Wireless Networks (2024) 30:4195–4208
https://doi.org/10.1007/s11276-022-02927-9

A robust method based on locality sensitive hashing for K-nearest neighbors searching

Dongdong Cheng[1,2] · Jinlong Huang[1] · Sulan Zhang[1] · Quanwang Wu[3]

Accepted: 14 February 2022 / Published online: 12 March 2022

Abstract
K-nearest neighbors searching (KNNS) is to find *K*-nearest neighbors for query points. It is a primary problem in clustering analysis, classification, outlier detection and pattern recognition, and has been widely used in various applications. The exact searching algorithms, like KD-tree, M-tree, are not suitable for high-dimensional data. Approximate KNNS algorithms for high-dimensional data based on locality sensitive hashing (LSH) is becoming popular. However, the existing searching strategies are sensitive to the parameters of constructing LSH index. To solve this problem, a robust strategy for KNNS, called Robust-LSH, is proposed. It makes full use of points that frequently appear together with the query points to improve the diversity of candidates, so that it can use fewer hash tables to obtain more valuable candidates for KNNS. We do experiments on synthetic and real data. The results show that in terms of searching accuracy and running time, Robust-LSH has better performance than the p-stable LSH, RLSH and KD-tree algorithms.

Keywords Locality sensitive hashing · K-nearest neighbors searching · Approximate method · Candidate seeds

1 Introduction

K-nearest neighbors searching (KNNS) is to find the *K* closest objects to an object. It is a primary problem in clustering analysis [5, 8–10], classification [35, 36], outlier detection [15]. It has widely applications in text information retrieval, search engine, content-based information query, duplication detection, etc.

✉ Dongdong Cheng
cdd@cqu.edu.cn

Jinlong Huang
hjl@yznu.edu.cn

Sulan Zhang
slzhang@cqu.edu.cn

Quanwang Wu
wqw@cqu.edu.cn

[1] College of Big Data and Intelligent Engineering, Yangtze Normal University, Chongqing, China

[2] Chongqing Key Laboratory of Computational Intelligence, Chongqing University of Posts and Telecommunications, Chongqing, China

[3] College of Computer Science, Chongqing University, Chongqing, China

The brute-force method for KNNS first computes the distances between the query point and every other point and then selects *K* points with the minimum distance. However, it has high computational cost especially for big data.

In order to overcome this shortcoming, some tree-based indexes such as KD-tree [29], M-tree [11] cover tree [2] are used to accelerate KNNS. In [6], the authors propose a fast exact KNNS method on the basis of semi-convex hull tree and GPU. Nevertheless, these tree-based methods employ bisection techniques to construct the tree structure and the partition method may not be suitable for high-dimensional data. The reason is that the internal nodes in high-dimensional data [22, 23] have high overlap with other nodes.

Some researchers construct flat-indexes instead of tree-structure by using clustering algorithms to search K-nearest neighbors. In [32], the authors propose KMKNN algorithm for KNNS. It first divides the data into different clusters with k-means algorithm, then selects K-nearest neighbors from the closest cluster to the query object and avoids unnecessary distance computations by using triangle inequality. However, the clusters produced by K-means may be undesired, so that the KNNS result of KMKNN is undesired. In [1], the authors present a novel KNNS method on the basis of various-widths clustering, called kNNVWC. It produces compact and well separated clusters for high-dimensional data by using a novel partitioning clustering algorithm, improving the performance of KNNS.

Since the dimensionality curse makes the exact nearest neighbors searching nearly infeasible to achieve, some approximate nearest neighbor searching algorithms [24] are proposed by researchers. Approximate strategy is a good way to balance accuracy and efficiency. In [7], FLANN with the priority k-means tree, an approximate KNNS method, is introduced to improve DBSCAN. Locality sensitive hashing (LSH)-based method is also an approximate KNNS algorithm that is often used for high-dimensional data. LSH constructs hash tables by employing a family of hash functions to project similar objects into the same bucket and different objects into different buckets. When searching K-nearest neighbors of a query point, LSH-based method only needs to compare the distances between the query point and the points in the same bucket with the query point, which greatly reduces the search range. The distance between objects determines the hash functions, which makes the problem of "curse of dimensionality" easy to solve.

Among the existing LSH-based KNNS methods, the p-stable LSH [13] is a popular one. It can directly work on points in Euclidean space. The authors have proved that if the data satisfies the "bounded growth", the p-stable LSH will find its exact K-nearest neighbors in $O(\log n)$ time. However, to maintain high accuracy of p-stable LSH, large memory are required for storing hash tables. To solve this problem, Lu et al. [21] present a randomness-based LSH for KNNS, called RLSH. When searching K-nearest neighbors, it only selects a hash table, instead of all the hash tables, to project the query object. Then, it selects candidate objects from the same bucket that the query object belongs to. It has been proved that RLSH promotes the quality of the candidates even if a few hash tables are used. However, for these methods, the parameters of constructing LSH index, such as the number of hash tables L and the number of hash functions in each hash table k, will affect the result of KNNS.

To solve the above mentioned problem, we propose a robust LSH algorithm for KNNS, called Robust-LSH. It is inspired by the idea that objects closer to the query object have a higher collision probability in many hash tables, that is, the query object and its nearest neighbors appear frequently in many hash tables. We find the objects frequently appear with the query object, which is called candidate seed. Then, objects frequently collide with objects in the candidate seed are added to the candidate set. Robust-LSH can use less hash tables to search K-nearest neighbors with high quality and less query time. The experiments by comparing with KD-tree, p-stable LSH and RLSH illustrate the advantage of the proposed algorithm.

The rest of this paper is organized as follows. Section 2 reviews the related work about LSH. Section 3 introduces the p-stable LSH. After that, we present the proposed algorithm Robust-LSH in Sect. 4. Experiments and analysis are shown in Section 5. Section 6 concludes this work and discusses the future work.

2 Related work

In this section, we will introduce KNNS methods based on data structures and LSH.

To search K-nearest neighbors, several indexing data structures are proposed, such as KD-tree, M-tree, cover tree, to name a few. These tree-based algorithms build the tree by employing binary partition strategy and the similar objects are in the same leaf node. The tree-based indexes is built by recursively partitioning the data space until all the data are contained in a node. However, the binary partitioning method may not be appropriate for data whose dimensionality is high [1]. Then, these tree-based KNNS methods fail to process high-dimensional data, because they do not scale well with the increase of the dimension [33].

In order to quickly obtain the K-nearest neighbors of high-dimensional and large-scale data, approximate KNNS methods are proposed. LSH is the most notable method [19]. It was first proposed by Indyk et al.[16, 17]. Its main idea is to use hash functions to map the data into hash buckets so that similar objects have a high probability of being in the same bucket. The hash functions are dependent on different distance measures, such as $l1$ distance [17], lp distance [13], the Jaccard similarity [3] and the kernel functions [19]. An LSH for angles (i.e. cosine similarity in Euclidean space) is presented by Charikar [4], which is called SIMHASH. The authors in [25] propose an entropy-based LSH to reduce the space consumption of SIMHASH. In [12], the authors use randomized Hadamard transforms in a non-linear setting to quickly estimate the Euclidean distance for high-dimensional data. Although the space requirement is reduced, its query time is increased. On the basis of stable distribution for lp norms, Datar et al. [13] developed the p-stable LSH.

In order to maintain high recall and precision for KNNS, LSH-based method must store enough hash tables. However, as the scale of data grows, hash tables take up more space, which will influence the efficiency of searching K-nearest neighbors. Therefore, some improved methods, like the multi-probe strategy, the binary code's storage form, and the distributed LSH methods, are proposed.

The multi-probe LSH [26] is one of the methods employing multi-probe strategy. It increases the number of candidate points for a query point by looking up multiple hashing buckets that more likely contains K-nearest neighbors, so that the recall of K-nearest neighbors searching is improved, and the space and time cost is reduced. Gu et al. [14] improve the probe sequence by using a new probability model and a query-adaptive algorithm. Entropy-based LSH [25] is also a method to utilize multi-probe strategy. RLSH (Randomness-based locality-sensitive hashing) [21] randomly selects a hash table, instead of all the hash tables, to project the query object and reconstructs the candidate points for searching K-nearest neighbors. The multi-probe strategy-based algorithms reduce the memory consumption and keep comparable recall and precision.

There are also some methods convert each point into a compact binary code to reduce memory consumption, such as spectral hashing [20, 34], semi-supervised hashing [30] and self-supervised hashing [31]. Spectral hashing uses spectral graph partitioning method to learn the binary codes, which shows promising performance. Wang et al. [30] propose semi-supervised hashing method. It tries to minimize empirical error over the labeled data, and maximize variance and independence of hash bits over the labeled and unlabeled data, so that efficient hash codes are obtained. In [31], the authors greatly reduce the memory demand with a self-supervised hashing method.

Some researchers focus on developing distributed or parallel LSH algorithms. Koga et al. [18] implement LSH in distributed way. In [37], the authors propose a hadoop and collision counting based KNN algorithm, called H-c2KNN. The algorithm uses MapReduce framework to realize KNNS for high-dimensional data. It fully exploits the occurrence information of the LSH. some algorithms are proposed to solve the problems of parameters setting in constructing LSH index. Slaney et al. [28] use two histograms to optimize the parameters.

Locality-sensitive hashing methods provide LSH index for searching K-nearest neighbors of large scale and high-dimensional data. However, it requires huge space and time cost when searching nearest neighbors in candidate points, and the difficulty of setting appropriate parameters has impact on their performance. We propose a robust LSH algorithm for KNNS, which is robust to the parameters setting, and reduces the space requirement and time cost.

3 The p-stable LSH method

LSH transforms points from the original space to a new space. If two points are close to each other in the original space, it has large probability that they are mapped into the same bucket, while two points that are far from each other have a small probability of being mapped to the same bucket. LSH is used to map the original data to a new space, and nearest neighbors are often in the same bucket. Then, searching K-nearest neighbors on the whole data is transformed into searching K-nearest neighbors on a small data that is in the same bucket with the query object after mapping. The p-stable LSH [13] is a widely used LSH method. It can be directly used for data in Euclidean space without any embedding.

This section introduces p-stable LSH in detail. First, it constructs multiple hash functions from a certain hash family based on p-stable distribution. Then, only the objects that is projected into the same hash buckets as the query object are considered for inclusion in the query range. Finally, it computes the distances between the objects in the query range and the query object and obtains K-nearest neighbors of the query object by sorting the distances.

In statistics, if the linear combination of two random variables from independently and identically distribution has the same distribution as themselves, then, the random variables obey a stable distribution. Next, we will introduce p-stable distribution.

Definition 1 (p-stable distribution). For any n real data, r_1, r_2, ..., r_n and n variables, X_1, X_2, ..., X_n following the independent and identically distribution D, if there exists $p \geq 0$ such that the random variables $\sum_i r_i X_i$ and $\left(\sum_i |r_i|^p\right)^{1/p} X$ are from the same distribution, then the distribution D is called p-stable distribution.

It has been proved that for any $p \in (0, 2]$, the stable distribution exists. A Gaussian distribution D_G with the density function $g(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$ is a 2-stable distribution. Therefore, p-stable LSH often uses Gaussian distribution to generate the hash functions.

Definition 2 (Locality sensitive hash). Any two points p, q are in the d-dimensional real data space. A function family $F = \{f : S \rightarrow U\}$ is (d_1, d_2, p_1, p_2)-sensitive hash family if it satisfies the conditions that when the distance between a and b in original space S is less than d_1, the probability of collision between a and b in new space U (i.e. $f(p) = f(q)$) is larger than p_1, and when the distance between a and b in the original space S is larger than d_2, the probability of collision between p and q (i.e. $f(p) = f(q)$) is less than p_2. Where $f(p)$ is the hash value of p after being mapped by $f \in F$.

It should be noted that the hash family F can be used if it satisfies that $p_1 > p_2$ and $d_1 < d_2$. It means that two points whose distance is less than d_1 are more likely to collide.

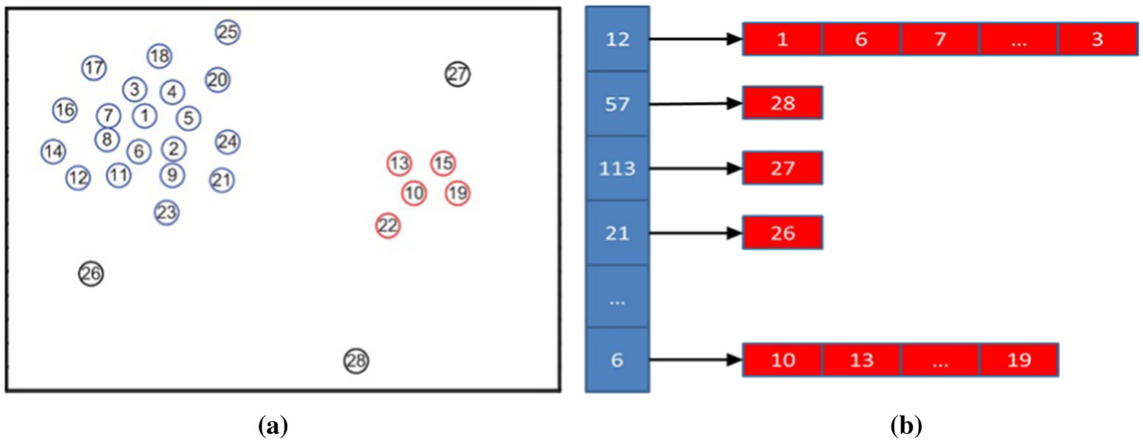

Fig. 1 An example of hash table. **a** The original data; **b** The corresponding hash table

LSH often concatenates multiple hash functions generated from a certain hash family to avoid false detection.

For a given integer k, we obtain the concatenating function $g(p) = (f_1(p), f_2(p), , f_k(p))$. The k hash functions are from the hash family $F = \{f : S \rightarrow U^k\}$, where $f_i \in F$. Therefore, when the distance between p and q is less than d_1, the probability of collision between p and q in new space U (i.e. $g(p) = g(q)$) is larger than ${p_1}^k$, and when the distance between p and q is larger than d_2, the probability of collision between p and q (i.e. $g(p) = g(q)$) is less than ${p_2}^k$.

Concatenating multiple hash functions decreases the collision probability of two points far apart, but at the same time it may reduce the collision probability of two close points. Therefore, multiple hash tables are required for solving this problem. We use L functions g, i.e., g_1, g_2, ..., g_L, and each function g_i is the concatenating of k hash functions. Each hash function is computed as:

$$f_{a,b}(v) = \lfloor \frac{a \cdot v + b}{W} \rfloor \quad (1)$$

In Eq. 1, v is the object in a d-dimension real data space, a and b are the parameters of a hash function, and W is the width of projection. a is a vector having the same dimensions as v and it is generated from Gaussian distribution. b is a real number, representing offset value, and it is generated from uniform distribution [0, W].

The steps of searching K-nearest neighbors based on p-stable LSH mainly include constructing hash tables and searching nearest neighbors for the query point. When constructing one hash table, it first randomly generates k vectors a and offset b. After that, each point is projected with Eq. 1 and obtained k hash values. Then, the k hash values are concatenated and converted to one integer. In a hash table, the points that are close to each other are assigned to the same bucket. Then it repeats the above steps L times to build L hash tables. The detailed process of constructing LSH index is shown in Algorithm 1.

Figure 1 gives an example of the hash table. Figure 1(a) is the original data and Figure 1(b) is a hash table obtained by mapping the original data. We can learn from the figure that points 1, 6, 7, 3, etc are mapped to the same bucket because their distance is relatively close. Similarly, points 10, 13, 22 and 19 are assigned to the same bucket. However, there are no other points who are assigned to the same bucket with points 26, 27 and 28.

When searching nearest neighbors, p-stable LSH method uses the same hash functions in each hash table to compute the hash value of the query point and obtains the candidates by finding the union of points having the same hash value as the query point in L hash tables. Then, the distances between the candidates and the query point are calculated and sorted to get K-nearest neighbors of the query point. The detailed procedure of searching K-nearest neighbors is described in Algorithm 2.

From the above algorithms, we can learn that the influence of p-stable LSH for searching K-nearest neighbors is influenced by three parameters: k, W, and L. A larger value of k requires more time of calculating hash values and decreases the probability that points who are far apart are assigned into the same bucket. The value of W affects the probability of collision for any two points and a bigger value will increase the probability, while a smaller value will reduce the probability. The bigger the value of L is, the higher the accuracy of K-nearest neighbors searching is, but it takes a longer time to construct the hash tables.

Algorithm 1: LSHIndex-Consturcting

```
Input: $X$: the dataset, $k$: the number of hash functions in each hash table, $L$: the
       number of hash tables, $W$: the width of the projection
Output: $HashTable$
Initializing: $n$=$|X|$, $HashTable$=$\phi$;
for $i = 1$ to $L$ do
    for $j = 1$ to $k$ do
        Generate vector $a_{ij}$ from the Gaussian distribution $N(0, 1)$;
        Generate offset $b_{ij}$ from the uniform distribution $[0, W]$;
        $HashTable(i).a = a_i$;
        $HashTable(i).b = b_i$;
    end
end
for each point $x_k$ in $X$ do
    for $i = 1$ to $L$ do
        for $j = 1$ to $k$ do
            $h(j) = \lfloor \frac{a \cdot x_n + b}{W} \rfloor$;
        end
        convert the $k$ hash values to one integer $Ind$;
        $HashTable(i).bucket(Ind) = k$; //put the index of point into the bucket;
    end
end
```

The time consumption of p-stable LSH mainly contains the time of constructing hash tables, obtaining candidates and computing the distances. Let T_p denote the time consumption of a single projection operation, T_u denote the time of an union operation, and T_d denote the time for calculating the distance between each pair of points. The time cost of constructing hash tables is $n * k * L * T_p$, where n is the data size. In the process of obtaining candidates, it includes mapping the query point and finding the union of points having the same hash value as the query point. Its running time is $k * L * T_p + L * T_u$. Assuming the number of candidates is n_c, then the time cost of computing distances is $n_c * T_d$. Thus, the time consumption of p-stable LSH is $n * k * L * T_p + k * L * T_p + L * T_u + n_c * T_d$. The space consumption of p-stable LSH is mainly for storing the hash tables, and it is $n * L$.

Algorithm 2: KNNS

```
Input: $HashTable$:the hash tables, $q$: the query point, $K$: the number of nearest
       neighbors
Output: $KNN$
Initializing: $Candidates$=$\phi$;
for $i = 1$ to $L$ do
    $a$=$HashTable(i).a$;
    $b$=$HashTable(i).b$;
    for $j = 1$ to $k$ do
        $h(j) = \lfloor \frac{a \cdot q + b}{W} \rfloor$;
    end
    convert the $k$ hash values to one integer $Ind$;
    $TmpSet$=$HashTable(i).bucket(Ind)$;
    $Candidates$=$Candidates \bigcup TmpSet$;
end
for each candidate $c$ in $Candidates$ do
    $d(i)$=ComputeDistance($q$,$c$);
end
$sortedInd$=sort($d$,'ascend');
$KNN$=$Candidates(sortedInd(1 : k))$;
```

4 The proposed method

The p-stable LSH algorithm has to set three parameters k, L and W to construct LSH index. To keep high accuracy of searching K-nearest neighbors, it has to construct multiple hash tables to obtain enough candidates, which requires huge space consumption. Besides, the p-stable LSH always selects points that having the same hash value with the query point which limits the diversity of the candidates. To solve these problems, we propose a robust LSH algorithm, called Robust-LSH. It is inspired by the idea that points close to the query point have more collisions in different hash tables. Based on this idea, we improve the strategy for KNNS. The proposed algorithm can obtain the same accuracy with fewer hash tables, which reduces the space consumption.

4.1 Robust-LSH method

The proposed method first constructs hash tables like p-stable LSH, and then uses a different strategy for KNNS. and uses these candidate seeds to obtain a high quality candidate set for searching K-nearest neighbors.

Definition 3 (Candidate seeds). For a query point q, its candidate seeds are points that having the same hash value as q in multiple hash tables.

To find the candidate seeds of the query point, we repeatedly randomly select several hash tables to compute the hash value of the query point, and points that having the same hash value with the query point are chosen for intersection operation. When the number of points in the intersection is less than K (K is the number of nearest neighbors), it terminates. The detailed algorithm of finding the candidate seeds is shown in Algorithm 3.

After obtaining the candidate seeds, we find the neighbors of each candidate seed. Then, these neighbors are combined into a candidate set for searching K-nearest neighbors of the query point. Next, we give the definitions of the neighbors of a candidate seed and candidates of a query point.

Algorithm 3: CS-Searching

```
Input: $HashTable$:the hash tables, $q$: the query point, $K$: the number of nest
       neighbors
Output: $CS$:the candidate seeds
Initializing: $CS=\phi$;
Randomly generate a sequence $RS$ of values from 1 to $L$;
$m$=1;
while $length(CS) \geq K$ do
    $i = RS(m)$;
    $a = HashTable(i).a$;
    $b = HashTable(i).b$;
    for $j = 1$ to $k$ do
        $h(j) = \lfloor \frac{a \cdot q + b}{W} \rfloor$;
    end
    convert the $k$ hash values to one integer $Ind$;
    $TmpSet$=$HashTable(i).bucket(Ind)$;
    if $CS == \phi$ then
        $CS = TmpSet$;
    end
    $CS = CS \bigcap TmpSet$;
    $m = m + 1$;
end
```

In the process of KNNS, p-stable LSH method only takes advantage of the points that having the same hash value as the query point. Considering that points that frequently appear together with the query point in multiple hash tables have high value in searching K-nearest neighbors, we improved the diversity of candidates by employing these points, which is called candidate seeds in this paper,

Definition 4 (The neighbors of a candidate seed). For a candidate seed S_i, its neighbors are the points having the same hash value as S_i in a random hash table H_r, which is denoted as SCS_i.

Definition 5 (Candidates of a query point). For a query point q, its candidates are the union of the neighbors of every candidate seed, which is denoted as $Candidates(q) = \bigcup_{i=1}^{ns} SCS_i$, where ns is the number of candidate seeds.

The proposed algorithm includes two steps: constructing hash tables and searching K-nearest neighbors for query points. Our work is mainly to improve the second step. Candidate seeds are points frequently appear together with the query points, and it is valuable for obtaining K-nearest neighbors of the query point. Therefore, making full use of candidate seeds will improve the diversity of candidates, so that we can use fewer hash tables to obtain more valuable candidates for KNNS. The process of Robust-LSH for KNNS is detailed in Algorithm 4.

Algorithm 4: Robust-LSH

```
Input: $HashTable$:the hash tables, $q$: the query point, $K$: the number of nest
       neighbors
Output: $KNN$:the K-nearest neighbors
Initializing: $Candidates=\phi$;
$CS$=CS-Searching($HashTable$,$q$,$K$); for each seed $S$ in $CS$ do
    Randomly select a hash table $HashTable_r$;
    $a=HashTable_r.a$;
    $b=HashTable_r.b$; for $j = 1$ to $k$ do
        $h(j) = \lfloor \frac{a \cdot S + b}{W} \rfloor$;
    end
    convert the $k$ hash values to one integer $Ind$;
    $TmpSet=HashTable_r.bucket(Ind)$;
    $Candidates=Candidates \bigcup TmpSet$;
end
for each candidate $c$ in $Candidates$ do
    $d(i)$=ComputeDistance($q$,$c$);
end
$sortedInd$=sort($d$,'ascend');
$KNN=Candidates(sortedInd(1:k))$;
```

4.2 Complexity analysis

For the proposed algorithm, its running time is mainly spent on constructing hash tables, obtaining the candidates, and computing the distances between the query point and the candidates. Let T_p denote the time consumption of a single projection operation, T_i denote the upper bound of the cost for an intersection operation, T_u denote the cost for an union operation, and T_d denote the time for calculating the distance between each pair of points. When constructing hash tables, each point needs to be projected $k * L$ times, then the time consumption of constructing hash tables is $n * k * L * T_p$ (n is the number of objects in a data). To obtain the candidates of the query point, we need to randomly select several hash tables to project the query point and obtain candidate seeds through the intersection operation. If the number of selected hash tables is L' ($L' < L$), the time consumption for obtaining candidate seeds is $k * L' * T_p + L' * T_i$. If the number of candidate seeds is n_s ($n_s \ll n$), then the upper bound time required for the union operation is $n_s * T_u$. Thus the running time of obtaining the candidates of the query point is $k * L' * T_p + L' * T_i + n_s * T_u$. The time for calculating the distances is $n_c * T_d$, where n_c ($n_c \ll n$) is the number of candidates. Thus, Robust-LSH's time consumption is $n * k * L * T_p + k * L' * T_p + L' * T_i + n_s * T_u + n_c * T_d$. The space of the proposed algorithm is mainly used for storing hash tables, therefore its space complexity is $n * L$.

Compared with the p-stable LSH, although Robust-LSH requires more time to find candidate seeds, it only needs to build a few hash tables to keep the accuracy of K-nearest neighbors searching.

5 Experiments and analysis

The performance of the proposed algorithm Robust-LSH is evaluated by comparing with KD-tree, p-stable LSH and RLSH on synthetic and real datasets[1]. KD-tree method is a typical tree-based nearest neighbors searching algorithm. It constructs tree structure for exact nearest neighbors searching. The p-stable LSH is a popular LSH method. RLSH is an improved p-stable LSH algorithm.

First, we prove the robustness of our method to the parameters when constructing LSH hash tables. Then, we explore the influence of data size and dimension on the algorithms. Finally, we use two real datasets to illustrate the robustness of Robust-LSH.

[1] Data will be available on reasonable request.

Table 1 The ACC and running time of algorithms with different L settings

		5		10		15		20		25	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
p-stable LSH	ACC	0.046	0.000	0.175	0.000	0.525	0.000	0.916	0.000	1.000	0.000
	time	0.334	0.003	0.472	0.003	0.571	0.007	0.621	0.004	0.645	0.003
RLSH	ACC	0.975	0.027	0.995	0.015	0.981	0.025	0.991	0.020	0.995	0.015
	time	16.730	2.155	16.267	3.339	17.429	3.504	20.010	3.479	21.734	3.841
Robust-LSH	ACC	0.814	0.066	0.994	0.013	1.000	0.000	1.000	0.000	1.000	0.000
	time	1.542	0.190	1.330	0.081	1.328	0.075	1.431	0.082	1.403	0.104
		30		35		40		45		50	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
p-stable LSH	ACC	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000
	time	0.667	0.003	0.692	0.018	0.731	0.023	0.744	0.003	0.771	0.019
RLSH	ACC	1.000	0.000	1.000	0.000	0.990	0.031	0.997	0.008	0.995	0.016
	time	23.119	4.434	25.936	3.822	27.750	4.218	25.900	2.229	25.130	5.815
Robust-LSH	ACC	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000
	time	1.443	0.080	1.480	0.052	1.525	0.102	1.527	0.052	1.584	0.084

We quantitatively evaluate the quality of KNNS by using accuracy (ACC). Given a query point q, $A(i)$ represents the actual i-th nearest neighbor of q and $R(i)$ represents the i-th nearest neighbor of q obtained by the algorithm. ACC is calculated Eq. 2.

$$ACC(q) = \frac{\sum_{i=1}^{K} \chi(A(i) - R(i))}{K} \tag{2}$$

In the above equation,

$$\chi(x) = \begin{cases} 1, & x = 0 \\ 0, & x \neq 0 \end{cases}.$$

ACC means the proportion of correct neighbors to actual neighbors. The range of ACC is [0, 1]. When ACC equals to 1, it tells that all the correct K nearest neighbors are found by the algorithm. We do experiments on a PC with Intel Core i7 1.8GHz, 8GB RAM, Windows 10 and MATLAB R2013a.

5.1 The influence of hash tables *L*

A Gaussian distribution is used to randomly generate a dataset containing 20000 objects and each object has 100 dimensions. 20 objects are selected as the query points and the 100-nearest neighbors of each query point are used to compute ACC of algorithms.

When constructing LSH index, we have to set three parameters: k, W, and L. To explore the influence of L for p-stable LSH, RLSH and Robuts-LSH, we fix k and W, and range the number of hash tables L form 5 to 50. According to [13], W is fixed as 4 and k is set to 5.

The ACC and running time are used to evaluate the performance of the algorithms. Since RLSH and Robust-LSH randomly select hash tables to construct candidates for the query point, each query (containing 20 query points) is repeated 10 times. Table 1 shows the mean and standard deviation of ACC and running time of 10 experiments. The average ACC and running time are shown in Figure 2.

The standard deviations in Table 1 illustrate that the ACC and running time of p-stable LSH and Robust-LSH are stable. From Figure 2(a), we can learn that as for p-stable LSH, its ACC score increases with hash tables when the number of hash tables is less than 25. Especially, when there are only a few hash tables, its ACC score is far less than that of RLSH and Robust-LSH. For RLSH and Robust-LSH, even though a few hash tables are used, their ACC scores are higher than 0.9. When the hash tables is greater than 10, the ACC score of Robust-LSH always maintains 1, which shows that Robust-LSH is robust to the parameter L. Besides, it means that we can use fewer hash tables to get good results and fewer hash tables means less storage space. From Figure 2(b), we can learn that Robust-LSH takes a little longer time than p-stable LSH and far less time than RLSH. Therefore, our algorithm Robust-LSH is much better than p-stable LSH and RLSH in terms of the storage space, query accuracy and running time.

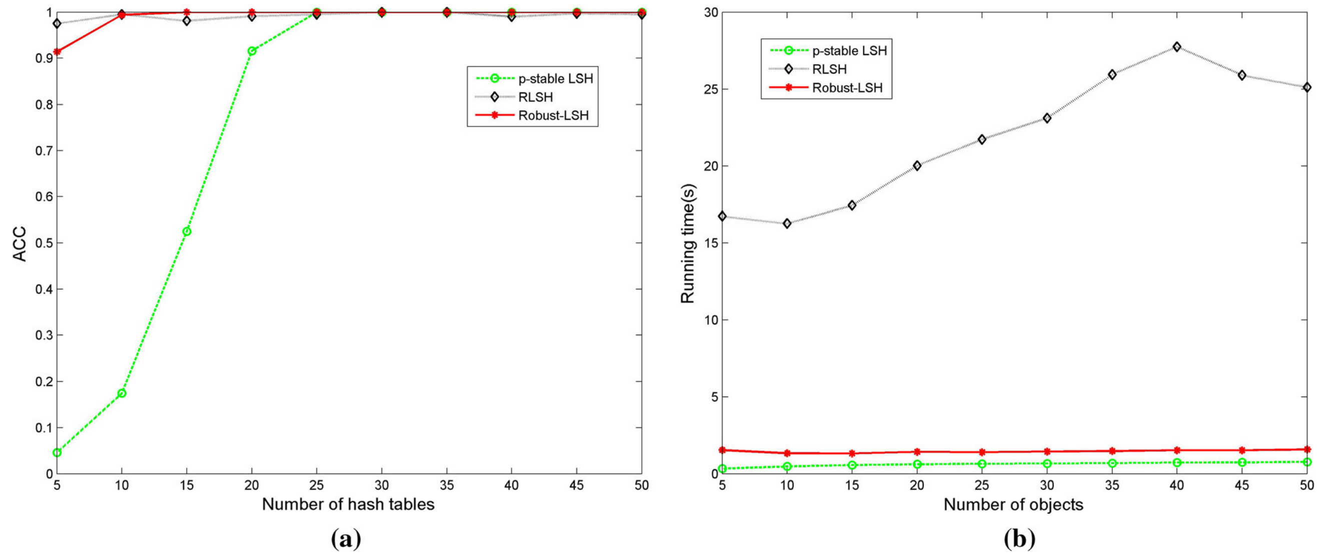

Fig. 2 The impact of hash tables on ACC and running time for p-stable LSH, RLSH and Robust-LSH. **a** The ACC with the increase of hash tables. **b** The running time with the increase of hash tables

5.2 The influence of data size and dimension

We also compare our algorithm with KD-tree and p-stable LSH on Gaussian distribute datasets to illustrate the robustness of the proposed algorithm to the dimension and size of data. KD-tree algorithm is a typical tree-based method for searching K-nearest neighbors. When doing experiment on large-scale data, since RLSH algorithm takes a lot of time to obtain the result, we only select p-stable LSH method for comparison. Like the first experiment, we repeat each query for 10 times, and each time we try to find 20 query points' 100 nearest neighbors, and use the average ACC and running time of 10 times as the evaluation index for the performance of algorithms.

First, we randomly generate 100000-sized Gaussian distribution data, and the dimension gradually increases from 100 to 1000. According to the first experiment, we can learn that when L is set to 10, it not only saves the space storage, but also achieve a higher ACC value for the proposed method. Therefore, in this experiment, we fixed L as 10. The parameter k is also set to 5. Since the data scale is much larger than that of the first experiment, we increase W to 10. Table 2 lists the mean and standard deviation of ACC and running time. Besides. Figure 3 shows the average ACC and running time as the dimension of data increases.

The standard deviations of Robust-LSH in Table 2 are small, which tells that the results of the proposed algorithm is stable. From Figure 3(a), we find that as the dimension increases, the ACC scores of KD-tree and our method are close to 1, but the ACC of p-stable LSH decrease from 0.996 to 0.0667. KD-tree is an exact method for searching K-nearest neighbors, and our method is an approximate method. From this result, our method almost obtains the same accurate K-nearest neighbor as KD-tree. Figure 3(b) tells that the running time of KD-tree increases much faster than p-stable LSH and our method with the increase of dimension. Therefore, considering ACC and running time, Robust-LSH is rarely affected by the dimensions.

Then, we randomly generate 1000-dimensional Gaussian distribution data, and the size of data ranges from 10000 to 100000. Like the above experiment, three parameters L, k and W are set to 10, 5, and 10, respectively. Table 3 shows the mean and standard deviation of ACC and running time with different sizes. Figure 4 shows the average ACC and running time as the size of data increases.

Figure 4(a) displays the ACC scores of the algorithms with the increase the data size and Figure 4(b) displays the running time of the algorithms. It can be seen that the ACC scores of p-stable LSH are kept lower than 0.1, which proves that p-stable LSH is not effective for these data. Since KD-tree is an exact method, its ACC scores maintain 1. The ACC scores of Robust-LSH are always higher than 0.97 and close to 1, which illustrates its effectiveness when searching K-nearest neighbors. The running time shown in Figure 4(b) tells that the running time of the three algorithms increases as the data size grows. However, the growth rate of KD-tree is much faster than p-stable LSH and Robust-LSH. Although Robust-LSH takes a little longer time than p-stable LSH, its ACC scores are much higher than p-stable LSH. Hence, combining these two

Table 2 The ACC and time of algorithms with different dimensions

		100		200		300		400		500		600		700		800		900		1000	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
KD-tree	ACC	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000
	time	8.526	0.172	14.962	0.084	21.572	0.147	28.780	0.130	35.782	0.083	44.138	1.206	53.716	1.521	65.481	2.591	81.273	3.125	97.830	4.821
p-stable LSH	ACC	0.996	0.012	0.940	0.111	0.816	0.152	0.657	0.171	0.406	0.163	0.166	0.066	0.177	0.099	0.149	0.101	0.093	0.060	0.066	0.015
	time	2.928	0.327	4.794	0.069	6.241	0.146	7.613	0.140	9.009	0.258	9.774	0.444	10.897	0.732	12.812	0.856	12.550	1.021	13.737	0.902
Robuts-LSH	ACC	1.000	0.000	1.000	0.000	1.000	0.000	0.996	0.011	0.998	0.005	0.999	0.004	0.996	0.012	0.991	0.018	0.990	0.016	0.971	0.044
	time	7.639	0.965	10.350	0.863	11.154	0.550	12.584	0.614	13.657	0.614	14.527	0.527	17.191	0.675	16.738	0.708	18.755	0.421	19.944	0.777

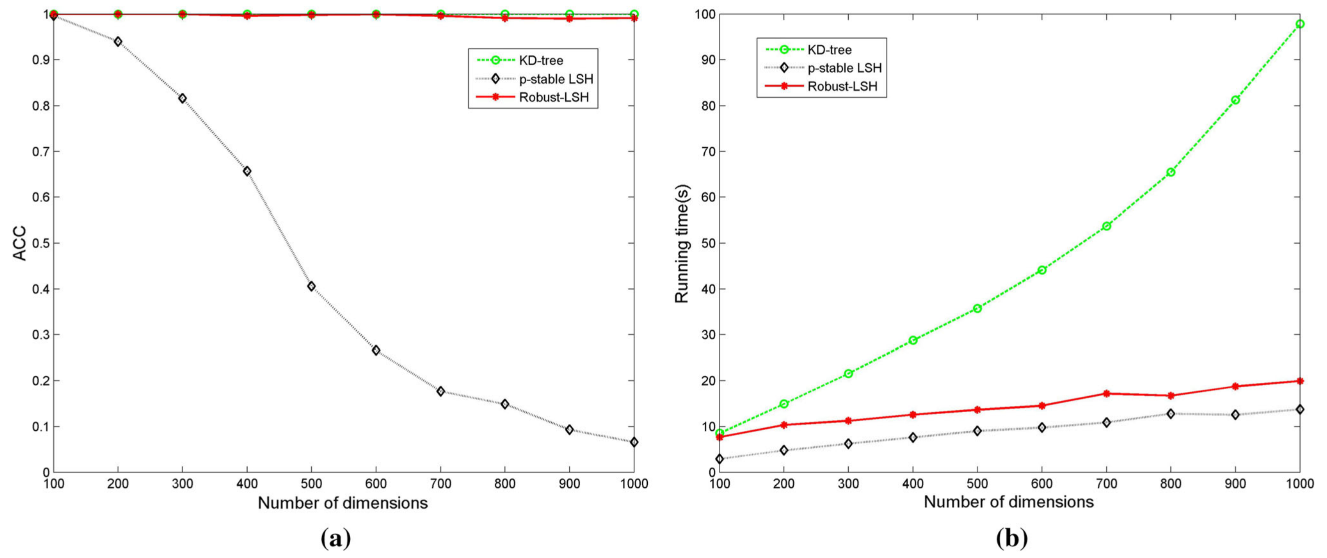

Fig. 3 The impact of dimensions on ACC and running time for KD-tree, p-stable LSH and Robust-LSH. **a** The ACC with the increase of dimensions. **b** The running time with the increase of dimensions

indicators, Robust-LSH outperforms KD-tree and p-stable LSH.

5.3 The experiment on real datasets

We further use two real datasets Patches and MNIST, which are from [27], to illustrate the effectiveness of Robust-SLH. Patches consists of 59500 points and each has 400 dimensions. MNIST contains 60000 points and each has 784 dimensions. The compared algorithms include p-stable LSH, RLSH and KD-tree. When constructing LSH index for p-stable LSH, RLSH and Robust-LSH, we set L to 10 like the previous setting, then roughly estimate W by divide the range of the data by 4, and range the number of hash functions k from 4 to 10. We still use 20 query points' 100 neighbors to compute ACC scores of algorithms. The average ACC and running time of 10 experiments are used as the evaluate indexes. The running time of p-stable LSH, RLSH and Robust-LSH includes that of

Table 3 The ACC and time of algorithms with different sizes

		10000		20000		30000		40000		50000	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
KD-tree	ACC	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000
	time	5.523	0.048	11.794	0.106	17.894	0.121	25.775	0.123	33.120	0.394
p-stable LSH	ACC	0.051	0.019	0.063	0.017	0.065	0.034	0.059	0.023	0.057	0.022
	time	1.943	0.105	3.215	0.130	4.205	0.377	5.324	0.316	6.390	0.436
Robuts-LSH	ACC	0.948	0.046	0.982	0.020	0.974	0.023	0.978	0.029	0.971	0.034
	time	2.979	0.052	5.071	0.133	6.511	0.176	8.371	0.187	9.991	0.318
		60000		70000		80000		90000		100000	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
KD-tree	ACC	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000
	time	40.378	0.337	54.929	1.090	65.841	2.078	80.950	1.258	95.364	2.356
p-stable LSH	ACC	0.066	0.027	0.044	0.009	0.069	0.021	0.057	0.017	0.058	0.015
	time	7.817	0.570	7.674	1.012	10.460	0.919	11.423	0.879	12.604	0.508
Robuts-LSH	ACC	0.989	0.014	0.984	0.016	0.979	0.024	0.987	0.020	0.981	0.026
	time	12.071	0.571	12.853	1.233	15.964	0.323	18.477	0.417	20.274	1.079

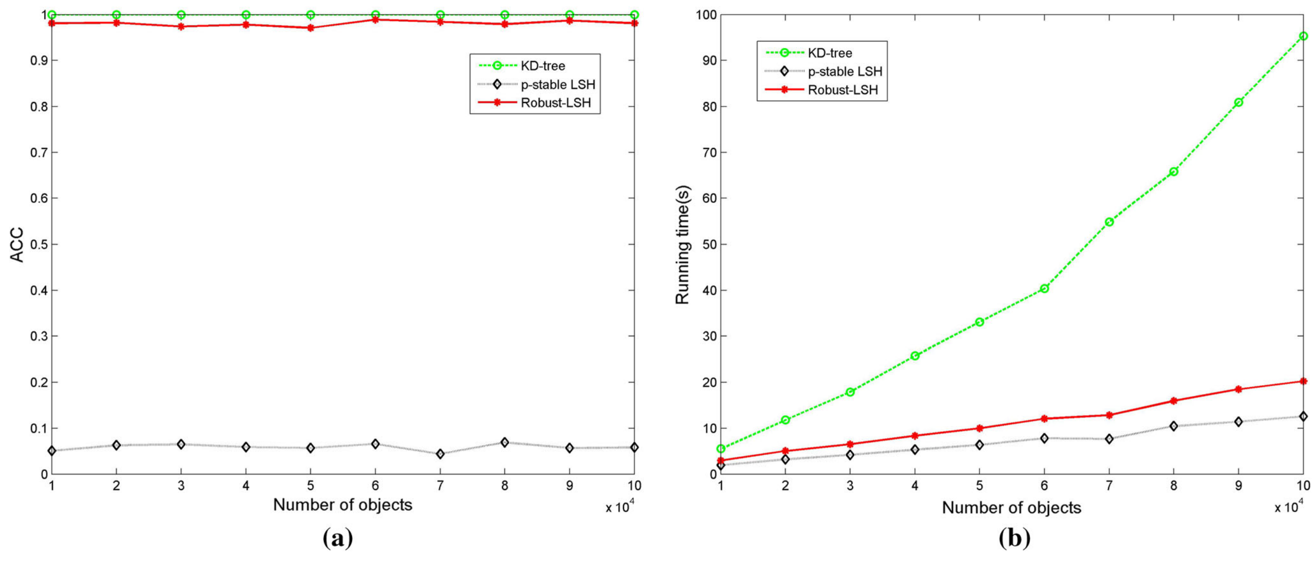

Fig. 4 The impact of dimensions on ACC and running time for KD-tree, p-stable LSH and Robust-LSH. **a** The ACC with the increase of data size. **b** The running time with the increase of data size

building the hash tables and searching K-nearest neighbors. Table 4 lists the mean and standard deviations of ACC and running time for Patches and Table 5 lists the mean and standard deviations of ACC and running time for MINIST.

The results in Table 4 and 5 tell that smaller k value takes longer time and obtains higher ACC score. The reason is that smaller k value produces more candidates for searching K-nearest neighbors. With the increase of k, ACC scores of three LSH methods decrease. Specifically, for Patches, p-stable LSH decreases from 1 to 0.565, RLSH decreases from 1 to 0.926, and Robust-LSH decrease from 1 to 0.941; for MINIST, p-stable LSH decreases from 0.975 to 0.310, RLSH decreases from 1 to 0.825, and Robust-LSH decrease from 1 to 0.864. Obviously, the ACC scores of Robust-LSH drop the slowest compared with p-stable LSH and RLSH, which demonstrates that Robust-LSH is robust to the parameter k. Besides, no matter what the value of k is, the proposed method obtains higher ACC scores than p-stable LSH and RLSH, and Robust-LSH's running time is far less than RLSH. For KD-tree, its ACC scores are always 1, because it is an exact K-nearest neighbors searching method. However, its running time for

Table 4 The ACC and running time of algorithms on Patches

		k=4		k=6		k=8		k=10	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std
p-stable LSH	ACC	1.000	0.000	0.859	0.000	0.727	0.000	0.565	0.000
	time	4.705	0.108	4.092	0.076	3.008	0.009	2.339	0.036
RLSH	ACC	1.000	0.000	0.993	0.016	0.930	0.042	0.926	0.044
	time	285.211	45.654	139.870	34.991	62.580	18.875	30.535	8.293
Robust-LSH	ACC	1.000	0.000	0.995	0.015	0.975	0.024	0.941	0.039
	time	10.446	1.375	8.598	0.857	6.713	0.376	7.310	0.627
KD-tree	ACC	1.000							
	time	13.896							

Table 5 The ACC and running time of algorithms on MINIST

		k=4		k=6		k=8		k=10	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std
p-stable LSH	ACC	0.975	0.000	0.769	0.000	0.559	0.000	0.310	0.000
	time	11.572	0.233	9.381	0.367	6.324	0.199	5.174	0.114
RLSH	ACC	1.000	0.000	1.000	0.000	0.983	0.022	0.925	0.039
	time	786.182	177.045	109.535	13.709	42.736	7.718	29.440	6.364
Robust-LSH	ACC	1.000	0.000	0.998	0.006	0.949	0.031	0.864	0.049
	time	15.513	0.508	13.343	0.245	12.542	0.337	12.328	0.855
KD-tree	ACC	1.000							
	time	16.521							

these two real data sets is longer than Robust-LSH. Therefore, in terms of ACC and running time, the proposed method Robust-LSH is more effective than other KNNS algorithms.

6 Conclusions and future work

LSH is a popular approximate KNNS method for high-dimensional data. However, the searching strategy of p-stable LSH is sensitive to the parameters in constructing LSH index. RLSH improves the query strategy by randomly selecting hash tables to project the query point, but it has to take much longer time to obtain a desired result. In this paper, we propose a novel strategy to search K-nearest neighbors, called Robust-LSH, which is robust to the parameters of constructing LSH index. It makes full use of points that frequently appear together with the query points to improve the diversity of candidates, so that it can use fewer hash tables to obtain more valuable candidates for searching K-nearest neighbors. The experimental results illustrate the robustness and effectiveness of our proposed algorithm.

The proposed algorithm is robust to the number of hash tables and we can use fewer hash tables to obtain high-quality of candidates for KNNS. However, the number of candidates obtained by Robust-LSH is larger than that of p-stable LSH, thus it takes a little longer time than p-stable LSH.

In the future, we will explore the parallel implementation of the algorithm to further improve the efficiency of Robust-LSH. At the same time, we will also explore its application in clustering analysis.

Acknowledgements This work is supported in part by National Natural Science Foundation of China under Grant 62006029, 62172065, 62106024, in part by Postdoctoral Innovative Talent Support Program under Grant CQBX2021024, in part by Natural Science Foundation of Chongqing (China) under Grant cstc2019jcyj-msxmX0683, cstc2019jcyj-msxmX0838, cstc2019jcyj-msxmX0871, cstc2020jscx-lyjsAX0008, cstc2019jcyj-cxttX0002, cstc2021ycjh-bgzxm0013, and in part by Project of Chongqing Municipal Education Commission, China under Grant KJQN202001434, KJQN202001442, KJQN201901408, KJZDM20190140, HZ2021008.

References

1. Almalawi, A. M., Fahad, A., Tari, Z., Cheema, M. A., & Khalil, I. (2016). *k* nnvwc: An efficient *k* -nearest neighbors approach based on various-widths clustering. *IEEE Transactions on Knowledge and Data Engineering, 28*(1), 68–81.
2. Beygelzimer, A., Kakade, S., & Langford, J. (2006). Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on machine learning, ICML '06*, p. 97C104. Association for Computing Machinery, New York, NY, USA.
3. Broder, A. Z., Charikar, M., & Frieze, A. M. (2000). Min-wise independent permutations. *Journal of computer and system sciences, 60*(3), 630–659.
4. Charikar, Moses, S. (2002). Similarity estimation techniques from rounding algorithms. In *The 34th symposium on theory of computing (STOC)*, pp. 380–388.
5. Chen, Y., Hu, X., Fan, W., Shen, L., Zhang, Z., Liu, X., Du, J., Li, H., Chen, Y., & Li, H. (2020). Fast density peak clustering for large scale data based on knn. *Knowledge-Based Systems, 187*, 104824.
6. Chen, Y., Zhou, L., Bouguila, N., Zhong, B., Wu, F., Lei, Z., Du, J., & Li, H. (2018). Semi-convex hull tree: Fast nearest neighbor queries for large scale data on gpus. In *2018 IEEE International Conference on Data Mining (ICDM)*, pp. 911–916 . https://doi.org/10.1109/ICDM.2018.00110
7. Chen, Y., Zhou, L., Pei, S., Yu, Z., Chen, Y., Liu, X., Du, J., & Xiong, N. (2019). Knn-block dbscan: Fast clustering for large-scale data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. https://doi.org/10.1109/TSMC.2019.2956527
8. Cheng, D., Huang, J., Zhang, S., Zhang, X., & Luo, X. (2021). A novel approximate spectral clustering algorithm with dense cores and density peaks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems.*https://doi.org/10.1109/TSMC.2021.3049490
9. Cheng, D., Zhu, Q., Huang, J., Wu, Q., & Yang, L. (2019). A novel cluster validity index based on local cores. *IEEE Transactions on Neural Networks and Learning Systems, 30*(4), 985–999.
10. Cheng, D., Zhu, Q., Huang, J., Wu, Q., & Yang, L. (2021). Clustering with local density peaks-based minimum spanning tree. *IEEE Transactions on Knowledge and Data Engineering, 33*(2), 374–387. https://doi.org/10.1109/TKDE.2019.2930056
11. Ciaccia, P., Patella, M., & Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97*, p. 426C435. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
12. Dasgupta, A., Kumar, R., & Sarlos, T. (2011). Fast locality-sensitive hashing. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1073–1081. Association for Computing Machinery, New York, NY, USA.
13. Datar, M., Immorlica, N., Indyk, P., & Mirrokni, V. (2004). Locality sensitive hashing scheme based on p-stable distributions. socg '04. In *SCG'04 proceedings of the twentieth annual symposium on computational Geometry*, pp. 253–262.
14. Gu, X., Zhang, L., Zhang, D., Zhang, Y., Li, J., & Bao, N. (2012). Query range sensitive probability guided multi-probe locality sensitive hashing.
15. Huang, J., Zhu, Q., Yang, L., & Ji, F. (2015). A non-parameter outlier detection algorithm based on natural neighbor. *Knowledge-Based Systems, 92*, 71–77.
16. Indyk, P., Motwani, R., Raghavan, P., & Vempala, S. (1997). Locality-preserving hashing in multidimensional spaces. In *The 29th Symposium on Theory of Computing (STOC)*, pp. 618–625.
17. Indyk, P., R., M. (1998). Approximate nearest neighbor: Towards removing the curse of dimensionality. In *The 30th symposium on theory of computing (STOC)*, pp. 618–625.
18. Koga, H., Oguri, M., & Watanabe, T. (2012). Mixed-lsh: Reduction of remote accesses in distributed locality-sensitive hashing based on l1-distance. In *2012 IEEE 26th international conference on advanced information networking and applications*, pp. 175–182 . https://doi.org/10.1109/AINA.2012.29
19. Kulis, B.K.G. (2012). Kernelized locality-sensitive hashing. *IEEE Transactions on Pattern Analysis & Machine Intelligence, 34*(6), 1092–1104.
20. Li, P., Wang, M., Cheng, J., Xu, C., & Lu, H. (2013). Spectral hashing with semantically consistent graph for image indexing. *IEEE Transactions on Multimedia, 15*(1), 141–152. https://doi.org/10.1109/TMM.2012.2199970
21. Lu, Y., Ma, T., Zhong, S., Cao, J., Wang, X., & Abdullah, A. (2014). Improved locality-sensitive hashing method for the approximate nearest neighbor problem. *Chinese Physics B, 23*(8), 80.
22. Luo, X., Zhou, M., Li, S., Hu, L., & Shang, M. (2020). Non-negativity constrained missing data estimation for high-dimensional and sparse matrices from industrial applications. *IEEE Transactions on Cybernetics, 50*(5), 1844–1855.
23. Luo, X., Zhou, M., Li, S., & Shang, M. (2018). An inherently nonnegative latent factor model for high-dimensional and sparse matrices from industrial applications. *IEEE Transactions on Industrial Informatics, 14*(5), 2011–2022.
24. Muja, M., & Lowe, D. G. (2014). Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis & Machine Intelligence, 36*(11), 2227–2240.
25. Panigrahy, R. (2005). Entropy based nearest neighbor search in high dimensions. In *SODA 06: proceedings of the seventeenth annual acm-siam symposium on discrete algorithms*, pp. 1186–1195.
26. Qin, L., Josephson, W., Zhe, W., Charikar, M., Kai, L. (2007). Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *International conference on very large data bases*, pp. 950–961.
27. Shakhnarovich, G., Darrell, T., & Indyk, P. (2006). Locality-sensitive hashing using stable distributions, pp. 61–72.
28. Slaney, M., Lifshits, Y., & He, J. (2012). Optimal parameters for locality-sensitive hashing. *Proceedings of the IEEE, 100*(9), 2604–2623. https://doi.org/10.1109/JPROC.2012.2193849
29. Sproull, R. F. (1991). Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica, 6*(1–6), 579–589.
30. Wang, J., Kumar, S., & Chang, S. F. (2012). Semi-supervised hashing for large-scale search. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 34*(12), 2393–2406. https://doi.org/10.1109/TPAMI.2012.48
31. Wang, W., Zhang, H., Zhang, Z., Liu, L., & Shao, L. (2021). Sparse graph based self-supervised hashing for scalable image retrieval. *Information Sciences, 547*, 622–640.
32. Wang, X. (2011). A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. In *The 2011 international joint conference on neural networks*, pp. 1293–1299 . https://doi.org/10.1109/IJCNN.2011.6033373
33. Weber, R., & Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-

dimensional spaces. In *1998 the 24th international conference on very large data bases (VLDB)*, pp. 194–205.
34. Weiss, Y., Torralba, A., & Fergus, R. (2009). Spectral hashing. *Advances in Neural Information Processing Systems, 282*(3), 1753–1760.
35. Xue, Z., & Wang, H. (2021). Effective density-based clustering algorithms for incomplete data. *Big Data Mining and Analysis, 3*, 183–194.
36. Yang, L., Zhu, Q., Huang, J., & Cheng, D. (2017). Adaptive edited natural neighbor algorithm. *Neurocomputing, 230*, 427–433.
37. Zhu, P., Zhan, X., & Qiu, W. (2015). Efficient k-nearest neighbors search in high dimensions using mapreduce. In *2015 IEEE fifth international conference on big data and cloud computing*, pp. 23–30 . https://doi.org/10.1109/BDCloud.2015.51

Dongdong Cheng is an associate professor of College of Big Data and Intelligent Engineering in Yangtze Normal University and she is also a postdoctoral fellow at Chongqing University of Posts and Telecommunications. She got a bachelor's degree in computer science from Chongqing Normal University in 2013 and a Ph.D degree from Chongqing University in 2018. Her research interests are clustering analysis and data mining.

Jinlong Huang is an associate professor of College of Big Data and Intelligent Engineering in Yangtze Normal University. He got doctoral degree from Chongqing University in 2017. His research interests are outlier detection and clustering analysis.

Sulan Zhang is currently a professor in the College of Big Data and Intelligent Engineering of Yangtze Normal University, China. She received her B.S. degree in department of Computer Science and Technology from Southwest University, China, 2006 and the Master degree in computer software and theory in Chongqing University, China, 2009. She received the Ph.D. degree in Computer Science and Technology from Chongqing University, China, 2013. Her main research interests include data mining, data analysis, and computer modeling and simulation, and so on.

Quanwang Wu is currently an associate professor of Computer College in Chongqing University, Chongqing, China. He received his B.S., M.S., and Ph.D. degrees in computer science from Chongqing University in 2007, 2010, and 2013. He was a special researcher at the digital content and media sciences research division of the National Institute of Informatics (NII) in Tokyo, Japan from 2014 to 2015. His main research interests include service- oriented computing, data mining and computational intelligence.