IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 4 July 2024, accepted 22 July 2024, date of publication 25 July 2024, date of current version 5 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3433524

RESEARCH ARTICLE

LK-Index: A Learned Index for KNN Queries

YONGXIN PENG
School of Mathematics and Computer Application, Shangluo University, Shangluo 726000, China
Engineering Research Center of Qinling Health Welfare Big Data, Universities of Shaanxi Province, Shangluo 726000, China
e-mail: pengyx185@163.com

This work was supported by the Science and Technology Research Project of Shangluo University of China under Grant 21SKY004.

ABSTRACT The k-Nearest Neighbor (kNN) search is a crucial problem in database and data mining, especially in high-dimensional space. However, traditional kNN algorithms based on distance metrics and brute-force search often have low search efficiency and accuracy, and high computational complexity when dealing with large-scale high-dimensional datasets. These limitations have made them a bottleneck in practical applications. Inspired by the recently learned index that replaces B-tree with machine learning models, I propose a method for kNN search based on a learned index, named LK-index. Initially, a conventional tree-based index is created to process queries. The tree index is then utilized to find the k-nearest neighbor points, and the neural network is trained to act as a learned index. Finally, the actual k-nearest neighbors are obtained by computing the potential k-nearest neighbor points. Experiments conducted on synthetic and real-world datasets demonstrate that the LK-index yields certain advantages regarding search accuracy and time consumption.

INDEX TERMS KNN, learned index, locality sensitive hashing (LSH), recursive model index (RMI), neural network.

I. INTRODUCTION

In the era of big data, efficient discovery of similarity and accurate or approximate pattern recognition and classification in massive high-dimensional data have become crucial research topics. One widely used machine learning method is the k-Nearest Neighbor (kNN) search algorithm, which is indispensable in supervised learning. In practical application scenarios such as image recognition, text categorization, recommender systems, and medical diagnosis, the kNN algorithm has achieved remarkable results. Therefore, the kNN algorithm is widely used in the field of supervised learning.

The kNN algorithm is a commonly used technique to identify the closest neighbor within a dataset. However, it becomes computationally expensive as the dataset grows. To improve its efficiency, data can be organized using tree data structures like KD-tree [2], R-tree [3], HD-index [4], and HCTree [5]. These structures reduce the number of comparisons needed by hierarchical division and partitioning. However, when dealing with high-dimensional data, the tree structure can lead to the "curse of dimensionality," which decreases the search efficiency and turns it into a linear scan.

Recent research on learned index [1], [6] has presented new avenues for index structures ((Fig. 1(a)). The paper investigates how a machine learning model can be utilized instead of a B-tree. It discusses the adoption of a regression approach to determine the point to be queried between a minimum error value and a maximum error value, which validates the feasibility of a learned index model on a B-tree.

However, most learned index focus on point and range queries, with few studies on specific tree structures. I start by asking myself the following question: *Can I design a new index structure (as shown in Fig. 1b) for kNN search that adopts a neural network to replace the traditional tree-based index structure to deliver high performance in time and efficiency?* My answer to that question is a new index structure called LK-index. I propose a learned index structure to improve traditional tree-based index structures for kNN search. The contributions of the proposed approach are summarized as follows:

To build a learned index to answer kNN queries, I formulate the kNN search as a multiclass classification problem.

The associate editor coordinating the review of this manuscript and approving it for publication was Domenico Rosaci.

The candidate k-nearest neighbor points are obtained by multiple indexing and the final result is computed.

I propose a learned index based on a neural network model to support kNN queries. The model is trained based on the traditional tree index.

I conducted comprehensive experiments on eight benchmark datasets, Empirically I found that my index structure leads most of the time in terms of search efficiency and search accuracy.

The following is an overview of the paper's structure: I will begin by giving an introduction and background in Section II. After that, I will explain how the kNN search task can be approached as a supervised multi-classification problem and provide a summary of my proposed method in Section III. In Section IV, I will present the results of my evaluation. Lastly, I will discuss future work and conclude my findings in Section V.

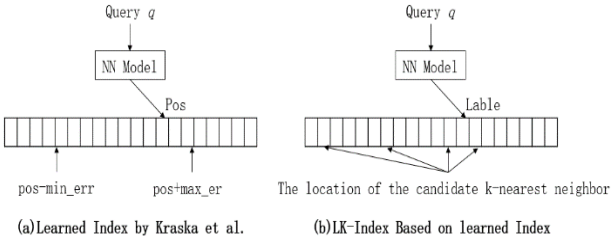

FIGURE 1. Tree-based index versus learned index.

II. BACKGROUND

A. RELATED WORK

This section provides an overview of the previous work in the kNN field. Traditional nearest neighbor search algorithms are broadly categorized into two types: tree-based [2], [3], [7], [8], [9], and locality-sensitive hash-based [10], [11], [12], [13].

Tree structure is a popular method for the kNN search. This structure requires space subdivision and comparison between the query and candidate data points that are likely to be close to it. One example of such a structure is the Ball-tree, which efficiently retrieves nearest neighbors. Each internal node of the ball tree represents a hypersphere containing multiple data points, and the hyperspheres represented by its child nodes are completely enclosed by the hyperspheres of the parent nodes. Other structures like R-tree [3] and A-tree [9] have also been proposed. However, with increasing data dimensionality in practical applications, using tree structures for exact searches in the kNN process can lead to a "curse of dimensionality." This phenomenon degrades the search-to-line scanning and significantly reduces query efficiency.

Locality Sensitive Hashing (LSH) is a widely used technique for kNN search. The basic principle behind this technique is to use the same hash function to map two adjacent data points in the original data space to the same storage space. Based on this, points that are closer to each other are likely to be in the same set. While the approximate nearest proximity algorithm has been able to satisfy the nearest neighbor query problem for some high-dimensional big data, some methods have been proposed to reduce the size of the index [14]. However, the search time of the algorithm is longer and the search accuracy also needs improvement.

Recently, Kraska and others from Google proposed the structure of a learned index [1], called the Recursive Model Index (RMI), and various improvements on the RMI [15], [16]. Since then, there has been a corpus of work on extending the ideas of the learned index to spatial and multi-dimensional data (e.g., [6], [17], [18], [19], [20], [21]). It is used to replace or enhance traditional tree-based indexes, such as HKC+-index [22]. There is also a learned index method built on LSH that can efficiently and flexibly map high-dimensional data to low-dimensional spaces, such as LLSH [23]. In addition, there is research focusing on the design and testing of learned index and providing findings to select suitable learned index under various practical scenarios [24].

B. PRELIMINARIES

Consider a dataset D with n data and dimension d. Given a query point $q = (i_1, \ldots, i_d)$ and a Euclidean distance function dis. Suppose that for the query point q, the true set of k-nearest neighbors is $T_k = (p_1, \ldots, p_k)$, where p_1 is the closest to q, followed by p_2, and so on. Assume that the set of k-nearest neighbors of the sequential candidates obtained by the final calculation is $C_k = (p'_1, \ldots, p'_k)$.

Definition 1: Denote by c the ratio of the distance between the output k-nearest neighbor and the true k-nearest neighbor for a query point q:

$$c = \frac{1}{k} \times \left(\sum_{i=1}^{k} \frac{dis\left(q, p'_i\right)}{dis\left(q, p_i\right)} \right) \tag{1}$$

If $c = 1$, it means that the predicted point is the true k-nearest neighbor; otherwise, the predicted result does not contain all the true k-nearest neighbors.

Definition 2: Use *acc* to denote the accuracy of the prediction, i.e., the proportion of data that correctly finds the true k-nearest neighbor points, out of all the data to be queried. Assuming that there are n' items of data such that $c = 1$ in (1), then:

$$acc = \frac{n'}{n} \tag{2}$$

Example 1: Suppose there are five two-dimensional data points, named q_1, q_2, q_3, q_4, and q_5, which need to be queried to find their nearest neighbor. Given that the actual nearest neighbors are (0.1, 0.2), (0.2, 0.3), (0.3, 0.4), (0.4, 0.5), and (0.5, 0.6), I need to determine the nearest neighbor for each point. The nearest neighbors found for the points are (0.1, 0.2), (0.2,0.3), (0.3,0.4), (0.4,0.6), and (0.7, 0.8). The *acc* is calculated as 3/5, which is equal to 60%, as (2).

Definition 3: Use *precision* to represent the proportion of correctly identified positive samples as a percentage of all samples predicted as positive by the classifier. Assuming that

the nearest neighbor to be found is k, and the number of candidate nearest neighbors is k', then:

$$precision = \frac{k}{k'} \tag{3}$$

Example 2: Find the three nearest neighbors of the query point q. The true labels are (1,1,1,0,0,0,0,0,0,0), and if the output is labeled (1,1,1,1,1,0,0,0,0,0), then the *precision* is 3/5=60%.

III. LK-INDEX

The LK-index is a technique that replaces the search and computation part of the tree index structure with a neural network model. For instance, when using a KD tree for the k-nearest neighbor search, fetching the k-nearest neighbor points requires numerous backtracking operations and distance calculations. However, with the LK-index approach, neural network operations can replace this process. Additionally, a small amount of computation and sorting are used to enhance the accuracy of the neural network, achieving similar results as the traditional approach.

Different index structures can be defined as regression or classification tasks as required. For example, the k-nearest neighbor search using KD-tree can be viewed as a multi-categorization problem. In other words, the point to be queried and the output result have similar attributes represented in the form of distances. The closer the distances, the more similar the classifications can be seen. Specifically, for a node, if this node may be one of the k neighbors of the point to be searched, i.e., candidate k nearest neighbors, it is considered as a positive class denoted by 1. On the contrary, it is a negative class indicated by 0.

Based on the above discussion, the process of constructing the LK-index is divided into two parts.

A. NEURAL NETWORK TRAINING AND PREDICTION

The neural network (Fig. 2) is trained using supervised learning, with a sigmoid activation function. To begin with, dataset D is transformed into a KD-tree, and each of the n data points in the tree is randomly assigned an index value from 0 to n, resulting in n categories. In this study, I use the Kohonen self-organizing mapping network (SOM) to identify k-nearest neighbor points in the training and test datasets. First, the entire dataset D is input into the SOM, and through unsupervised learning, the network automatically maps the data points to the corresponding nodes in the output layer. Each node represents a class of data points in the spatial location of the output layer and maintains the topological relationship of the input data. Then, for each query point q, I find its corresponding winner node through SOM. Based on the location of the winner node, combined with the neighborhood property of the SOM, I am able to identify the k nearest-neighbor nodes of q. The index values of these nodes are subsequently converted into corresponding category labels, which are used to construct the label sets for the training and testing sets. For the labels of the training and test sets, the categorical values

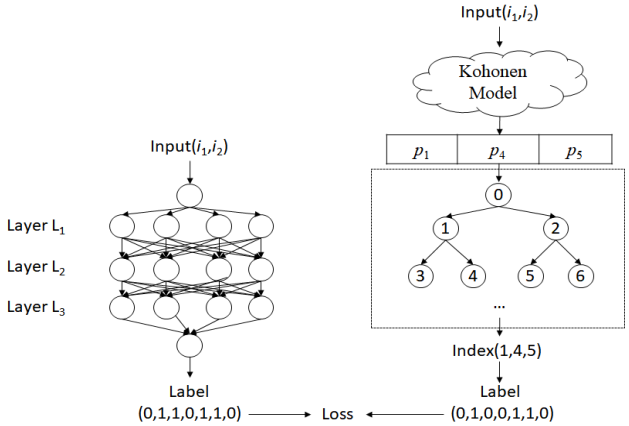

FIGURE 2. The supervised strategy in the neural network stage.

corresponding to the index values are set to 1 and the rest to 0. When the model is trained, for a given query point q, the candidate k nearest neighbor points are outputted. To find the corresponding nodes in the KD-tree, indexing is used to calculate their distances from the point q being searched, and then they are sorted. Finally, the first k nodes are taken as the true k-nearest neighbors.

In a multiclassification model, the search for each class can be seen as a binary classification problem. This means that a point can either be a k-nearest-neighbor point (represented by 1) or not (represented by 0). The classifier predicts the probability distribution as $p=Pr(y=1)$. As a result, the loss function is defined accordingly:

$$loss(y, p) = -logPr(y|p) = -(ylog(p) + (1 - y)log(1 - p)) \tag{4}$$

When the actual label of a piece of data is $y = 1$, the loss is calculated as $-log(p)$. This means that the lower the probability of the classifier's predicted probability $p=Pr(y=1)$, the higher the loss of classification. Conversely, the higher the predicted probability $p=Pr(y=1)$, the lower the loss. On the other hand, when the label $y = 0$, the loss is calculated as $-log(1-p)$. In this case, the higher the predicted probability $p=Pr(y=1)$, the higher the loss of classification.

B. LK-INDEX FOR kNN SEARCH

In this section, I will provide a technical explanation of my proposed solution. My approach consists of five stages (see Fig. 3). The entire training and prediction process is done using a feedforward neural network.

1) INPUT STAGE

The input data used for a search should have the same dimension. An input data point is represented as $q = (i_1, \ldots, i_d)$, where $i_1, \ldots, i_d$ denote the values in each dimension. If there are x data points to be searched, then the whole input data is represented as $D = q_1, \ldots, q_x$, where $q_1, \ldots, q_x$ have the same d values.

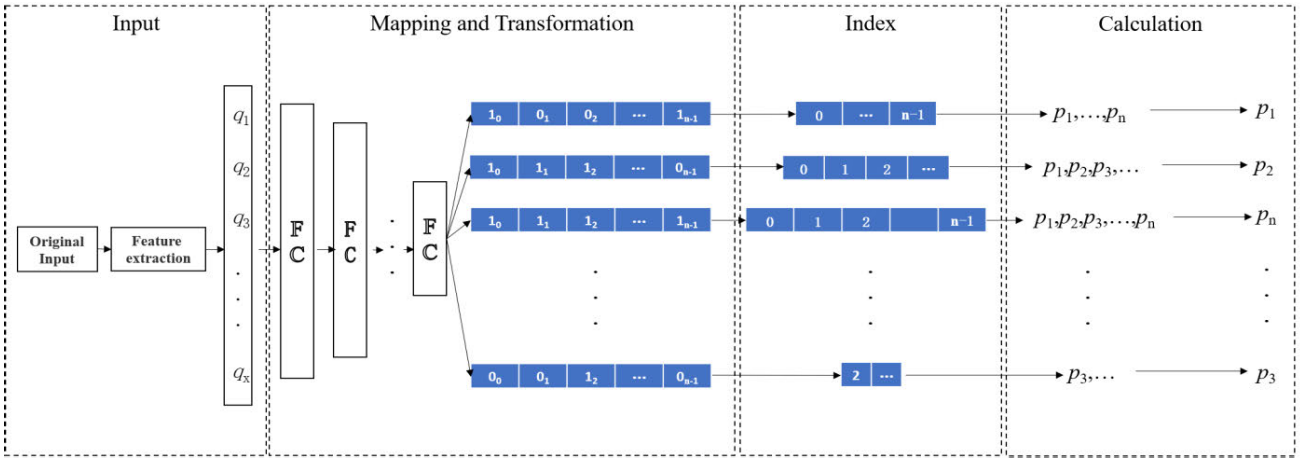

FIGURE 3. Overview of my approach.

TABLE 1. A two-dimensional KD-tree.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
i_1	2	1	8	6	7	2	6	9	1	8
i_2	6	5	9	3	1	9	5	7	8	1
Index	0	1	2	3	4	5	6	7	8	9

2) MAPPING STAGE

The neural network is composed of multiple fully connected layers that produce an x^*n matrix, with each value being a number between 0 and 1.

3) CONVERSION STAGE

To convert numbers distributed between 0 and 1 to 0 or 1, a threshold α is required. The output of this stage is an x^*c matrix where each value in the matrix is either 0 or 1. If a position in the matrix has a value of 1, it means the node corresponding to that position in the KD-tree is a potential k-nearest neighbor. On the other hand, if the position has a value of 0, it can be assumed that it is not a k-nearest neighbor, and there is no need to process this position afterward. Therefore, positions with a 0 value are not considered k-nearest neighbors and can be skipped during processing.

4) INDEXING STAGE

This stage converts the 1-labeled positions into index values to obtain the candidate k' nearest neighbors.

5) CALCULATION STAGE

To find the k nearest neighbors of a query point q in a KD-tree, the index value of the output results is compared to the KD-tree to identify the node it represents. Then, the distance between the query point and the k candidate's nearest neighbor points is calculated, and the nodes are ranked based on this distance. The first k nodes in the ranking are the true k nearest neighbors. The whole process is described in an algorithm as follows:

Algorithm 1 LK-Index Search (q, D, α, *KD-Tree*, k)

Input: query point q, dataset D, threshold α, KD-tree structure *KD-Tree*, number of nearest neighbors k
Output: list of indices of the k nearest neighbors of q
BEGIN
  M_q = Neural_Network_Map(q)
  M_D = [Neural_Network_Map(q_i) for each q_i in D]
  B_D = [[(m_ij >= α)? 1: 0 for j in range(columns)] for i in range(rows)]
  candidate_indices = [(i, j) for i in range(len(D)) for j in range(len($D[i]$)) if $B\_D[i][j]$ == 1]
  nearest_neighbors_indices=*KD_Tree*.Find_Neighbors(q, *candidate_indices*, k)
  RETURN *nearest_neighbors_indices*
END

Consider a two-dimensional KD tree with 10 data entries, as shown in Table 1, where "i_1" and "i_2" refer to the values on the first and second dimensions, respectively. The 10 classifications are represented by index values 0-9. Table 2 provides an example of training and testing two specific input data, q_1 and q_2, in a neural network. Assuming $k = 3$, the three nearest neighbors of q_1 can be found through the KD tree, which are p_1, p_2, and p_4. This translates into index values of 0,1, and 3, respectively, so the actual labels are 1, 1, 0, 1, 0, 0, 0, 0, 0, and 0. The prediction of this piece of data is considered correct if the index value corresponding to the position where 1 is located in the output of the neural network contains the index value corresponding to the true k-nearest

TABLE 2. Training and testing of neural networks.

	i_1	i_2	k=3	*Index*	*The actual label*	*Output labels*	*predicted outcomes*
q_1	1	2	p_1,p_2,p_4	0,1,3	1,1,0,1,0,0,0,0,0,0	1,1,0,1,0,0,0,0,0,1	*correct*
q_2	6	5	p_4,p_7,p_8	3,6,7	0,0,0,1,0,0,1,1,0,0	1,0,0,0,0,0,1,1,0,0	*wrong*

TABLE 3. The search process.

	i_1	i_2	*Label*	*Index*	*Candidate kNN(q)*	*True kNN(q)*
q_3	5	1	0,0,0,1,1,0,0,0,0,1	3,4,9	p_4, p_5, p_{10}	p_4, p_5, p_{10}
q_4	8	5	1,0,0,1,0,0,1,1,0,0	0,3,6,7	p_1, p_4, p_7, p_8	p_4, p_7, p_8

neighbors. This ensures that the true k-nearest neighbor is obtained by the calculation. The output of the neural network is allowed to exceed the number of k-values by 1, indicating the candidate's k-nearest neighbor. For example, for the query point q_4 (Table 3), the neural network outputs 1, 0, 0, 1, 0, 0, 1, 1, 0, and 0. There are 4 candidate k-nearest neighbors, and the first 3 nodes are the true k-nearest neighbors through calculation and sorting. The algorithm is described below:

Algorithm 2 K_Nearest_Neighbors_Search(q, *KD_Tree*, k, *neural_network*)

Input: an n-dimensional vector query point q, an n-dimensional KD tree *KD tree* containing a collection of data points, number of nearest neighbors to be found k, The trained neural network model *neural network*
Output: *nearest_neighbors_indices*
BEGIN
  neural_network_output = neural_network.predict(q)
  candidate_list = []
    for i = 1 to length(*neural_network_output*):
      if *neural_network_output*[i] == 1:
        candidate_list.append(i)
    if length(candidate_list) > k:
      candidate_list = candidate_list[: k + 1]
    for *index* in candidate_list:
      candidate_point= KD_Tree.find_point_by_index(*index*)
      for each point in KD_Tree:
        if *point.index* == *index*:
          distance=KD_Tree.calculate_distance(q, *point*)
      KD_Tree.update_nearest_neighbors(*nearest_neighbors_*
indices, *point.index*, *distance*, k)
      return *nearest_neighbors_indices*
END

Extending this to the more general case is shown in Fig. 4.

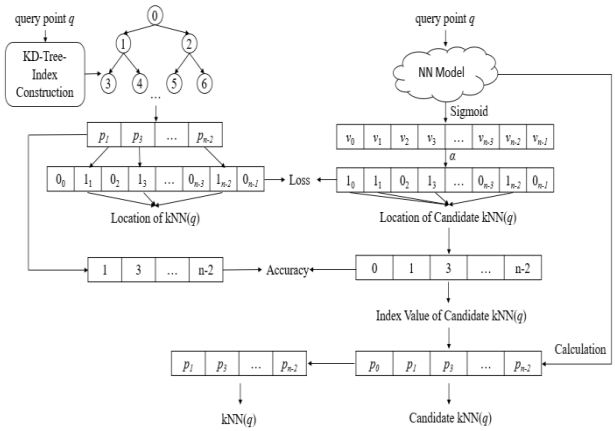

FIGURE 4. A more generalized situation.

IV. EXPERIMENTS

This section provides a detailed overview of the experiments conducted. All the experiments were performed on a GPU server with 128 GB of RAM, two 2.1 GHz Intel(R) E5 processors, and two GTX1080Ti GPU cards with 11 GB of dedicated RAM. The operating system used was Ubuntu 14.10, and the implementation was done in Python 3.6 and Keras 3.0. I ran each experiment ten times and considered the median or the average of the ten outcomes as the final performance.

In this study, a flexible neural network is constructed for a binary classification problem with arbitrary input dimension n. The following is a detailed parameter description of the neural network configuration:

Input layer: the input layer of the network receives a vector with n features, which allows the model to handle data instances of varying complexity and diversity.

Hidden Layer: the NN consists of multiple fully connected hidden layers, each using a Leaky ReLU activation function to introduce nonlinearities and enhance the expressive power of the model. The exact number of layers and the number of neurons per layer can be adjusted according to the complexity of the task.

Output layer: the last layer of the network is an output layer with a single neuron that is activated by a sigmoid activation function, which outputs a value between 0 and 1.

Loss function: to train the network, I used a binary cross-entropy loss function. This loss function is applied to the sigmoid output layer and ensures that the predicted probabilities learned by the model are consistent with the true labels.

TABLE 4. The datasets used in the experiments: four are randomly generated, and four are from the real world.

Dataset	Type	Dimension	Mean	Dataset	Type	Dimension	Mean
Uniform	Random	100	0.5	Tiny Images	GIST	384	0.11
Normal	Random	100	0	Ann SIFT	SIFT	128	27.05
Lognormal	Random	100	1.65	Nytimes	word2vec	250	0
Exponential	Random	100	1	Glove	word2vec	200	0

Optimization algorithm: I choose the Adam optimizer to minimize the loss function with a learning rate of 1e-4.

The weights of the network are initialized using the Xavier initializer, which helps to avoid the problem of vanishing or exploding gradients at the beginning of training. The bias term of the network is initialized to zero, which ensures that there is no preset bias at the beginning of training.

With this configuration, the neural network is able to adapt to different dimensions of input data and learn a complex mapping from input features to predicted probabilities, while the binary cross-entropy loss function ensures that the probabilistic interpretation of the network's outputs is consistent with the true labels.

A. DATASETS

In my experiments, I use two types of datasets: synthetic and real. The purpose of this is to simulate data with different distributions that are commonly found in real-world applications. To achieve this, I have created four synthetic datasets that accurately reflect the characteristics of uniform, exponential, normal, and lognormal distributions.

Additionally, I used several key real-world datasets to support my experiments and analysis, namely Tiny Image, Approximate Nearest Neighbor with SIFT (Ann SIFT), Nytimes, and Global Vectors for Word Representation (Glove).

Tiny Image is an image dataset that is a subset of the ImageNet dataset containing 60,000 tinted images divided into 100 categories. This dataset provides rich visual information for image processing tasks. The Ann SIFT dataset utilizes the Scale Invariant Feature Transform (SIFT) descriptor to provide key points and descriptors for image recognition and computer vision tasks. The SIFT algorithm has been widely adopted in feature extraction due to its invariance to image scaling, rotation, and luminance changes. In addition, I used the Nytimes dataset, which contains a large number of news articles from The New York Times that are tagged accordingly and are well suited for natural language processing tasks such as text categorization, topic modeling, and sentiment analysis. Finally, to provide semantic representations of word items in natural language processing tasks, I employ the Glove model, a word embedding technique based on global statistical information, which generates word vectors that capture the rich semantic information of words and is widely used in various NLP applications.

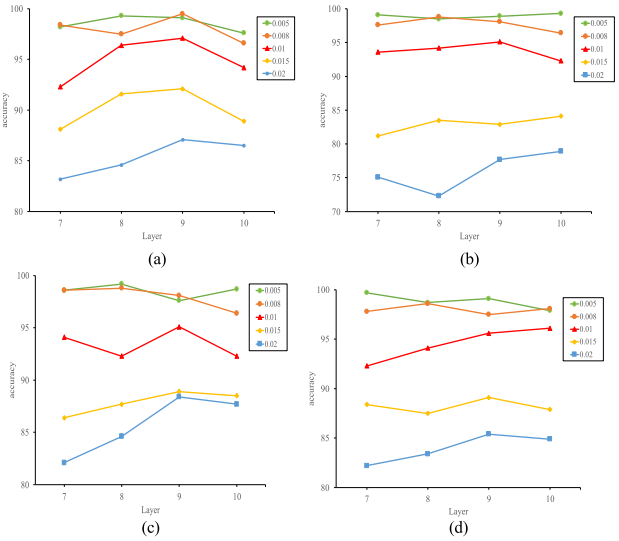

FIGURE 5. The accuracy rate of the various number of threshold α.

Each dataset has different types, scales, and dimensions, which are detailed in Table 4. The value of k is fixed at 100 for all experiments.

B. ABLATION STUDY

Various factors can impact the efficiency of the LK-index. However, the two main parameters that significantly affect its performance are the number of layers in the neural network L and the threshold value α. In this section, I will explore some ways to determine the optimal values for these parameters to enhance the effectiveness of the LK-index.

Indexing performance is generally believed to improve with an increase in the number of neural network layers. However, my experimental results have shown that this is not always the case. The query accuracy of different numbers of layers was tested on several datasets, and the results are

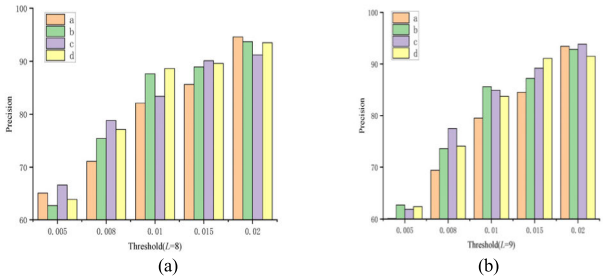

FIGURE 6. The precision rate with the different datasets of threshold α.

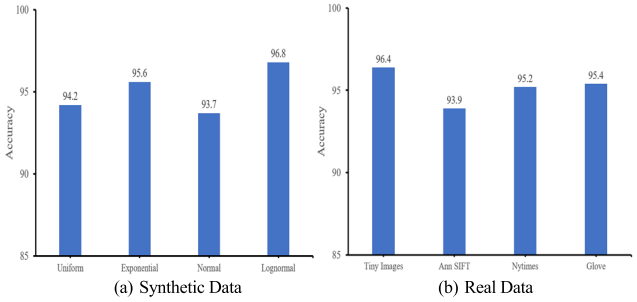

FIGURE 7. The accuracy rate on different datasets.

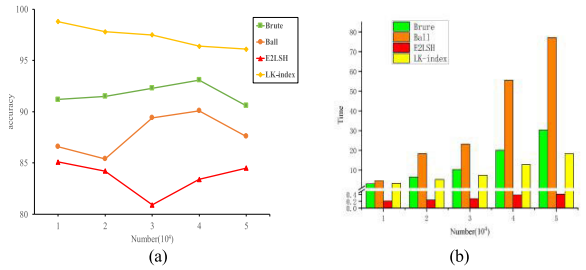

FIGURE 8. Query accuracy & Time consumption vs. data with different magnitudes.

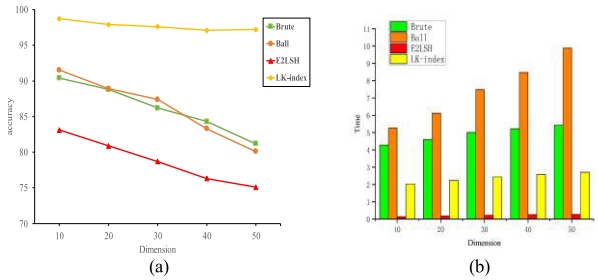

FIGURE 9. Query accuracy & Time consumption vs. data with different dimensions.

shown in Fig. 5. I used uniform(a), normal(b), and lognormal distribution(c) datasets, as well as a real image dataset Tiny Images(d). As the number of layers increases, search accuracy generally increases and then decreases, with some fluctuations in individual results. This suggests that it is important to choose a suitable number of layers to achieve better results. In Fig. 5, it can be seen that most of the search accuracy peaks when $L = 8$ or $L = 9$. Additionally, I observed that a smaller value of α leads to higher search accuracy. However, in Fig. 6, it can be observed that the *precision* decreases rapidly when α is smaller, indicating that the model's performance starts to degrade towards linear search. When α is set to 0.01, a certain search accuracy (above 90%) can be guaranteed while maintaining high precision.

Therefore, in the following experiments, I set the key parameters of the proposed LK-index as $\alpha = 0.01$ and $L = 9$ to pursue optimal performance.

C. FEASIBILITY VERIFICATION

The distribution of data in the real world can be more complicated. To test the accuracy of LK-index, I conducted experiments on four synthetic datasets and four real-world datasets. The details of the experiments are provided in Table 4, and the results for the synthetic and real-world datasets are shown in Fig. 7.

The results presented in Fig. 7(a) and 7(b) demonstrate that the LK-index, which is based on a learned index, achieves outstanding performance in kNN search. The accuracy rates for randomly generated datasets are impressive, reaching 94.2%, 95.6%, 93.7%, and 96.8% for uniform, exponential, normal, and lognormal distributions, respectively. These results indicate that the LK-index can potentially replace traditional tree structures with a high degree of accuracy. Furthermore, experiments conducted on four real-world datasets revealed that the accuracy rates can reach up to 96.4%, 93.9%, 95.2%, and 95.4% for Tiny Images, Ann SIFT, Nytimes, and Glove, respectively. This shows that the LK-index is a reliable option for both synthetic and real-world datasets, making it suitable for specific application scenarios.

D. EVALUATION OF LK-INDEX

In this section, I have designed experiments to assess the effectiveness of the LK-index in the kNN search process regarding search accuracy and time consumption. I have used data that varies in magnitude and dimensions and compared it with three traditional nearest neighbor search methods namely, Brute, Ball-tree, and E2LSH. I draw data from a uniform distribution with a magnitude range from 1×10^4 to 5×10^4 as the validation dataset and each dataset dimension is set to 20.

Fig. 8(a) demonstrates that the LK-index has a significant advantage when it comes to accuracy. When compared to the traditional tree structure, for instance, the LK-index achieves an average increase of over 9%. The comparison with the LSH-based algorithm E2LSH is even more stark, increasing by over 13%. In terms of time consumption (Fig. 8(b)), the search time of the traditional tree structure increases significantly as the data volume increases. In contrast, the LK-index maintains a lower level of increase, and its advantage becomes more apparent over time. While E2LSH has the best performance when it comes to time consumption, considering the search accuracy (Fig. 8(a)), the LK-index

undoubtedly has a greater advantage in some scenarios where more emphasis is placed on search accuracy.

I have compared the search accuracy and time consumption of the LK-index with three other methods for different data dimensions in Fig. 9. The data dimension has been set from 10 to 50 and the magnitude to 10^4. As the dimension increases, the search performance of the tree structure-based and LSH-based methods decreases faster. However, the performance of the LK-index based on the learned index only decreases slightly, which is minimally affected by the dimension and maintains a high accuracy rate in general (Fig. 9(a)). Regarding time consumption, the tree structure gradually increases with the increase of dimension due to the influence of the ''curse of dimension''. However, the LK-index and E2LSH increase more slowly (Fig. 9(b)). Compared to the traditional tree structure, the LK-index demonstrates superiority in terms of time consumption.

As previously discussed, the LK-index, which is based on the learned index, has significant advantages over the traditional tree structure. It provides better search accuracy and speed, making it a more practical solution that avoids the ''curse of dimension''. Compared to the classical LSH algorithm, while there is still a gap in time consumption, the LK-index provides superior search accuracy.

V. CONCLUSION

In the face of increasing data and dimensionality, traditional index structures encounter many challenges. To address this issue, I investigated the kNN search algorithm based on the learned index and proposed a new learned index called the LK-index. The experimental results have demonstrated the feasibility and superiority of this proposed method. In the future, further research will be conducted to develop effective methods for kNN search in high dimensional space based on the LK-index.

REFERENCES

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, ''The case for learned index structures,'' in *Proc. Int. Conf. Manage. Data*, Houston, TX, USA, May 2018, pp. 489–504.

[2] J. L. Bentley, ''Multidimensional binary search trees used for associative searching,'' *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[3] A. Guttman, ''R-trees: A dynamic index structure for spatial searching,'' in *Proc. ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, 1984, pp. 47–57.

[4] A. Arora, S. Sinha, P. Kumar, and A. Bhattacharya, ''HD-index: Pushing the scalability-accuracy boundary for approximate kNN search in high-dimensional spaces,'' 2018, *arXiv:1804.06829*.

[5] L. Li, J. Xu, Y. Li, and J. Cai, ''HCTree+: A workload-guided index for approximate kNN search,'' *Inf. Sci.*, vol. 581, pp. 876–890, Dec. 2021.

[6] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, ''Learning multi-dimensional indexes,'' in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 985–1000.

[7] M. Komorowski and T. Trzciński, ''Random binary search trees for approximate nearest neighbour search in binary spaces,'' *Appl. Soft Comput.*, vol. 79, pp. 87–93, Jun. 2019.

[8] B. Leibe, K. Mikolajczyk, and B. Schiele, ''Efficient clustering and matching for object class recognition,'' in *Proc. Brit. Mach. Vis. Conf.*, 2006, pp. 789–798.

[9] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima, ''A-tree: An index structure for high-dimensional spaces using ralative approximation,'' Dept. Inf. Sci., Tech. Rep. TR2000011, 2000.

[10] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, ''Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval,'' *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 12, pp. 2916–2929, Dec. 2013.

[11] P. Indyk and R. Motwani, ''Approximate nearest neighbors: Towards removing the curse of dimensionality,'' in *Proc. 13th Annu. ACM Symp. Theory Comput.*, 1998, pp. 604–613.

[12] V. Satuluri and S. Parthasarathy, ''Bayesian locality sensitive hashing for fast similarity search,'' 2011, *arXiv:1110.1328*.

[13] L. Wang, Y. Zhong, and Y. Yin, ''Nearest neighbour Cuckoo search algorithm with probabilistic mutation,'' *Appl. Soft Comput.*, vol. 49, pp. 498–509, Dec. 2016.

[14] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, ''Multi-probe LSH: Efficient indexing for high-dimensional similarity search,'' in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 950–961.

[15] P. Ferragina and G. Vinciguerra, ''The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds,'' *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, Apr. 2020.

[16] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, ''RadixSpline: A single-pass learned index,'' in *Proc. 3rd Int. Workshop Exploiting Artif. Intell. Techn. Data Manage.*, Jun. 2020, pp. 1–5.

[17] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska, ''ALEX: An updatable adaptive learned index,'' in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 969–984.

[18] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, ''FITing-tree: A data-aware index structure,'' in *Proc. Int. Conf. Manage. Data*, Jun. 2019, pp. 1189–1206.

[19] A. Hadian, A. Kumar, and T. Heinis, ''Hands-off model integration in spatial index structures,'' 2020, *arXiv:2006.16411*.

[20] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber, ''Designing succinct secondary indexing mechanism by exploiting column correlations,'' in *Proc. Int. Conf. Manage. Data*, Jun. 2019, pp. 1223–1240.

[21] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, ''FITing-tree: A data-aware index structure,'' 2018, *arXiv:1801.10207*.

[22] L. Li, J. Cai, and J. Xu, ''A learned index for approximate kNN queries in high-dimensional spaces,'' *Knowl. Inf. Syst.*, vol. 64, no. 12, pp. 3325–3342, Dec. 2022.

[23] R. Liu, J. Zhao, X. Chu, Y. Liang, W. Zhou, and J. He, ''Can LSH (locality-sensitive hashing) be replaced by neural network?'' *Soft Comput.*, vol. 28, no. 2, pp. 1041–1053, Jan. 2024.

[24] Z. Sun, X. Zhou, and G. Li, ''Learned index: A comprehensive experimental evaluation,'' *Proc. VLDB Endowment*, vol. 16, no. 8, pp. 1992–2004, Apr. 2023.

YONGXIN PENG received the B.S. and M.S. degrees from the School of Software, Yunnan University, in 2017 and 2020, respectively. He is currently a member of Shangluo University. His current research interests include machine learning and big data computing.