



THE UNIVERSITY OF
MELBOURNE

An Evaluation of Prioritized Dynamic Continuous Indexing

Shuai Wang

Student Number: 830166

Subject Code: COMP90055

Credit Points: 25 pt

Project Type: Conventional Research Project

Supervised by

Trevor Cohn & Matthias Petri

June 3, 2018

Abstract

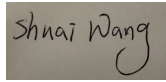
k -nearest neighbors searching (k -NN) is a fundamental problem in pattern recognition, statistical classification, computer vision, databases, recommendation systems, DNA sequencing, plagiarism detection, chemical similarity and many other fields. There are two types of k -NN algorithms: exact and approximate algorithms. Most exact algorithms have encountered the curse of dimensionality, which describes the problem of query time complexity depending exponentially on dimensionality. In this paper, I will introduce two recently proposed approximate algorithms: dynamic continuous indexing (DCI) (Li and Malik (2016)), and its improved version namely prioritized DCI (Li and Malik (2017)), which can avoid the curse of dimensionality. I will show the evaluation of their performance compared to exhaustive linear search, K-D tree (Bentley (1975)) and Ball tree (Omohundro (1989)). The two algorithms achieve low accuracy and slow runtime performance. Some possible reasons will be explained.

Keywords. k -nearest neighbors search; Exact and approximate algorithms; Dynamic Continuous Indexing (DCI); Prioritized DCI.

Declaration

I certify that

- this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- where necessary I have received clearance for this research from the University's Ethics and have submitted all required data to the Department.
- the thesis is 6600 words in length (excluding text in images, table, bibliographies and appendices).

A rectangular box containing a handwritten signature in black ink. The signature appears to read "Shuai Wang" in a cursive, flowing script.

Acknowledgements

Shuai Wang thanks Trevor Cohn and Matthias Petri for supervising the project.

Contents

1	Introduction	7
2	Related Work	10
3	Dynamic Continuous Indexing (DCI)	12
4	Prioritized DCI	18
5	Experiences	22
6	Conclusions	26
7	Future Work	27

List of Tables

1	The query time complexities of algorithms	12
2	Explanations of notations	15
3	Time and space complexities of DCI	17
4	Time and space complexities of Prioritized DCI	23
5	Server machine configurations	23
6	Datasets	24
7	Evaluation results	25

List of Figures

1	Example of k -nearest neighbors search	7
2	The curse of dimensionality	9
3	Example of projection	13
4	The structure of dynamic continuous index	16
5	Projection along east direction	18
6	Projection along north direction	19
7	Example of priority	20

1 Introduction

K -Nearst Neighbors Search Minsky and Papert first raised the problem of k -nearest neighbors (k -NN) (Minsky and Papert (2017)), which was defined as given a collection of n points and a query in d -dimensional space, find the k points that are nearest to the query. For example, we have a number of points in Figure 1.

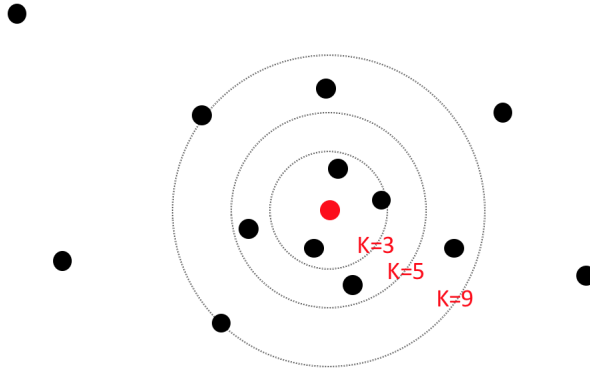


Figure 1: Example of searching k -nearest points to the red query point. The distance of the three points lying in the smallest circle are the top 3 shortest over all points. Thus, the three points are the 3-nearest neighbors to the query point. Similarly, the five points in the middle circle are the 5-nearest neighbors while the nine points in the biggest circle are the 9-nearest neighbors.

Which points are the top 3 nearest neighbors of the red point? A simple way to do that is to compute the euclidean distance¹ from the red point to every other point. Those three points with shortest distances are returned as the 3-nearest neighbors of the red point. Here, the three points lying in the smallest circle are the top 3 nearest points of the red point. In the same way, we can find the 5-nearest and 9-nearest points. This is the basic idea of k -nearest neighbors.

k -NN subsequently became a fundamental problem in pattern recognition, statistical classification, computer vision, databases, recommendation systems, DNA sequencing, plagiarism detection, chemical similarity and many other fields

¹The euclidean distance between points x and y is the distance from x to y or y to x in Euclidean space. Let $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$ be two points, the distance between x and y can be computed by:

$$dist(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

(Guru, Sharath, and Manjunath (2010)).

Exact & Approximate K -NN Two types of algorithms have been proposed to tackle the problem of k -NN: exact and approximate algorithms. Exact algorithms return the k true nearest points to a given query point. For example, RP tree (Dasgupta and Freund (2008)), Spill tree (Liu, Moore, Yang, and Gray (2005)) and naive exhaustive linear search are exact algorithms. On the other hand, the $(1+\epsilon)$ approximate algorithms return points whose distance from the query is no more than $(1+\epsilon)$ times the distance of the true k th nearest neighbor (Liu et al. (2005)). The focus of approximate algorithms is on the trade-off between query time and accuracy. Several algorithms, such as ANNOY (Bernhardsson (2016)), FLANN (Muja and Lowe (2009)), DOLPHINN (Avarikioti, Emiris, Psarros, and Samaras (2016)), Locality Sensitive Hashing (LSH) (Indyk and Motwani (1998)), K-D tree (Bentley (1975)) and Metric tree (Uhlmann (1991)), have been posed and their implementations are widely used in a variety of real world applications.

Curse of Dimensionality Most exact algorithms have encountered the curse of dimensionality posed by Bellman in 1961, which describes the problem of query time complexity depending exponentially on dimensionality. In other words, a linear increase in dimensionality can result in exponential increase in query time and memory space as showed in Figure 2.

The main reason is that a linear increase in dimensionality leads to exponential increase in the number of points near to a query. Consequently, in high dimensional space, the performance of exhaustive search can be much better than some exact algorithms.

There are two types of dimensionalities: the ambient dimensionality d and the intrinsic dimensionality d' where often $d' < d$.²

Ambient dimensionality represents the number of features or dimensions of data points in the space. Intrinsic dimensionality refers to the number of those fea-

²An simple example to explain ambient and intrinsic dimensionalities: given $f(x, y)$ is a two-variable function while $g(x)$ is one-variable function, and $f(x, y) = g(x)$. Both the values of f and g vary with the variable x . But g is constant in terms of the variable y . At the same time, f is also constant with variable y . As a result, the values of f and g only depend on the variable x . In this case, the ambient dimensionality of (x, y) is 2 while the intrinsic dimensionality is 1.

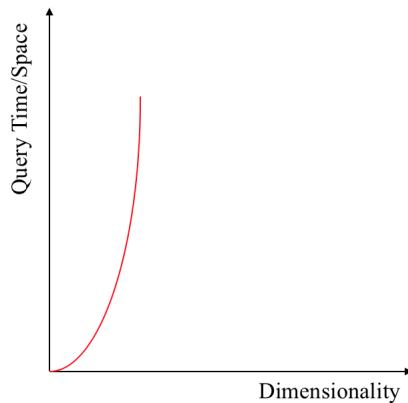


Figure 2: A linear increase in the number of dimensions of data in space results in exponential increase in the query time when algorithms take to find k -nearest neighbors and the space required by the algorithms.

tures that contain intrinsic properties of the data. The query time complexity of different exact algorithms might vary from ambient dimensionality to intrinsic dimensionality. For instance, the query time of algorithms like K-D tree and R tree (Guttman (1984)) have an exponential increase in ambient dimensionality. Algorithms like Spill tree, RP tree (Dasgupta and Freund (2008)) and Cover tree (Beygelzimer, Kakade, and Langford (2006)) can avoid exponential increase in ambient dimensionality but still have exponential increase on query time in intrinsic dimensionality.

These algorithms like K-D tree and LSH suffer from the curse of dimensionality due to their strategy, which is space partitioning. They partition the vector space into several discrete cells and construct a data structure to organize the points in each cell. When a query arrives, find all cells that contain the query point and perform brute force to identify k -nearest points. When the number of dimensionality linearly increases, the number of cells and the volume of cell exponentially increase due to exponential increase of the volume of the space (Li and Malik (2016)). This results in exponential increase in query time complexity and space complexity.

DCI & Prioritized DCI The main focus of this paper are two algorithms proposed by Ke Li and Jitendra Malik at ICML 2016 and 2017 called dynamic continuous indexing (DCI) (Li and Malik (2017)) and prioritized DCI (Li and

Malik (2017)). Instead of partitioning the vector space into a couple of cells, DCI constructs continuous indices, each of which has a relative order of all points. At query time, retrieve data points that are nearest to query point over all indices. Prioritized DCI can speed up the points retrieval by assigning a priority to each index. The two algorithms can escape from the curse of dimensionality. We will compare their performance with exhaustive linear search, K-D tree and Ball tree (Omohundro (1989)). Specifically the contributions of this thesis are:

1. First public implementation and evaluation of the proposed methods.
2. Careful evaluation of accuracy and runtime performance of both DCI and Prioritized DCI.
3. Discussion of the intuition behind both algorithms and their actual performance.

Let’s briefly review the development of a variety of k -nearest neighbors search algorithms.

2 Related Work

Over past several decades, a rich collection of algorithms for k -nearest neighbors search have been proposed. There are two types: exact algorithms and approximate algorithms.

Exact Algorithms The majority of exact algorithms are based on space partitioning and points are stored in tree-based data structures. For example, K-D tree, R tree and X tree (Berchtold, Keim, and Kriegel (2001)) partition the vector space into discrete cells and maintain a data structure to keep track of the points in each cell. At query time, they identify cells that include the query or near to the query, and then exhaustive search is performed over points in those cells to find k nearest points. Even if they perform effectively for low-dimensional data, their query time still depends exponentially on ambient dimensionality. This is because there is an exponential increase in the number of leaves in trees that need to be visited as the number of dimensionality linearly increases.

An improved approach posed by Meister (Meiser (1993)) utilizes overlapped cells, which can reduce query time complexity from exponential to polynomial in

ambient dimensionality while it still has exponential space complexity. Sequentially, Spill tree, RP tree and Virtual Spill tree are improved by randomizing the direction of hyperplane at each node of the tree. While the idea of randomization allows them to escape from exponential dependence on ambient dimensionality, they cannot avoid exponential dependence on intrinsic dimensionality.

Instead of partitioning space, some algorithms (Orchard (1991); Clarkson (1999); Karger and Ruhl (2002)) are based on local search strategies. At query time, they start with a random point and find a new point near to query from its neighbors in each iteration. The query time of the algorithm (Karger and Ruhl (2002)) still exponentially depends on intrinsic dimensionality. The space complexity of the algorithm (Orchard (1991)) and the algorithm (Clarkson (1999)) are quadratic in the size of the dataset, which cannot be used in large datasets. Some other algorithms, such as Navigating Nets (Krauthgamer and Lee (2004)), Cover tree and Rank Cover tree (Houle and Nett (2015)) utilize a coarse-to-fine granularity. Coarse subsets of points at different scales are maintained and points are visited with decreasing radii at increasingly finer scales. Unfortunately, they still run exponential time in intrinsic dimensionality.

Approximate Algorithms Generally, exact algorithms are not able to completely remove the curse of dimensionality. As a result, approximate algorithms were proposed. An optimal algorithm (Arya and Mount (1993)) is one of them. But the query time is still exponential. Locality Sensitive hashing (LSH) (Indyk and Motwani (1998)) became popular, which groups points in vector space into buckets on some distance metric. The weakness of this algorithm is that it is hard to adapt datasets with varying data density. Sequentially, some algorithms based on data-dependent hashing schemes, such as k-means (Paulevé, Jégou, and Amsaleg (2010)) and spectral partitioning (Weiss, Torralba, and Fergus (2009)), were proposed. But they still have difficulty in adapting datasets with large variation in data density. The algorithm recently posed by Anagnostopoulos (Anagnostopoulos, Emiris, and Psarros (2015)) gives a different idea, which can obtain approximate relative order between points by projection. The query time is linear in ambient dimensionality and sublinear in the size of dataset while exponential in intrinsic dimensionality.

Table 1 shows the query time complexities of above algorithms where d' refer to intrinsic dimensionality, d to ambient dimensionality and n the number of

points in dataset.

Algorithm	Query Time Complexity
Exact algorithms:	
RP Tree	$O((d' \log d')^{d'} + \log n)$
Spill Tree	$O((d')^{d'} + \log n)$
Cover Tree	$O(2^{12d'} \log n)$
Rank Cover Tree	$O(2^{O(d' \log h)} n^{\frac{2}{h}})$ for $h \geq 3$
Navigating Net	$2^{O(d')} \log n$
Approximate Algorithms:	
K-D Tree	$O((\frac{1}{\epsilon})^d \log n)$
LSH	$O(dn^{\frac{1}{(1+\epsilon)^2}})$

Table 1: The query time complexities of some exact and approximate algorithms. d' , d and n refer to intrinsic dimensionality, ambient dimensionality and the size of dataset. Adapted from the paper (Li and Malik (2017)).

3 Dynamic Continuous Indexing (DCI)

Li and Malik (2016) proposed a new k -nearest neighbors search algorithm called Dynamic Continuous Indexing (DCI) based on the idea of projections. The main intuition of this algorithm is that a point can be viewed as a nearest neighbor if each of its projections along multiple random directions are near to the query projection. We will illustrate this with an example.

First we discuss the concept of projections. In Figure 3(a), the projection of the vector \vec{a} onto the vector \vec{b} is the segment OC . It can be computed by:

$$proj_{\vec{b}} \vec{a} = \langle \vec{a}, \vec{b} \rangle = |\vec{a}| \cdot \cos(\theta) = \frac{\vec{a} \cdot \vec{b}}{|\vec{b}|} \quad (1)$$

Here, $\vec{a} = (3, 4)$, $\vec{b} = (5, 1)$. Hence,

$$proj_{\vec{b}} \vec{a} = \frac{(3, 4) \cdot (5, 1)}{|(5, 1)|} = \frac{3 \times 5 + 4 \times 1}{\sqrt{5^2 + 1^2}} = 3.72$$

This value indicates how much of \vec{a} is pointing in the same direction as the vector \vec{b} . Next we discuss the concept of random projections onto unit vectors as utilised by DCI. It projects a vector onto a random unit vector (the direction

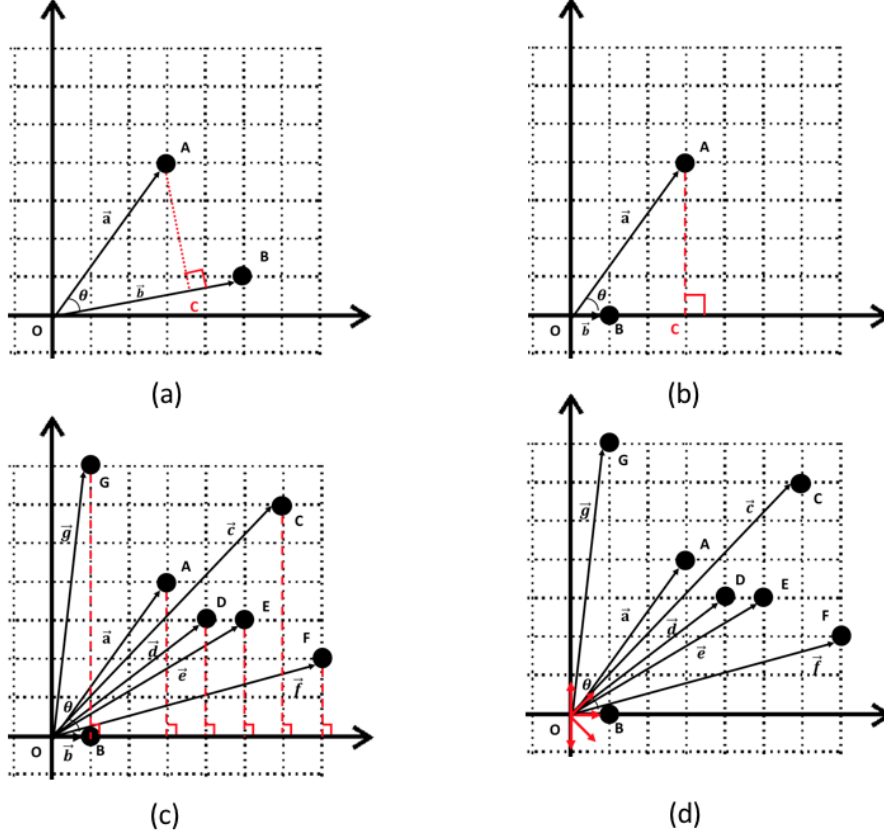


Figure 3: (a) An example of projection: \vec{a} onto \vec{b} . The projection of \vec{a} on \vec{b} is the length of segment OC . (b) An example of one-dimensional random projection. \vec{b} is a unit vector with the same direction as x -axis. Projecting \vec{a} on \vec{b} is a one-dimensional random projection. (c) Find the nearest point to point A along the direction. Obviously, the projection of point D is the closest to the projection of A . Thus, point D is the nearest point to A . (d) Find nearest points to point A along multiple random directions. Project all points onto each random unit vector. The projection of point D will be the closest one to the projection of point A along each direction. As a result, Point D can be viewed as the true nearest neighbor of point A .

is random and the length is one). In Figure 3(b), vector \vec{b} becomes a unit vector in the same direction as x -axis. So $|\vec{b}| = 1$. We project \vec{a} onto \vec{b} ,

$$proj_{\vec{b}} \vec{a} = \frac{\vec{a} \cdot \vec{b}}{|\vec{b}|} = \vec{a} \cdot \vec{b} \quad (2)$$

Thus, the dot product of \vec{a} and \vec{b} represents the one-dimensional random projection of \vec{a} onto \vec{b} . Here, $\vec{a} = (3, 4)$ and $\vec{b} = (1, 0)$. Hence,

$$proj_{\vec{b}}\vec{a} = (3, 4) \cdot (1, 0) = 3 \times 1 + 4 \times 0 = 3$$

We have 6 points in Figure 3(c), which one is the nearest point to A? We first project the 6 points onto the unit vector. Given $\vec{a} = (3, 4)$, $\vec{c} = (6, 6)$, $\vec{d} = (4, 3)$, $\vec{e} = (5, 3)$, $\vec{f} = (7, 2)$, $\vec{g} = (1, 7)$ and $\vec{v} = (1, 0)$ for point A, C, D, E, F, G and unit vector respectively. We can obtain their projections by just computing dot production between unit vector and their vectors. So,

$$proj_{\vec{b}}\vec{a} = 3, \quad proj_{\vec{b}}\vec{c} = 6, \quad proj_{\vec{b}}\vec{d} = 4, \quad proj_{\vec{b}}\vec{e} = 5, \quad proj_{\vec{b}}\vec{f} = 7 \text{ and } proj_{\vec{b}}\vec{g} = 1$$

We found the projection of D is closer to the projection of A than other points. Thus, we approximately think D is the nearest point to A along this random direction.

What if we have many points in the space? Which points are the k -nearest neighbors of a query point? In the same way above, we project all points onto all the random unit vectors. Those points whose projections are near to the query projection along all directions are viewed as the k -nearest neighbors. For example, in Figure 3(d), points D and E are the 2-nearest neighbors of point A.

Based on the intuition above, Li and Malik (2016) designed the DCI algorithm. It consists of the index construction algorithm outlined in Algorithm 1 and the k -NN retrieval algorithm shown in Algorithm 2. Some notations used in the algorithms are explained in Table 2.

Let's first go through the index construction algorithm. What is a continuous index? It is a data structure used to efficiently search for a specific point. More concretely, DCI uses two types of continuous indices: simple index and composite index. A simple index consists of a random unit vector and a self-balancing binary search tree like AVL tree. A composite index contains m simple indices. L composite indices are used in DCI. Thus DCI needs to construct mL simple indices in total. The data structures of simple and composite indices are showed in Figure 4.

In detail, Algorithm 1³ constructs the DCI index as follows:

³The pseudocode of Algorithm 1 adapted from the paper by Li and Malik (Li and Malik

Notation	Explanation
C_l	Array of size n , used to find candidate points, $\forall l \in [L]$
d	The ambient dimensionality
D	The dataset of n points p^1, p^2, \dots, p^n
ϵ	Maximum tolerable failure probability ϵ
h_{jl}^i	The data point that is the i th closest to query point in the tree T_{jl}
k_0	The number of points to retrieve in each composite index
k_1	The number of points to visit in each composite index
L	The number of composite indices
m	The number of simple indices that composite a composite index
n	The number of data points in the dataset
\bar{p}_{jl}^i	The projection of point p^i on the direction u_{jl}
P_l	Min heap of size m , $\forall l \in [L]$
q	Query point
\bar{q}_{jl}	The projection of query point q on the direction u_{jl}
S_l	A set of candidate points for each composite index, $\forall l \in [L]$
T_{jl}	AVL tree, $j \in [m], l \in [L]$
u_{jl}	Random unit vector in \mathbb{R}^d , $j \in [m], l \in [L]$
(T_{jl}, u_{jl})	Simple index, $j \in [m], l \in [L]$

Table 2: Explanations of the notations used in the pseudocode of DCI and Prioritized DCI algorithms. Adapted from the paper (Li and Malik (2017)).

Algorithm 1 Data structure construction procedure.

```

1: function CONSTRUCT( $D, m, L$ ):
2:  $\{u_{jl}\}_{j \in [m], l \in [L]} \leftarrow mL$  random unit vectors in  $\mathbb{R}^d$ 
3:  $\{T_{jl}\}_{j \in [m], l \in [L]} \leftarrow mL$  empty AVL trees
4: for  $j = 1$  to  $m$  do
5:   for  $l = 1$  to  $L$  do
6:     for  $i = 1$  to  $n$  do
7:        $\bar{p}_{jl}^i \leftarrow \langle p^i, u_{jl} \rangle$ 
8:       Insert  $(\bar{p}_{jl}^i, i)$  into  $T_{jl}$  with  $\bar{p}_{jl}^i$  being the key and  $i$  being the value
9: return  $\{(T_{jl}, u_{jl})\}_{j \in [m], l \in [L]}$ 

```

1. generate mL random unit vectors for each simple index (Line 2).
2. project every point onto each unit vector (Line 7).
3. insert into AVL tree (Line 8).

Data points in dataset D are organized into AVL tree in terms of their projection values along a random direction. Searching a specific point in a simple index is (2016)).

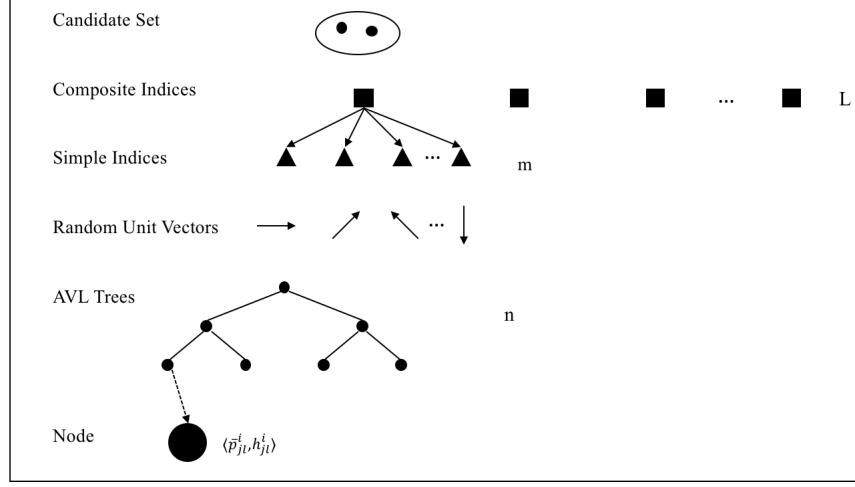


Figure 4: DCI constructs L composite indices, each of which consists of m simple indices. Each simple index contains a random unit vector and an AVL tree that contains n nodes. Each node is denoted as a key-value pair: $\langle projection, pointid \rangle$. After DCI finds a candidate point, puts it into a candidate set.

efficient due to the logarithmic properties of the AVL tree. On the other hand, if we have more data points going into the dataset, we just insert them into the AVL tree (Adelson-Velsky, 1962). The inserting time is also logarithmic. Dynamically updating dataset is efficient. Moreover, the AVL tree approximately preserves the relative order between points. Points with similar projection values tend to be closer to each other in the AVL tree while near points in space tend to have similar projection values. Equivalently, the nearest points of a query in space are probable close to the query in the AVL tree. Let's see how DCI can find k -nearest neighbors of a query using the AVL tree in Algorithm 2.

In detail, Algorithm 2⁴ retrieves k -nearest neighbors as follows:

1. project query onto each unit vector (Line 3).
2. return a point as a candidate if its projection is nearest to query projection in all indices (Line 8-12).
3. compute Euclidean distance from query to each candidate (Line 14).

⁴The pseudocode of Algorithm 2 adapted from the paper (Li and Malik (2016)).

Algorithm 2 Algorithm for k-nearest neighbour retrieval

```

1: function QUERY ( $q, \{(T_{jl}, u_{jl})\}_{j,l}, \epsilon$ )
2:  $C_l \leftarrow$  array of size  $n$  with entries initialized to 0  $\forall l \in [L]$ 
3:  $\bar{q}_{jl} \leftarrow \langle q, u_{jl} \rangle \forall l \in [m], l \in [L]$ 
4:  $S_l \leftarrow \emptyset \forall l \in [L]$ 
5: for  $i = 1$  to  $n$  do
6:   for  $l = 1$  to  $L$  do
7:     for  $j = 1$  to  $m$  do
8:        $(\bar{p}_{jl}^i, h_{jl}^i) \leftarrow$  the node in  $T_{jl}$  whose key is the  $i^{th}$  closest to  $\bar{q}_{jl}$ 
9:        $C_l[h_{jl}^i] \leftarrow C_l[h_{jl}^i] + 1$ 
10:      for  $j = 1$  to  $m$  do
11:        if  $C_l[h_{jl}^i] = m$  then
12:           $S_l \leftarrow S_l \cup \{h_{jl}^i\}$ 
13:    if Stopping Condition Satisfied( $i, S_l, \epsilon$ ) then break
14: return  $k$  points in  $\bigcup_{l \in [L]} S_l$  that are the closest in Euclidean distance in  $\mathbb{R}^d$  to  $q$ 

```

4. retrieve k candidates with shortest distance as k -nearest neighbors to the query (Line 14).

Points are returned in terms of their positions in AVL tree instead of the locations in space. Therefore, DCI can adapt to large variation of data density in dataset. The locations of points in space are actually ignored because one-dimensional random projection reduces the number of dimensions of points to one dimension. In other words, a scalar represents a vector. Moreover, DCI allows user to trade-off accuracy against speed by adjusting the number of candidate points retrieved from a composite index.

The construction and query time complexity and space complexity are showed in Table 3.

Property	Complexity
Construction	$O(mLn(d + \log n))$
Query	$O(d \max(\log n, n^{1-\frac{1}{d'}}))$
Space	$O(mLn)$

Table 3: The construction time complexity if for constructing mL simple indices. The query time complexity is for retrieving k -nearest neighbors of a single query. The space complexity shows how much memory space required by DCI. Adapted from the paper (Li and Malik (2016)).

Projecting a point onto a random unit vector takes $O(d)$ time due to dot production. As a result, projecting n points onto mL random unit vectors takes $O(mLnd)$ time. Construct an AVL tree in a simple index by inserting n points. It takes $O(n \log n)$ time. Constructing mL AVL tree takes $O(mLn \log n)$ time. Overall, index constructing algorithm takes $O(mLn(d + \log n))$ time.

The query time complexity of DCI is linear in ambient dimensionality d , sub-linear in intrinsic dimensionality d' , and linear in the size of dataset n .

DCI needs to store mL simple indices, each of which contains n points. Therefore, the space complexity of DCI is $O(mLn)$.

As a result, DCI can avoid the curse of dimensionality.

4 Prioritized DCI

At query time, DCI looks up constituent simple indices in a composite index one by one to find candidates. DCI assumes each simple index has the same probability to retrieve a nearest point. However, this is not true in practice.

For example, we have two simple indices with different projections along two random directions: the first simple index has a direction to the east as shown in Figure 5 while the second simple index has a direction to the north as shown in Figure 6.

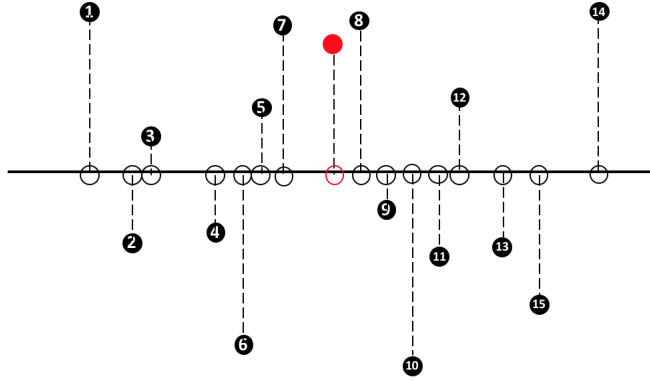


Figure 5: Project all data points along east direction. The projections are uniformly distributed on the left and right sides along the direction. In addition, those projections are close to the red query projection.

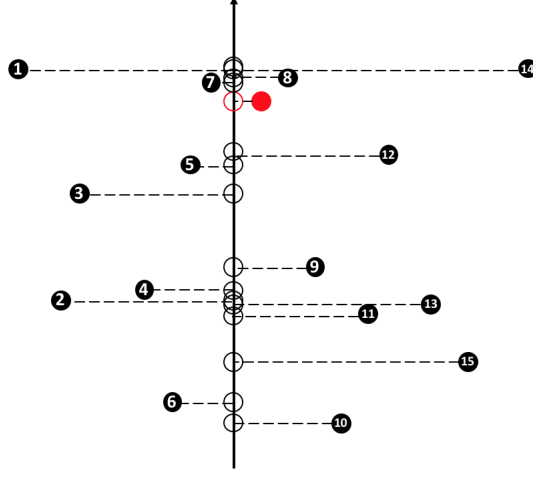


Figure 6: Projects all data points along north direction. Most of projection are located in the bottom far away from the red query projection.

We can observe that the projections of points are approximately evenly distributed on the left and right of the red point (query) projection along the east direction as shown in Figure 5, while majority of points distribute on the bottom along the north direction as shown in Figure 6. In fact, it's more likely to find true nearest neighbors as shown in Figure 5 because more points are near to query. So we want to first look up points in the first simple index and then visit the second index.

How to set the visiting order of simple indices? Li and Malik (Li and Malik (2017)) proposed a variant of DCI which assigns a priority to each simple index that is used to indicate which index to visit in the next iteration. They refer to this variant as Prioritized DCI. The priority is set to the negative absolute difference between the query projection and the data point next to be visited in the index:

$$priority_{jl} = -|\bar{p}_{jl}^{i+1} - \bar{q}_{jl}| \quad (3)$$

Prioritized DCI first finds the point whose projection is the nearest to query projection from a simple index. Compute the priority with Equation (3), and then assign it to the simple index. Calculate the priority for each simple index. In the first iteration, the simple index with highest priority will be visited first. Retrieve the nearest point in this index, and then compute the new priority

For example, in the first iteration, the priority of the simple index in Figure 7 is -0.5 . In the second iteration, the priority is updated to -1 .

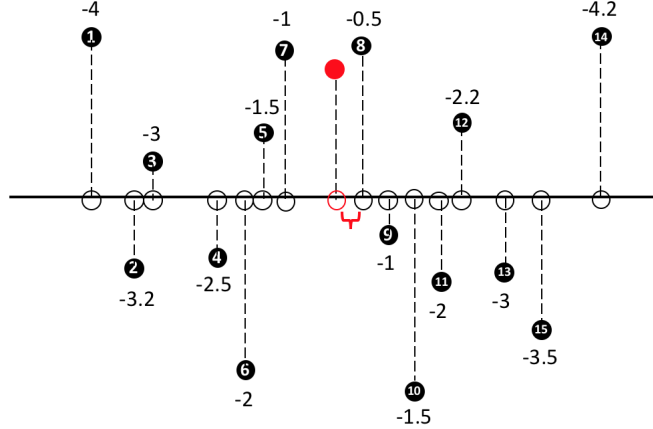


Figure 7: Example of prioritizing points in one simple index projection by their distance to the red query point. Point 8 is assigned highest priority as it has the closest absolute distance (0.5) to the query point, whereas point 7 has the second highest priority with absolute distance (1).

with the second nearest point and update the current priority. In the second iteration, compare the updated priority to other priorities, find the new index with highest priority and visit it. Retrieve the nearest point in the index except those points retrieved before, and update priority with the next nearest point. Iteratively visit simple index by priority until points retrieved are enough to get true k -nearest neighbors.

Let's go through the query Algorithm 3⁵ of Prioritized DCI:

1. project query onto each unit vector (Line 3).
2. find the point in each simple index whose projection is the nearest to query projection (Line 8).
3. compute the priority of the simple index (Line 9).
4. insert it into a heap (Line 9).
5. pop the nearest point from the heap which is from the simple index with highest priority (Line 14).

⁵The pseudocode of Algorithm 3 adapted from the paper by Li and Malik (Li and Malik (2017)).

Algorithm 3 k-nearest neighbour querying procedure

```

1: function QUERY( $q, \{(T_{jl}, u_{jl})\}_{j,l}, k_0, k_1$ )
2:  $C_l \leftarrow$  array of size  $n$  with entries initialized to 0  $\forall l \in [L]$ 
3:  $\bar{q}_{jl} \leftarrow \langle q, u_{jl} \rangle \forall j \in [m], l \in [L]$ 
4:  $S_l \leftarrow \emptyset \forall l \in [L]$ 
5:  $P_l \leftarrow$  empty heap  $\forall l \in [L]$ 
6: for  $l = 1$  to  $L$  do
7:   for  $j = 1$  to  $m$  do
8:      $(\bar{p}_{jl}^1, h_{jl}^1) \leftarrow$  the node in  $T_{jl}$  whose key is the closest to  $\bar{q}_{jl}$ 
9:     Insert  $(\bar{p}_{jl}^1, h_{jl}^1)$  with priority  $-|\bar{p}_{jl}^1 - \bar{q}_{jl}|$  into  $P_l$ 
10: for  $i' = 1$  to  $k_1 - 1$  do
11:   for  $l = 1$  to  $L$  do
12:     if  $|S_l| < k_0$  then
13:        $(\bar{p}_{jl}^i, h_{jl}^i) \leftarrow$  the node with the highest priority in  $P_l$ 
14:       Remove  $(\bar{p}_{jl}^1, h_{jl}^1)$  from  $P_l$  and insert the node in  $T_{jl}$  whose key
       is the next closest to  $\bar{q}_{jl}$ , which is denoted as  $(\bar{p}_{jl}^{i+1}, h_{jl}^{i+1})$ , with
       priority  $-|\bar{p}_{jl}^{i+1} - \bar{q}_{jl}|$  into  $P_l$ 
15:        $C_l[h_{jl}^i] \leftarrow C_l[h_{jl}^i] + 1$ 
16:       if  $C_l[h_{jl}^i] = m$  then
17:          $S_l \leftarrow S_l \cup \{h_{jl}^i\}$ 
18: return  $k$  points in  $\bigcup_{l \in [L]} S_l$  that are the closest in Euclidean distance in
     $\mathbb{R}^d$  to  $q$ 

```

6. update the priority of the simple index (Line 14).
7. push the next nearest point into the heap from the simple index (Line 14).
8. check whether the popped point is a candidate (Line 16).
9. perform brute force on candidate set to find k -nearest points (Line 18).

Prioritized DCI visits points in simple indices by priority. As a result, it will find nearest points quicker than DCI. To ensure it returns exact k -nearest points with highest probability, the number of points to retrieve k_0 in each composite index and the number of points to visit k_1 are set to K_0 and K_1 ⁶:

$$K_0 = k \max(\log(\frac{n}{k}), (\frac{n}{k})^{1-\frac{m}{d'}}) \quad (4)$$

$$K_1 = mk \max(\log(\frac{n}{k}), (\frac{n}{k})^{1-\frac{1}{d'}}) \quad (5)$$

For example, we want to find 10-nearest neighbors from a dataset of 1,000 points. Each point has a intrinsic dimensionality d' of 20. A composite index consists of 25 simple indices. In this case, we need to visit $K_1 = 19,858$ points to retrieve $K_0 = 46$ candidates to make sure we have highest probability to get true 10-nearest points from candidates.

However, the settings for k_0 and k_1 are impractical. Generally, the size of dataset is huge like 100,000. If other parameters are the same with previous example, then we need to visit 1,577,393 points. It will take infeasible time to execute. In fact, the settings only exist in theory. In practice, users will trade-off accuracy against time by using smaller k_0 and k_1 . For example, $k_0 = 0.5K_0$ and $k_1 = 0.5K_1$, the accuracy is lower while runtime is shorter.

The time and space complexities of Prioritized DCI are showed in Table 4. The query time complexity of Prioritized DCI is linear in ambient dimensionality d , sublinear in intrinsic dimensionality d' , and linear in the size of dataset n . The space complexity is linear in the size of dataset n .

5 Experiences

We will compare the performance of DCI and Prioritized DCI to Brute Force, K-D tree and Ball tree. Our interest is in the accuracy and query time. More

⁶Check the derivation of K_0 and K_1 in the paper by Li and Malik (Li and Malik (2017)).

Property	Complexity
Construction	$O(m(dn + n \log n))$
Query	$O(dk \max(\log(\frac{n}{k}), (\frac{n}{k})^{1-\frac{m}{d'}}) + mk \log m(\max(\log(\frac{n}{k}), (\frac{n}{k})^{1-\frac{1}{d'}})))$
Insertion	$O(m(d + \log n))$
Deletion	$O(m \log n)$
Space	$O(mn)$

Table 4: The construction time complexity is for constructing m simple indices of a composite index. The query time complexity is for retrieving k -nearest neighbors. The insertion and deletion time complexities are for inserting and deleting a point in an AVL tree. The space complexity shows how much memory space required by Prioritized DCI. Adapted from the paper (Li and Malik (2017)).

concretely, the accuracy is denoted as how many true k -nearest neighbors can be identified.

$$Accuracy = \frac{\text{The number of returned points that are true nearest neighbors}}{\text{The number of true nearest neighbors}} \quad (6)$$

In addition, the actual query time of each algorithm is measured on the same server machine, which exclude construction time. The server machine configurations outlined in Table 5:

Item	Configuration
CPU	Intel(R) Xeon(R) Gold 6144 CPU @ 3.50GHz
RAM	503 GB

Table 5: Server machine configurations.

The higher accuracy and shorter query time, the better the performance.

Datasets The performance of those algorithms will be evaluated on two random datasets D1 and D2, and MNIST dataset as showed in Table 6. The random datasets are generated by scikit-learn. Each data point has 200 ambient dimensionalities and 20 intrinsic dimensionalities. Such data points uniformly distribute in the vector space. D1 contains 1,000 data points while D2 contains 10,000 data points. MNIST dataset consists of 28*28 grayscale images of handwritten digits. Normalized vector of 784 dimensionalities represents an

Dataset	Points	dimensionality
D1	1,000	200
D2	10,000	200
MNIST	70,000	784

Table 6: The performance of proposed methods, Brute Force, K-D Tree and Ball Tree are evaluated on three datasets: D1 dataset contains 1,000 points, each of which has 200 dimensions; D2 dataset contains 10,000 points, each of which has 200 dimensions; and MNIST dataset contains 70,000 points, each of which has 784 dimensions.

image. The training set of 60,000 points and the testing set of 10,000 points are combined to form a new dataset of 70,000 points. All the three datasets contain high dimensional data. We want to check whether those algorithms can perform well in high dimensional space.

Implementation DCI and Prioritized DCI are implemented in Python. The implementations of K-D tree and Ball tree are from scikit-learn. Brute Force computes distance between points under Euclidean distance. All the algorithms run on the same server machine.

Hyper-parameters To get best performance on each dataset, we need tune the hyper-parameters of DCI and prioritized DCI. As a result, we use $m=25$ and $L=2$ for both algorithms on all the three datasets. Prioritized DCI has three different group parameters settings: lower bound: $k_0 = 0.15K_0$ and $k_1 = 0.55K_1$; middle: $k_0 = 0.55K_0$ and $k_1 = 0.8K_1$; upper bound: $k_0 = K_0$ and $k_1 = K_1$. The number of nearest neighbors for each query $k = 10$.

Results The results of accuracy and query time of those algorithms are showed in Table 7. We can observe that the performance of Brute Force, K-D tree and Ball tree are much better than DCI and Prioritized DCI, each of which got accuracy of 1 and miniseconds. They all found whole 10 true nearest neighbors of the query, and finished searching very fast. Unfortunately, our proposed algorithms DCI and Prioritized DCI performed poorly. The accuracy is quite low. Both algorithms found only 2 or 3 true nearest neighbors out of 10.

Dataset	DCI	PDCI(L)	PDCI(M)	PDCI(U)	BF	KD	Ball
1,000	0.1 7 m	0.2 16 s	0.3 22 s	0.4 26 s	1 3 ms	1 0.9 ms	1 0.5 ms
10,000	0 4.4 h	0.1 19 m	0.1 32 m	0.1 32 m	1 17 ms	1 7 ms	1 2 ms
MNIST 70,000	0 1 d	0.2 14.8 h	0.1 22.3 h	0.1 27.5 h	1 321 ms	1 99 ms	1 97 ms

Table 7: Accuracy and query time of DCI, Prioritized DCI, Brute Force, K-D Tree and Ball Tree. In each cell, the first value is accuracy and the second value is query time. Time units used: s: second; ms: millisecond; m: minute; h:hour; d:day. Generally, Brute Force, K-D Tree and Ball Tree performs much better than DCI and Prioritized DCI. They got higher accuracy and shorter time.

Analysis In our experiments, DCI and Prioritized DCI didn’t perform well as Li and Malik stated in their papers. Why our implementations are so slow and inaccurate? Here are some possible reasons.

- **Python.** One possible reason might be Python. As we known, Python is a higher level language than C language, which is much slower than C because code is interpreted at runtime instead of being compiled. Python might have slight influence on our implementation.
- **For Loops.** The construction algorithm 1, DCI query algorithm 2 and Prioritized DCI query algorithm 3 use many ‘for’ loops. In addition, most of them are double or triple nested ‘for’ loops. In our experiments, the number of points n in datasets is large: $n = 1,000$ in $D1$ dataset; $n = 10,000$ in $D2$ dataset; $n = 70,000$ dataset. The number of simple indices $m = 25$. The number of composite indices $L = 2$. For example, in DCI query algorithm 2 on MNIST dataset, there is a triple nested ‘for’ loop, which will execute $n \cdot L \cdot m = 70000 \times 25 \times 2 = 350000$ times. This will take long time to execute. Therefore, ‘for’ loops highly affect the query performance.
- **AVL Tree.** Both DCI and Prioritized DCI implementations use AVL tree to store the projections of n points. Even through searching a point just takes $\log(n)$ time. But finding the i th nearest point takes at least $k \cdot \log(n)$ time for each query. If we use a list to replace AVL tree, finding the position

of a query in list will take $\log(n)$ time. But then finding the i th nearest point will take constant time. Therefore, using list to replace AVL tree is able to speed up the query performance.

- m & L . The parameter m represents the number of simple indices. In other words, it represents the number of random directions. To trade-off the accuracy and speed, we set $m = 25$, which means we have 25 random directions used to projection. Even though a point is closest to a query along all the 25 directions, it's not 100% that the point is truly closest to the query in space. Only when the point is closest to the query along infinite number of random directions, the point can be a true nearest point to the query in space. Therefore, m is totally not enough to find accurate nearest points. This results in very low accuracy. To improve the accuracy, we can use large m but it will take very long time to execute.

6 Conclusions

In this report, we pointed out the curse of dimensionality which many exact traditional k -nearest neighbours searching methods had encountered. We then introduced a new approximate approach which can avoid the curse of dimensionality called dynamic continuous indexing (DCI) and illustrated that it runs in linear time in ambient dimensionality and sublinear time in intrinsic dimensionality and sublinear in the size of the dataset, and takes constant space in dimensionality and linear in the size of dataset. Later, we introduced an improved version of this algorithm called prioritized DCI, which visits simple index by priority. It showed a remarkable improvement on standard DCI. We compared DCI and prioritized DCI to classical algorithms like Brute Force, K-D tree and Ball tree. We found our implementations got bad performance: low accuracy and long runtime. The reasons are: Python executes very slowly; many 'for' loops in algorithms; AVL tree to store projections of points rather than list; the hyper-parameter m is small.

7 Future Work

Many optimizations of the implementations of DCI and Prioritized DCI have been left for the future due to lack of time. Future work concerns the optimizations and deeper analysis of performance. There are some ideas that I would have liked to try:

1. Implement DCI and Prioritized DCI in C language. As we known, C program executes much faster than equivalent Python program. We'll see how well the two algorithms performs in C.
2. Remove 'for' loops when coding. Use matrix to store data points. Dot product of matrix can be used to conduct projection. This is a way to emit 'for' loop. In addition, we'll continue to refine our code.
3. Replace AVL tree with list or array. As we discuss in experiments section, this is able to speed up the query algorithms.
4. Use large m to improve the accuracy. We'll try to run our implementations on powerful server machine within practical time.
5. Use Cross-validation technique to get average accuracy and query time. We can randomly choose 100 points in each dataset as queries. More concretely, 100 points are selected as queries and the rest points as a dataset in which we need to retrieve the k -nearest neighbors for each query. We record the accuracy and query time of each query. The average accuracy and query time over the 100 queries are stored. We repeat the procedure for 10 folds, each with a different random selection of queries. In the end, the average accuracy and query time over the 10 folds are used to evaluate the performance of those algorithms.

References

- Anagnostopoulos, E., Emiris, I. Z., & Psarros, I. (2015). Low-quality dimension reduction and high-dimensional approximate nearest neighbor. In *Lipics-leibniz international proceedings in informatics* (Vol. 34).
- Arya, S., & Mount, D. M. (1993). Approximate nearest neighbor queries in fixed dimensions. In *Annual acm-siam symposium on discrete algorithms* (Vol. 93, pp. 271–280).

- Avarikioti, G., Emiris, I. Z., Psarros, I., & Samaras, G. (2016). Practical linear-space approximate near neighbors in high dimension. *arXiv preprint arXiv:1612.07405*.
- Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 509–517.
- Berchtold, S., Keim, D. A., & Kriegel, H.-P. (2001). *The x-tree: an index structure for high-dimensional data, readings in multimedia computing and networking*. Morgan Kaufmann Publishers Inc., San Francisco, CA.
- Bernhardsson, E. (2016). Annoy: Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk, 2013. URL <https://github.com/spotify/annoy>, 2.
- Beygelzimer, A., Kakade, S., & Langford, J. (2006). Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on machine learning* (pp. 97–104).
- Clarkson, K. L. (1999). Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1), 63–93.
- Dasgupta, S., & Freund, Y. (2008). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual acm symposium on theory of computing* (pp. 537–546).
- Guru, D., Sharath, Y., & Manjunath, S. (2010). Texture features and knn in classification of flower images. *IJCA, Special Issue on RTIPPR (1)*, 21–29.
- Guttman, A. (1984). *R-trees: A dynamic index structure for spatial searching* (Vol. 14) (No. 2). ACM.
- Houle, M. E., & Nett, M. (2015). Rank-based similarity search: Reducing the dimensional dependence. *IEEE transactions on pattern analysis and machine intelligence*, 37(1), 136–150.
- Indyk, P., & Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual acm symposium on theory of computing* (pp. 604–613).
- Karger, D. R., & Ruhl, M. (2002). Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the thirty-fourth annual acm symposium on theory of computing* (pp. 741–750).
- Krauthgamer, R., & Lee, J. R. (2004). Navigating nets: simple algorithms for proximity search. In *Proceedings of the fifteenth annual acm-siam symposium on discrete algorithms* (pp. 798–807).
- Li, K., & Malik, J. (2016). Fast k-nearest neighbour search via dynamic con-

- tinuous indexing. In *International conference on machine learning* (pp. 671–679).
- Li, K., & Malik, J. (2017). Fast k-nearest neighbour search via prioritized dc. *arXiv preprint arXiv:1703.00440*.
- Liu, T., Moore, A. W., Yang, K., & Gray, A. G. (2005). An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems* (pp. 825–832).
- Meiser, S. (1993). Point location in arrangements of hyperplanes. *Information and Computation*, 106(2), 286–303.
- Minsky, M., & Papert, S. A. (2017). *Perceptrons: an introduction to computational geometry*. MIT press.
- Muja, M., & Lowe, D. (2009). Flann-fast library for approximate nearest neighbors user manual. *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*.
- Omohundro, S. M. (1989). *Five balltree construction algorithms*. International Computer Science Institute Berkeley.
- Orchard, M. T. (1991). A fast nearest-neighbor search algorithm. In *Acoustics, speech, and signal processing, 1991. icassp-91., 1991 international conference on* (pp. 2297–2300).
- Paulevé, L., Jégou, H., & Amsaleg, L. (2010). Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11), 1348–1358.
- Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information processing letters*, 40(4), 175–179.
- Weiss, Y., Torralba, A., & Fergus, R. (2009). Spectral hashing. In *Advances in neural information processing systems* (pp. 1753–1760).