Church. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 287–296, 2006.

[LZS+16] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement (v1.0), 2016.

[McG12] Catherine C. McGeoch. *A Plan of Attack*, page 17–49. Cambridge University Press, 2012.

[ML14] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[Ora] Oracle. HashMap (Java SE 11 & JDK 11) — docs.oracle.com. https://docs.oracle.com/en%2Fjava%2Fjavase%2F11%2Fdocs%2Fapi%2F%2F/java.base/java/util/HashMap.html. [Accessed 07-05-2024].

[PP16] Ninh Pham and Rasmus Pagh. Scalability and total recall with fast coveringlsh, 2016.

[SAH08] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[Ses13] Peter Sestoft. Microbenchmarks in java and c#. *Lecture Notes, September*, 2013.

[Sim] Harsha Simhadri. Practical vector search: Neurips 2023 competition.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

A Implementation Appendix

```
import java.util.HashMap;

public class MyClass {
    public static void main(String[] args) {

        int[] a = {1, 2, 3, 4, 5};
        int[] b = {1, 2, 3, 4, 5};

        HashMap<int[], String> myMap = new HashMap<>();
        myMap.put(a, "foo");
        myMap.put(b, "bar");
        // Expected size (number of entries) is 1
        System.out.println(myMap.size());
        // Actual is 2

        HashMap<Integer, String> myOtherMap = new HashMap<>();
        myOtherMap.put(1, "foo");
        myOtherMap.put(1, "bar");
        // Expected size (number of entries) is 1
        System.out.println(myOtherMap.size());
        // Actual is 1

    }
}
```

Listing 1: Code example of unexpected behavior of the Java `HashMap` class.

B Results Appendix

Best achieved SPS over 0.8 recall - Fashion-MNIST

	Lookup	Voting	Natural Classifier	Quick-select Natural Classifier
RP-Forest	901	2029	**2318**	2124
RKD-Forest	1725	1882	**3182**	2946
LSH	357	521	**877**	793
C2LSH	278	-	**447**	398

Table 9: Raw results of best achieved SPS for a recall of 0.8 or greater obtained from the Fashion-MNIST dataset. The best result achieved for each index structure is highlighted in bold. All values are rounded to the nearest integer.

Algorithm Parameters, Fashion-MNIST

RP-Forest		
Lookup Search	$leafSize = 16$, $\|L\| = 60$	-
Voting Search	$leafSize = 128$, $\|L\| = 65$	$\tau = 4$
Natural Classifier Search	$leafSize = 16$, $\|L\| = 30$	$\tau = 0.008$
Quick-select Natural Classifier Search	$leafSize = 8$, $\|L\| = 35$	$\nu = 200$
RKD-Forest		
Lookup Search	$leafSize = 8$, $\|L\| = 125$	-
Voting Search	$leafSize = 16$, $\|L\| = 145$	$\tau = 2$
Natural Classifier Search	$leafSize = 4$, $\|L\| = 55$	$\tau = 0.02$
Quick-select Natural Classifier Search	$leafSize = 4$, $\|L\| = 70$	$\nu = 100$
LSH		
Lookup Search	K = 15, r = 4000.0, $\|L\| = 75$	-
Voting Search	K = 15, r = 6000.0, $\|L\| = 50$	$\tau = 3$
Natural Classifier Search	K = 13, r = 4000.0, $\|L\| = 25$	$\tau = 0.004$
Quick-select Natural Classifier Search	K = 15, r = 4000.0, $\|L\| = 25$	$\nu = 600$
C2LSH		
Lookup Search	K = 16 , $partitionSize = 800$, t = 12	-
Voting Search	-	-
Natural Classifier Search	K = 14, $partitionSize = 100$, t = 10	$\tau = 0.001$
Quick-select Natural Classifier Search	K = 14, $partitionSize = 100$, t = 10	$\nu = 2000$

Table 10: Overview of the single best-performing set of algorithm parameters for each index structure and search strategy combination. Note that in many cases, other algorithm configurations of similar characteristics performed almost identically.

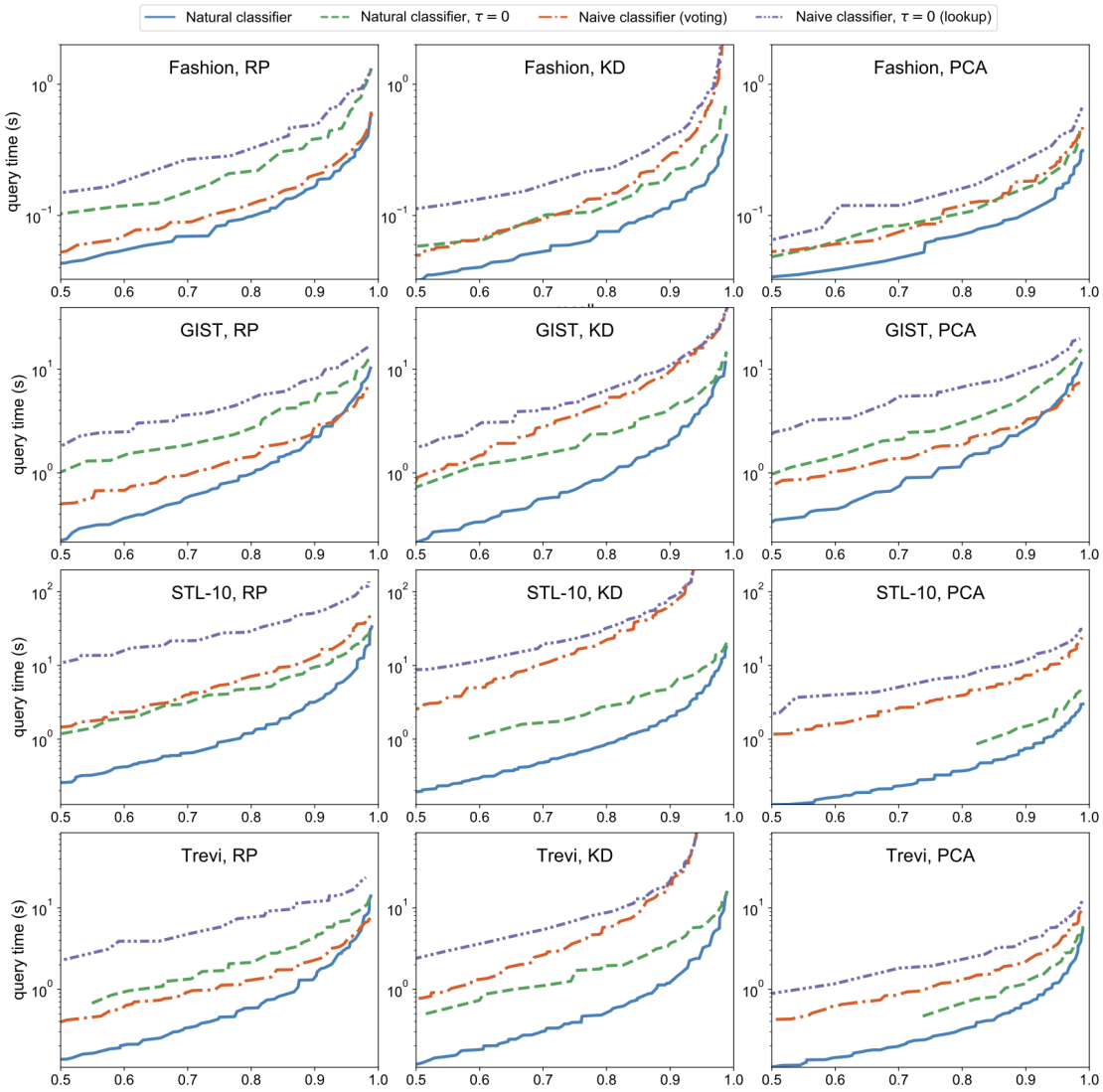

Figure 21: Results from the experiments in [HJR22b]. Original caption: "Recall vs. query time (log scale) of ensembles of, RP, k-d, and PCA trees. The solid blue line is the natural classifier proposed in this paper; the dash-dotted red line is the natural classifier with $\tau = 0$ that is included for completeness; the dashed green line is voting; and the double-dash-dotted violet line is lookup search. The natural classifier is the fastest and the lookup search is the slowest of the methods for each tree type".

Best achieved SPS over 0.8 recall - SIFT

	Lookup	Voting	Natural Classifier	Quick-select Natural Classifier
RP-Forest	391	**749**	444	292
RKD-Forest	579	630	**808**	724
LSH	166	**274**	193	156
C2LSH	9	-	11	8

Table 11: Overview best achieved SPS for a recall of 0.8 or greater obtained from the SIFT dataset. The best result achieved for each index structure is highlighted in bold. All values are rounded to the nearest integer.

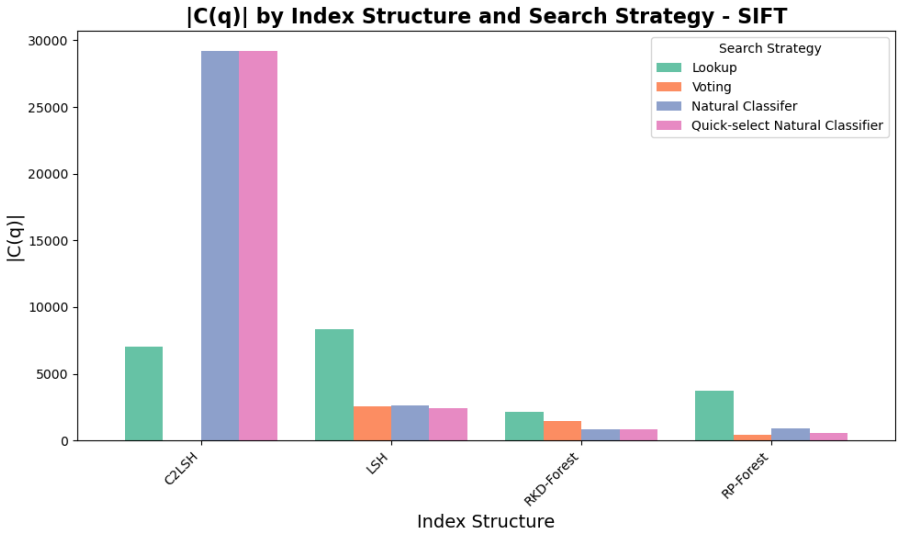

Figure 22: Overview best achieved $|\mathbf{C}(q)|$ by index structure and search strategy for a recall of 0.8 or greater obtained from the SIFT dataset.

Algorithm Parameters, SIFT

RP-Forest		
Lookup Search	$leafSize = 16$, $\|L\| = 310$	-
Voting Search	$leafSize = 128$, $\|L\| = 236$	$\tau = 3$
Natural Classifier Search	$leafSize = 16$, $\|L\| = 200$	$\tau = 0.0009$
Quick-select Natural Classifier Search	$leafSize = 16$, $\|L\| = 200$	$\nu = 1300$
RKD-Forest		
Lookup Search	$leafSize = 16$, $\|L\| = 266$	-
Voting Search	$leafSize = 32$, $\|L\| = 344$	$\tau = 2$
Natural Classifier Search	$leafSize = 4$, $\|L\| = 274$	$\tau = 0.0027$
Quick-select Natural Classifier Search	$leafSize = 4$, $\|L\| = 300$	$\nu = 850$
LSH		
Lookup Search	K = 20, r = 755.0, $\|L\| = 400$	-
Voting Search	K = 22, r = 1000.0, $\|L\| = 400$	$\tau = 3$
Natural Classifier Search	K = 20, r = 750.0, $\|L\| = 225$	$\tau = 0.0003$
Quick-select Natural Classifier Search	K = 20, r = 755.0, $\|L\| = 225$	$\nu = 3750$
C2LSH		
Lookup Search	K = 14, $partitionSize = 26500$, t = 10	-
Voting Search	-	-
Natural Classifier Search	K = 14, $partitionSize = 4300$, t = 10	$\tau = 0.0$
Quick-select Natural Classifier Search	K = 14, $partitionSize = 4300$, t = 10	$\nu = 40000$

Table 12: Overview of the single best-performing algorithm set of parameters for each index structure and search strategy combination. Note that in many cases, other algorithm configurations of similar characteristics performed almost identically.

LSH with Natural Classifier Search Configurations - SIFT

Build Parameters			Search Parameters	Performance	
K	r	$\|L\|$	τ	**Recall**	**SPS**
16	600	400	0.00015	0.8	185
18	750	150	0.00033	0.8023	174
18	750	225	0.0005	0.803	160
20	750	200	0.00025	0.802	169
20	**750**	**225**	**0.0003**	**0.804**	**193**
20	750	350	0.00053	0.802	182
20	800	200	0.0004	0.802	185
20	900	125	0.0004	0.799	154
20	900	200	0.00075	0.75	134
22	750	325	0.0002	0.804	158
24	750	400	0.00022	0.776	169

Table 13: Overview of select alternative algorithm configurations for the LSH with Natural Classifier Search algorithm for the SIFT dataset. The best-achieved performance is highlighted in bold.

Algorithm Parameters, GloVe

RP-Forest		
Voting Search	$leafSize = 1024$, $\|L\| = 420$	$\tau = 3$
Natural Classifier Search	$leafSize = 64$, $\|L\| = 500$	$\tau = 0.000062$
RKD-Forest		
Natural Classifier Search	$leafSize = 128$, $\|L\| = 320$	$\tau = 0.0$
LSH		
Voting Search	K = 14, $\|L\| = 650$	$\tau = 3$
Natural Classifier Search	K = 17, $\|L\| = 600$	$\tau = 0.000065$

Table 14: Overview of the single best-performing set of algorithm parameters for each index structure and search strategy combination.