Fast distributed kNN Graph construction using auto-tuned Locality Sensitive Hashing

CARLOS EIRAS-FRANCO* and DAVID MARTÍNEZ-REGO, Research Center on Information and Communication Technologies (CITIC) - Universidade da Coruña.
LESLIE KANTHAN, Department of Maths and Computer Science - University College London.
CÉSAR PIÑEIRO, Centro de Investigacion en Tecnoloxías da Informacion (CiTIUS) - Universidade de Santiago de Compostela
ANTONIO BAHAMONDE, Department of Computer Science - Universidad de Oviedo.
BERTHA GUIJARRO-BERDIÑAS and AMPARO ALONSO-BETANZOS, Research Center on Information and Communication Technologies (CITIC) - Universidade da Coruña.

The k nearest neighbors graph is a popular and powerful data structure that is used in various areas of Data Science, but the high computational cost of obtaining it hinders its use on large datasets. Approximate solutions have been described in the literature using diverse techniques, among which Locality Sensitive Hashing is a promising alternative that still has unsolved problems. We present Variable Resolution Locality Sensitive Hashing, an algorithm that addresses these problems to obtain an approximate k nearest neighbors graph at a significantly reduced computational cost. Its usability is greatly enhanced by its capacity to automatically find adequate hyperparameter values, a common hindrance to LSH-based methods. Moreover, we provide an implementation in the distributed computing framework Apache Spark that takes advantage of the structure of the algorithm to efficiently distribute the computational load across multiple machines, enabling practitioners to apply this solution to very large datasets. Experimental results show that our method offers significant improvements over the state-of-the-art in the field and shows very good scalability as more machines are added to the computation.

CCS Concepts: • **Computing methodologies → Machine learning algorithms**; **MapReduce algorithms**;

Additional Key Words and Phrases: Big Data, scalability, k nearest neighbors, Locality Sensitive Hashing, AutoML

ACM Reference format:
Carlos Eiras-Franco, David Martínez-Rego, Leslie Kanthan, César Piñeiro, Antonio Bahamonde, Bertha Guijarro-Berdiñas, and Amparo Alonso-Betanzos. 2019. Fast distributed kNN Graph construction using auto-tuned Locality Sensitive Hashing. *ACM Trans. Intell. Syst. Technol.* -, -, Article 1 (September 2019), 18 pages.
https://doi.org/-

1 INTRODUCTION

Data science has drastically gained popularity in the last few years thanks, in part, to the enormous amount of data that is available, what is commonly known as Big Data [28]. This phenomenon has

*Corresponding author.

2157-6904/2019/9-ART1 $15.00
https://doi.org/-

increased the reach of the results and predictions made by machine learning algorithms, but it also has forced researchers to devise new ways to process data at such scale. The ability to work, as the amount of data grows, is known as scalability and new algorithms and revisions of existing ones have been created having scalability as a primary goal.

A crucial strategy for dealing with Big Data is distributed computing. It consists of dividing the work at hand across multiple computational units, accelerating the response time and distributing the storage load. This approach gained significant traction with the introduction of the Map Reduce paradigm [16], an abstraction presented by Google in 2008 that facilitates the distribution of computations as long as they conform to two very general types of processing, namely, Map and Reduce operations. An open-source implementation of these idea was soon launched under the name Apache Hadoop [20]. Later on, more specialised frameworks were developed, among which Apache Spark [35], that was developed with the objective of maintaining reusable data in memory as long as possible and providing a flexible programmer API, is probably the most popular. The success of these frameworks and its suitability for data science led to the creation of powerful libraries such as Mahout [21] for Hadoop and MLLib [29] for Spark, that contain distributed implementations of Machine Learning (ML) algorithms.

A popular data structure that is widely used in Machine Learning but missing from these libraries is the k nearest neighbors graph (kNNG), which is a representation of all elements of a dataset $\mathcal{D}$ as a directed graph in which for n data points, $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_n\}$, edges $(\mathbf{x}_i, \mathbf{x}_j)$ indicate that $\mathbf{x}_j$ is amongst the k most similar elements to the point $\mathbf{x}_i$ under a specified similarity measure $\sigma(\mathbf{x}_i, \mathbf{x}_j)$. This data structure allows one to easily navigate elements that are similar to each other. This is useful in areas such as Data Mining [15], Computer Graphics [32] and Machine Learning, specifically outlier detection [26], feature selection [27] and classification [14]. Despite being a conceptually simply idea, the computational cost necessary to obtain the kNNG by brute force is high, since it requires performing $n(n-1)/2$ pairwise comparisons, which amounts to $O(n^2)$ time complexity. As a result, there have been attempts to obtain algorithms that compute this graph at a lower cost.

In this paper, we present a novel approach to compute an approximate version of the kNNG that is based on Locality Sensitive Hashing (LSH) [3] schemes. LSH is a technique designed to speed up the retrieval of points in a dataset that are similar to a query point by pre-building a data structure. The main idea behind LSH is that if two points are close in the original space, they will continue to be so after a projection, which is used to group points that are similar. LSH-based methods leverage this property to efficiently compute an approximate kNNG, but have many dataset-dependent hyperparameters that can greatly affect their performance. In general, Machine Learning application to real world problems has been a challenge, in part due to the need of tuning these hyperparameters in many ML algorithms. Automating this process offers the advantages of producing faster and simpler solutions for non-practitioners, as they do not need to be an expert to make use of ML models and techniques. Besides, requiring minimum human intervention, is a desirable feature and a competitive advantage over many of the state-of-the-art methods when learning from large-scale datasets as, in these cases, it may not be easy or even feasible to tune many hyperparameters due to the long computational times it would require. This work offers three main contributions:

(1) we present a LSH-based algorithm that can automatically tune the involved hyperparameters efficiently without user intervention.
(2) an iterative, multi-resolution focus is used to shed unnecessary computations, reducing computational complexity, and to ensure good accuracy of the approximate kNNG regardless of the dataset.

(3) the presented algorithm is structured to enable parallel computation, and we provide a distributed implementation in Apache Spark which can use a computer cluster to run this algorithm on very large datasets.

Our experimental results show that the proposed algorithm offers important improvements over the current state-of-the-art alternative algorithms that make it a ready solution for the problem at hand.

The rest of this paper is organized as follows: in Section 2 we discuss the state of the art in the field, in Section 3 we describe the presented algorithm, while in Section 4 we report the experiments performed to assess the validity of our proposal. Finally in Section 5 we summarize our conclusions and we reflect on which future developments of the algorithm could be made.

2 RELATED WORK

The great number of applications of the *k*NNG and the complexity of its calculation has motivated researchers to obtain efficient variations of the *k*NN algorithm. The literature reflects solutions that are computationally effective under certain conditions. When the dimensionality of the input space is small, the use of multidimensional binary search trees named k-d trees has been proven fast [8], but this solution rapidly becomes inefficient as the dimension of the input space grows (curse of dimensionality). An effective approach has also been proposed for when the similarity metric used is the cosine similarity [2], which first computes an approximation of the graph and then refines it by using the theoretic properties of this particular similarity measure.

However, so far the only way to cope with general metrics and high dimensional datasets at a reasonable computational cost is to build an approximate version of the *k*NNG, managing the tradeoff between the computational effort invested and the accuracy of the obtained graph with respect to its exact counterpart. Different approaches have been proposed using a number of techniques to reduce computational complexity. Some success was achieved by using divide-and-conquer based approaches that include the use of recursive inexpensive bisection steps [12, 33] that still amount to a high, although reduced with respect to the original, computational complexity.

2.1 Local-search methods

Local search approaches in this context take advantage of the fact that the neighbor of a neighbor is likely to also be a neighbor. Updating the nearest neighbors list for a point with the nearest of the neighbors of its neighbors is called a *neighbor descent* approach. The seminal work for this approach is NN-Descent [17], which starts with a random *k*NNG which is iteratively refined by performing neighbor descent steps until the graph converges to a state that is not significantly changed by performing additional iterations. The results of these local search methods suffer when the dataset has high intrinsic dimensionality, and are prone to converging to local optima. Moreover, they become computationally costly for large values of k. Several modifications of NN-Descent have been proposed to address these shortcomings [10, 22], but so far none of them has given a universal solution. Still, neighbor descent can be an effective tool to increase the accuracy of an approximate *k*NNG, and many algorithms resort to it by either providing a good initialization for the descent steps or by using a single descent step to refine an approximate *k*NNG calculated using other techniques. One recent such method is EFANNA [22], which applies a divide-and-conquer approach to derive binary trees that in turn are employed to obtain an approximate graph that is used as the starting point for NN-Descent.

Finally, the application of LSH enables a generic strategy for approximating the *k*NNG under any similarity measure [36]. Since this approach is the base of our proposal, we will analyze both its theoretical foundations and the existing methods that use it in the next subsection.

2.2 kNNG using Locality-Sensitive Hashing

The use of LSH for the construction of the kNN graph is based on its ability to group elements that are similar. In particular, the main idea underlying LSH is that if two points are similar, they will continue to be so after a projection. This idea is used to reduce the search space for a given query point, that is, given $\mathbf{x}$, when trying to retrieve its k nearest neighbors the use of LSH allows to search only points that are likely to be similar to $\mathbf{x}$ instead of the whole dataset $\mathcal{D}$. This is accomplished with a Locality-Sensitive Hash function, that is, a function that maps elements from a high-dimensional space, which is generally sparse, to a lower-dimensional more dense space and does so in a manner such that elements that are close in the input space are mapped to the same point of the image space with a high probability. A family of hash functions $\mathcal{H}$ is called $(r; cr; P_1; P_2)$-sensitive with respect to a given similarity measure σ if for any two points $\mathbf{p}, \mathbf{q} \in \mathfrak{R}^d$:

$$\sigma(\mathbf{p}, \mathbf{q}) \leq r \longrightarrow Pr(h(\mathbf{p}) = h(\mathbf{q})) \geq P_1 \tag{1}$$

$$\sigma(\mathbf{p}, \mathbf{q}) > cr \longrightarrow Pr(h(\mathbf{p}) = h(\mathbf{q})) \leq P_2 \tag{2}$$

with $h \in \mathcal{H}$. Specifically, given $\mathbf{p}, \mathbf{q} \in \mathfrak{R}^d$ if $\sigma(\mathbf{p}, \mathbf{q}) \leq r$, that is, the distance between the points is less than the sensitivity radius r, they will be considered similar and the random hash function h will produce a collision, that is, assign them the same value, with a probability at least P_1. Conversely, if $\sigma(\mathbf{p}, \mathbf{q}) > cr$, $\mathbf{p}$ and $\mathbf{q}$ will not be considered similar and the probability of h assigning them the same value will be lower than P_2. If $P_1 >> P_2$ then those points that are given the same hash value will be very likely to be similar in the input space. Moreover, if a point is given a hash value $h(\mathbf{x})$ then most elements similar to $\mathbf{x}$ will be given the same hash value. These two characteristics make LSH very useful for reducing the search space to elements that are similar to the query. In some cases P_1 is just slightly larger than P_2; a common approach to increase this difference consists in concatenating several hashing function values [3]. Additionally, in order to increase the number of collisions it is also a common practice to generate several hash keys for each point using various hashing functions from $\mathcal{H}$.

As mentioned above, this technique was originally used to perform similarity between queries in sublinear time [3, 13] by constructing a data structure that organizes the input data according to the values assigned by the LSH function. Specifically, a group of hashing functions is computed and the hash values of existing points in the dataset are stored. For a given query point, only those elements of the dataset that share the same hash value (i.e. very likely to be similar) are compared to it, greatly reducing the number of pairwise comparisons and, therefore, reducing the computational complexity. Despite this being the usual approach to leveraging LSH, there is still some degree of uncertainty given its dependence on probabilities P_1 and P_2. The data structures and the query methods used in LSH are an active area of research [13]. As a result, the optimal way of exploiting LSH still remains an open problem.

The described scheme is used to tackle two problems closely related to kNNG construction. The first one is named *nearest neighbor search* [33], which consists in retrieving the k nearest neighbors in a dataset $\mathcal{D}$ to a query point $\mathbf{p}$ not present in the dataset, and has been successfully used in fields such as search engines [23], image search [24], computational linguistics [31] and computational biology [11], working even on cross-modal data [25]. The second one is *spherical range reporting* [1, 30], that requires retrieving all points $\mathbf{x} \in \mathcal{D}$ such that $\sigma(\mathbf{x}, \mathbf{p}) < r$ for a given query point $\mathbf{p}$ not in $\mathcal{D}$ and a threshold value r. Still, computing the kNNG entails a different set of restrictions from the aforementioned problems. Mainly, the focus for approximate kNNG algorithms is obtaining a graph as accurate as possible in the least possible amount of time so that additional processing can be done using it as a starting point. The data structure built in the process is discarded, which is a contrast to *nearest neighbor search* and *spherical range reporting*,

in which besides accuracy, both the size of the resulting data structure and the speed of each query answer (i.e. the effectiveness of the data structure) need to be taken into account, but the time invested in computing the structure is not crucial. Therefore, it may be advisable in such problems to invest some more time in computing a finely tuned data structure. These differences make the adaptation of algorithms that solve *nearest neighbor search* or *spherical range reporting* to tackle kNNG construction non trivial. Up to the authors knowledge, so far only one work, by Zhong *et al.* [36], has used LSH to compute the kNNG. This algorithm first splits data into groups of similar elements using LSH, then computes the pairwise similarities of the elements in each of these groups, which are used to build a partial graph for each group. These partial graphs are finally merged, producing an approximate kNNG. This process is repeated for several iterations, merging the resulting graphs. Finally, the obtained graph is refined with a neighbor descent step. This algorithm has many dataset-dependent hyperparameters that need to be tuned, which complicates the obtention of good results, a common theme to LSH methods [18]. Additionally, datasets that have very uneven density of elements in different regions may obtain poor performance.

Our approach is based on the algorithm proposed by Zhong *et al.*, but it addresses the mentioned shortcomings: iterative processing at decreasing resolutions ensures sensibility to regional density changes and an automatic hyperparameter tuning procedure manages to obtain good results regardless of the characteristics of the dataset. Moreover, we structured the algorithm so that it has many parts that can be computed in parallel and can be implemented following the MapReduce paradigm. Also, we provide an implementation in the distributed computation framework Apache Spark, which can use a cluster of computers to perform the computation, amounting to a great scalability of the method.

3 IMPLEMENTING THE ALGORITHM

We present Variable Resolution LSH (VRLSH)[1], an algorithm that uses LSH repeatedly to explore groups of similar points that increase in size at each step. Additionally, the points that have been sufficiently explored are removed from the dataset at each step. This iterative approach is a major difference with the existing LSH based algorithm [36], and it enables the proposed algorithm to adapt its exploration to the density of each region of a dataset without affecting the overall computational cost.

VRLSH works as described in Algorithm 1. First, every element $\mathbf{x}$ of the dataset $\mathcal{D}$ is given a hash value $h(\mathbf{x})$ using a LS hashing function that will produce collisions for elements with a similarity value larger than a given resolution. After that, elements with the same value of $h(\mathbf{x})$ are grouped, forming buckets of points with a high probability of being similar. A kNN subgraph is computed for each of these buckets by computing all possible pairwise distances and linking each element in the bucket to its k nearest neighbors. Afterwards, overlapping subgraphs are merged, forming an approximate kNNG. At this step, all points that have already been involved in a fixed number of pair-wise computations (C_{MAX} in Algorithm 1) are removed from the dataset. The line of reasoning for this step is that, since all points compared to a given one, $\mathbf{x}$, are very likely to be similar to it, thanks to the LSH filtering performed, once a point has been involved in a large number of such comparisons, it will be very probable that all of its k nearest neighbors will have already been compared to it. Moreover, all points for which $\mathbf{x}$ is one of their k nearest neighbors will be very likely to have also been involved in those pairwise comparisons and one can, therefore, remove said point $\mathbf{x}$ from the dataset. Finally, the resolution is decreased so that in the following iteration the similarity threshold is lower, that is, points that were not considered to be similar in the current step because they are too different may be considered to be similar with a lower

[1]Spark implementation available for download at https://github.com/eirasf/KNiNe

resolution. The process is then repeated on until the simplified dataset $\mathcal{D}'$ is empty or has very few elements or all of its elements end in the same bucket. After that, the elements that ended up in the graph with less than k neighbors are returned to the dataset. This can occur when an element is removed from the dataset for having been involved in more than C_{MAX} comparisons; C_{MAX} is always selected to be larger than k, but since the elements involved in the comparisons can not be recorded, some comparisons can be repeated, and, in rare cases, this can amount to a number of relevant comparisons lesser than k. A closing step is performed in the algorithm if needed, for the rare cases when very few points are left in the simplified dataset. If that is the case, instead of continuing the hashing process which would, presumably, yield few meaningful collisions, it is preferable to compare these points to the neighbors of its neighbors in the partial approximated graph, that is, perform a local search using neighbor descent, step that is described from line 10 on. Finally, the full approximate graph is refined in all cases with a neighbor descent step, which, as stated in Section 2, is a common approach to increase the accuracy of the approximate graph.

Managing the resolution of the similarity function as described allows the algorithm to process mostly small buckets of elements that are very likely to be close, avoiding performing numerous unnecessary pairwise comparisons. The mentioned dataset simplification step manages to keep the number of elements in each bucket small when the resolution is decreased. Using these two innovations, VRLSH manages to compare each point $\mathbf{x}$ to points that are very likely to be near neighbors, which works towards the accuracy of the approximated kNNG, while maintaining the number of pairwise comparisons low, which leads to low computational cost.

The resulting algorithm is a good fit for parallel computation, which transforms it in a very scalable solution. The hashing step can be performed in parallel across several computing units, then the data can be distributed so that each computing node calculates the subgraph for a subset of the resulting buckets. This parallel processing speeds up the computation substantially. The addition of a registry that records the pairwise distances that have already been computed would allow the avoidance of duplicate calculations, but it would also impact the memory usage and, more importantly, it would diminish the suitability for distributed computation, so we opted not to include it.

3.1 Hyper-parameter tuning

As mentioned in Section 2, state-of-the-art LSH methods are hindered by the number of hyper-parameters that need to be tuned for the LSH scheme to be efficient. The optimal value for these hyperparameters varies with the dataset, which further complicates its calculation. This constitutes a problem for all LSH algorithms. Although there has been some work aimed at tuning the hyperparameters in the particular case of *nearest neighbor search* problems [7, 18], the current research in the field offers no general solution for this problem. The process of hyperparameter tuning is even more important in the case of kNNG construction since, as stated in Section 2, it is a one-shot algorithm that attempts to speed up a computationally costly process and any time devoted to hyperparameter tuning decreases the temporal efficiency of the method, making the algorithm less valuable. This is a contrast to LSH algorithms tackling *nearest neighbor search* for which the main goal is speed and accuracy at query time and, consequently, those algorithms can spend more time in hyperparameter tuning. We present a fast hyperparameter tuning process that performs a guided search of the hyperparameter space until finding a suitable set of values. In the next subsections we describe how each hyperparameter is managed and we detail the complete process.

3.1.1 Resolution. Although in many cases setting an initial resolution R of 0.1 is a valid value that will trigger the creation of aptly-sized buckets [19], this may not be the case for some datasets,

Algorithm 1: Pseudo-code for VRLSH algorithm.

Input: $\mathcal{D}$,k ←Set of points, Number of neighbors to be obtained
Input: R_0 ← Initial resolution
Input: C_{MAX} ←Max number of comparisons per element
Output: G ← Graph containing the k nearest neighbors for each point

$G \leftarrow \emptyset$, $\mathcal{D}' \leftarrow \mathcal{D}$, $R \leftarrow R_0$
while $|\mathcal{D}'| > k$ ***and*** $|buckets| > 1$ **do**
  $hashElems \leftarrow LShash(\mathcal{D}', R)$
  $buckets \leftarrow hashElems.groupByHash()$
  foreach b ***in*** $buckets$ **do**
    if $(b.size > 1)$ **then** $G \leftarrow G \cup exactKNN(b.elems, k)$ **end**
  end
  $\mathcal{D}' \leftarrow \mathcal{D}' - G.getNodesWithAtLeastComparisons(C_{MAX})$
  $decrease\ R$
end
$\mathcal{D}' \leftarrow \mathcal{D}' \cup G.getNodesWithFewerNeighborsThan(k)$
if $|\mathcal{D}'| > 1$ **then**
  foreach p ***in*** $\mathcal{D}'$ **do**
    if $|p.neighbors| = 0$ **then**
      $p.neighbors \leftarrow randomSample(\mathcal{D}, k)$
    end
    else
      $p.neighbors \leftarrow topK(k, p.neighbors \cup neighborDescent(p, G))$
    end
  end
end
$G \leftarrow neighborDescent(G)$

which may end up creating buckets with too many (or too few) elements, which would amount to a great number of unnecessary pairwise comparisons (or unnecessary iterations of the algorithm), resulting in extra computational cost. To address this problem, we added a quick estimation procedure that, given a desired initial bucket size, obtains a suitable R_0 value. First, with R_0 set to 0.1, the whole dataset is hashed and the size of the resulting buckets is checked. If they contain too few or too many elements, the resolution is halved or doubled, respectively, and the process is repeated. If two R_0 values are found to be one too small and the other too large then a binary search is performed. This process is stopped as soon as a suitable R_0 value is found. Although this procedure may require a sizeable number of hashing and grouping steps, it can be performed rapidly since these operations are carried out in parallel across the computing nodes. The resulting execution time of this procedure is very small, compared to the total execution time of the kNNG computation, and the impact of using a R_0 of the correct size in the total time of the algorithm can be considerable. Therefore the use of this tuning procedure is very advisable.

3.1.2 Hyperparameters for Euclidean distance as a similarity measure. Also, in the particular case of using the Euclidean distance as a similarity measure, the family of locality sensitive hashing functions that is normally used is based on performing random projections of the datapoints. In this case, hash keys are vectors calculated using Equation 3 where R is the resolution and M_i

is a projection matrix that is obtained as explained hereafter. For a given sample $\mathbf{x} \in \Re^d$ each component c of the key is calculated as the integer part of the dot product $\mathbf{x} \cdot \mathbf{w_c} + b_c$ where $\mathbf{w_c}$ is a vector with d components randomly sampled from a $N(0,1)$ and b_c is a scalar bias sampled in the same way. For ease of notation, the corresponding α $\mathbf{w}$ vectors and α biases that determine a hash are joint into a matrix $\underset{(d+1)\times\alpha}{\mathrm{M}} | m_{i,j} \sim \mathcal{N}(0,1)$. Equation 3 can be interpreted as projecting the samples onto a random hyperplane and segmenting the projected vectors according to their length.

$$\mathbf{hash_i}(\mathbf{x}) = floor\left((\mathbf{x}, 1) \cdot M_i \cdot R\right) \tag{3}$$

A fixed number β of such hashes are calculated for each element, as described in Equation 4, to ensure that there are enough meaningful collisions.

$$Keys(\mathbf{x}) = \left\{\mathbf{hash_0}(\mathbf{x}), \mathbf{hash_1}(\mathbf{x}), \ldots, \mathbf{hash}_\beta(\mathbf{x})\right\} \tag{4}$$

This formulation introduces two additional hyperparameters: α (or key length), representing the length of the hashes, and β (or number of tables) which accounts for the number of hashes generated per element. The effect of these hyperparameters in the performance of the algorithm can be characterized as follows: *alpha* affects the size of the buckets since it dictates how many projections determine a hash. A large α will produce hashes that are very specific and, therefore, generate fewer collisions than a small α, although the elements assigned to the same bucket will have a higher probability of being similar for larger values of α. We would, then, prefer to use the largest value of α that produces a suitable number of collisions. The effect of β is increasing the number of collisions by assigning several hashes to each element.

In order to tune these hyperparameters with the initial resolution, we use a procedure described in Algorithm 2, that extends the above-mentioned. We empirically discovered that setting the β hyperparameter to a fixed constant value and then tuning α and R was the more suitable choice. When providing a value for β we should take into account the fact that high dimensional spaces are more sparse than low dimensional spaces and, consequently, a higher β is needed in order to produce enough collisions as the dimension of the space grows. Therefore, we opted for a simple logarithmic formula depending on the input dimension d and set $\beta = \left(\log_2 d\right)^2$. Additionally, a suitable value for the α hyperparameter is estimated as $\alpha_0 = ceil\left(\log_2\left(\frac{n}{d}\right)\right) + 1$, formula inspired by the work of Zhong *et al.* [36]. Then a binary search for a suitable α is performed in the range $[\alpha_0/2, \alpha_0 * 1.5]$, selected to tolerate some variation in the found α while maintaining it close to α_0. This search corresponds to the loop on line 4. To conduct this search, R is set to $R = 0.1$ and the hashing and counting procedures described for the resolution hyperparameter tuning are performed. If any α value in that range produces buckets of the desired size, then all three hyperparameters have been set. Otherwise, a suitable R is searched using the procedure described at the beginning of this section (which is represented in the pseudocode by the function *findResolution*) and once that value is set, the search for α in the aforementioned range is repeated. This procedure yields a combination of the three hyperparameters that configures LSH to produce buckets of the desired size, and does so without having much impact in the execution time of the method, since the operations involved are much less costly than the numerous pairwise distance measurements involved in the iterations of the main algorithm.

3.1.3 C_{MAX} and desiredSize. Finally, the algorithm has another hyperparameter named C_{MAX} that represents the number of comparisons in which an element of the dataset should be involved for it to be removed from the dataset in a simplification step. This ensures that every element in the final graph will be compared to, at least, C_{MAX} other elements. The closely related *desiredSize*

Algorithm 2: Pseudo-code for the hyperparameter tuning procedure.

Input: $\mathcal{D}$,k ←Set of points, Number of neighbors to be obtained
Input: $desiredSize$ ←Desired bucket size
Output: R_0 ← Initial resolution
Output: α, β ←Euclidean distance LSH hyperparameters.

$R \leftarrow 0.1$, $\beta \leftarrow (\log_2(\mathcal{D}.dimension))^2$
$minS \leftarrow desiredSize * 0.5$, $maxS \leftarrow desiredSize * 1.5$
$\alpha_0 \leftarrow ceil(log_2(|D|/D.dim)) + 1$, $left\alpha \leftarrow \alpha_0 * 0.5$, $right\alpha \leftarrow \alpha_0 * 1.5$
while *True* **do**
  $current\alpha \leftarrow (left\alpha + right\alpha)/2$
  $hashElems \leftarrow EucLSHash(\mathcal{D}, R, \alpha, \beta)$
  $sizes \leftarrow hashElems.countByHash()$
  if $sizes.max \in [minS, maxS]$ **then**
    return $R, current\alpha, \beta$
  end
  else
    if $sizes.max < minS$ **then**
      $right\alpha \leftarrow current\alpha$
    end
    else
      $left\alpha \leftarrow current\alpha$
    end
    if $left\alpha >= right\alpha$ **then**
      $R \leftarrow findResolution(current\alpha, \beta, minS, maxS)$ $left\alpha \leftarrow \alpha_0 * 0.5$,
       $right\alpha \leftarrow \alpha_0 * 1.5$
    end
  end
end

hyperparameter indicates how large the buckets generated by the LSH procedure should be. In Section 4 we detail the experiments performed in order to determine how to handle these two hyperparameters.

4 EXPERIMENTAL DESIGN AND RESULTS

In order to verify the validity of our approach we performed various sets of experiments on three real-world datasets, listed in Table 1. These datasets, representing audio signals, 3D shapes and images, respectively, were selected because they were employed by other authors in previous works to benchmark the approximate kNNG-building algorithm NN-Descent [17] and an LSH approach to the *nearest neighbor search* problem [18].

We used three performance measures in our experiments. The first one is related to the *accuracy* of the computed graph for which we employed the *recall* measure, defined as the ratio of common edges between the approximate and the exact graphs with respect to the total number of edges. This metric is the most usual when assessing the quality of the retrieved k nearest neighbors [4]. Secondly, we gauged the performance of the algorithm by counting the *number of pairwise computations* performed and dividing that number by the number of pairwise computations that

Dataset	Size	Dimensionality
Audio	54387	192
Shape	28775	544
Corel	662317	14

Table 1. Datasets used in the study.

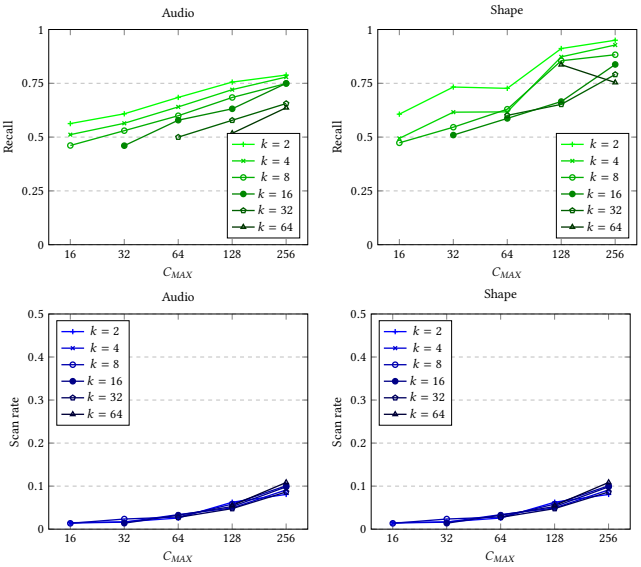

Fig. 1. Recall vs C_{MAX} for *Audio* and *Shape* datasets and Scan rate vs C_{MAX} for those same datasets, using $desiredSize = 4 \cdot C_{MAX}$ in both cases.

the naïve algorithm would use, which is $n(n-1)/2$ where n is the number of elements of the dataset. This metric, known as *scan rate*, is also most commonly used in the literature. Finally, to measure more precisely the quality of the approximate graphs by making a difference between graphs containing the same number of mistakes, we added an additional measure that quantifies those mistakes: the mean error (ME) in the distance of the retrieved neighbors, defined as

$$ME = \frac{\sum_{i=0}^{n} \sum_{j=0}^{k} \sigma(p_i, n(p_i)_j) - \sigma(p_i, n^*(p_i)_j)}{n \cdot k} \tag{5}$$

where $n(p)_k$ represents the k-th neighbor of p in the approximate graph and $n^*(p)_k$ represents the k-th neighbor of p in the exact graph.

4.1 Handling C_{MAX} and $desiredSize$

As mentioned in Subsection 3.1, C_{MAX} establishes a threshold to the number of comparisons per element. Once an element is compared to candidate neighbors more than C_{MAX} times, it will be removed from the dataset, working on the assumption that it has been compared to enough elements as to have a high probability of having encountered its k nearest neighbors. To observe the effect of C_{MAX} in the obtained recall and scan rate we ran the algorithm using different values of C_{MAX} for the *Audio* and *Shape* datasets. The results of these experiments are showed in Figure 1.

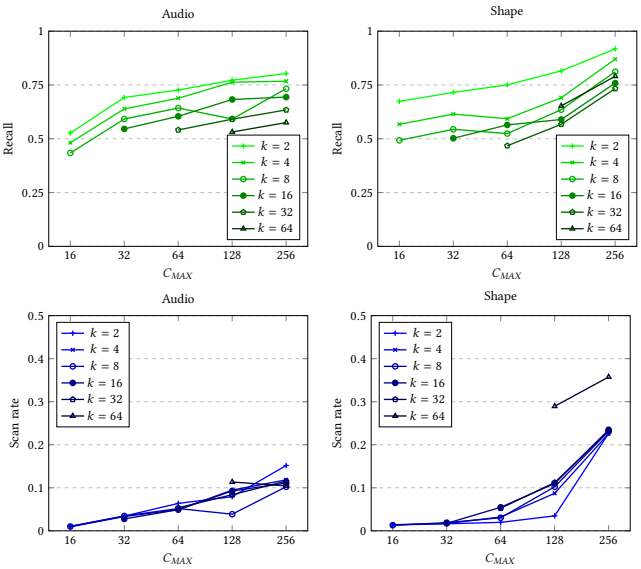

Fig. 2. Recall vs C_{MAX} for *Audio* and *Shape* datasets and Scan rate vs C_{MAX} for those same datasets, using $desiredSize = 0.8 \cdot C_{MAX}$ in both cases.

The recall of the obtained graphs has a positive dependence on C_{MAX}. As C_{MAX} grows, the recall grows linearly in both datasets. This is consistent with the expected effect: the larger C_{MAX} is, the more accurate the resulting graph will be, since there are more possibilities of finding the k nearest neighbors in a larger set of elements, but also the costlier the computation will be, since a larger number of pairwise comparisons will be performed. This can be appreciated in the plots that represent the Scan rate vs C_{MAX}. Moreover, this dependency is superlinear, that is, the scan rate grows at an increasing and faster rate than C_{MAX}. It can be seen, in consequence, that this parameter manages the balance between accuracy and computational cost that is intrinsic to this problem. It is important to note that since C_{MAX} has a stronger effect on the scan rate than on the recall of the graph, it is not advisable to use large values for C_{MAX} since the computational cost would become too large. We decided to allow the user to modify this hyperparameter to manage the balance between precision and speed of computation, but we have, nonetheless, provided a default value $C_{MAX} = 10 \cdot k$ (truncated to a max of 250 except for $k > 225$ in which case it is $C_{MAX} = 1.1 \cdot k$) which we empirically found to offer a suitable balance.

On the other hand, the *desiredSize* hyperparameter, which is highly related to C_{MAX}, indicates how large the buckets created in the LSH steps should be. Its relation with C_{MAX} determines how many hashing steps will most elements in the dataset endure. If *desiredSize* is much smaller than C_{MAX}, elements will need to be hashed several times until they reach the necessary number of comparisons. Conversely, if *desiredSize* is larger than C_{MAX}, many elements will undergo a single hashing and grouping step. To analyze this behaviour we ran the mentioned experiment with two values for *desiredSize*, representing two different configurations: $desiredSize = 4 \cdot C_{MAX}$, which should force many elements to be discarded for having enough comparisons after a single hashing step, depicted in Figure 1 and $desiredSize = 0.8 \cdot C_{MAX}$, shown in Figure 2, which should keep elements for a longer number of iterations of the hashing step before removing them in a simplifying step.

Method	Approach	Hyperparameters	Values used
VRLSH	LSH-based	α, β, R	Automatically tuned
FastKNN	LSH-based	α, β, $blockSz$, $iterations$	20, 1, 100, 20
NN-Descent	Local search	$Samplerate$	1
EFANNA	Divide-&-conquer + Local search	$\#trees$, $depth$, $iterations$, L, $check$, S	8, 8, 8, 30, 25, 10

Table 2. Methods compared in the study.

These experiments show that using $desiredSize = 4 \cdot C_{MAX}$ offers predictable results in terms of scan rate, while the computational effort required for the computation becomes much more variable when $desiredSize = 0.8 \cdot C_{MAX}$. Moreover, scan rates are slightly higher when $desiredSize = 0.8 \cdot C_{MAX}$, but contrary to the expected behaviour, this increase in computational effort does not revert in higher recall values; on the contrary, the recall values for the graphs obtained are slightly lower than those obtained when $desiredSize = 4 \cdot C_{MAX}$. These results can be explained because the increased number of iterations required for each element when $desiredSize = 0.8 \cdot C_{MAX}$ results in more comparisons $\sigma(\mathbf{p}, \mathbf{q})$ being repeated for the same values of $\mathbf{p}$ and $\mathbf{q}$, as described in Section 3, resulting in turn in more elements accumulating purposeless repeated computations that count towards the C_{MAX} threshold and amount to more elements being left without k neighbors after the LSH loop. These points need to be added for completion in the final steps of the algorithm, in the process described from line 9 on in Algorithm 1. Since these steps are more costly and do not benefit from the locality-sensitive reduced search space generated with the LSH steps, the scan rate increases without a significant improvement of the recall. Therefore, we decided to set $desiredSize = 4 \cdot C_{MAX}$, to ensure the predictability of the results and optimize the use of the LSH steps.

4.2 Performance of the method

In order to establish the fitness of the proposed method compared to the current state of the art for this problem, we performed another set of experiments in which the results obtained by VRLSH were compared to those of other methods capable of computing an approximate kNNG on high dimensional datasets using generic distance metrics, in particular using the Euclidean distance. The following methods were selected:

- NN-Descent [17][2], which is the most representative of the local-search methods and also has the advantage of having only one hyperparameter.
- FastKNN [36], which is the only previously available LSH-based method and, therefore, the most similar algorithm to VRLSH in the literature.
- EFANNA [22][3], which is an approximate nearest neighbor search algorithm that contains a module to compute an approximate kNNG.

A summary of the characteristics of these methods is listed on Table 2. For each of them we selected the hyperparameter values recommended by their authors.

The measurements of the computational cost shown in Table 3 show a clear disparity between NN-Descent and the rest of the methods. VRLSH, FastKNN and EFANNA[4] require a scan rate that remains nearly constant regardless of the number of neighbors. This is in contrast with NN-Descent, which requires a scan rate that grows very fast as the number of neighbors is incremented,

[2]Implementation available at https://code.google.com/archive/p/nndes/

[3]Implementation available at https://github.com/ZJULearning/efanna

[4]The available EFANNA implementation failed to complete the process for Corel with $k = 64$

Data	Method	k					
		2	4	8	16	32	64
Audio	VRLSH	0.031	0.024	0.027	0.034	0.062	0.177
	NNDES	0.001	0.007	0.022	0.067	0.214	0.762
	FastKNN	0.037	0.037	0.038	0.0415	0.052	0.083
	EFANNA	0.050	0.050	0.050	0.050	0.050	-[4]
Shape	VRLSH	0.029	0.030	0.030	0.045	0.115	0.306
	NNDES	0.002	0.014	0.039	0.120	0.382	1.471
	FastKNN	0.069	0.069	0.071	0.075	0.088	0.119
	EFANNA	0.065	0.065	0.065	0.065	0.065	-[4]
Corel	VRLSH	0.003	0.003	0.003	0.003	0.006	0.015
	NNDES	0.000	0.001	0.002	0.007	0.023	0.077
	FastKNN	0.003	0.003	0.003	0.003	0.004	0.006
	EFANNA	0.003	0.003	0.003	0.003	0.003	-[4]

Table 3. Scan rate required by the compared methods while calculating the kNNG with different values of k on the studied datasets.

making its use unadvisable for large values of k. This constitutes an important disadvantage for NN-Descent when $k > 16$. Moreover, its recall (shown in Table 4) only becomes competitive when $k >= 8$. Both shortcommings restrict the use of this algorithm to very specific values of k. Among VRLSH, FastKNN and EFANNA, FastKNN shows inferior performance in terms of recall, especially for small values of k. Finally, while EFANNA displays spectacular performance for *Audio* and *Shape* datasets, its very low recall on *Corel* highlights its greatest hindrance, which is the possibility of converging to local optima. Moreover, since EFANNA is designed for the *nearest neighbor search* problem (which, as stated in Section 2.2, demands more careful model fitting than approximate kNNG computation), it has many hyperparameters that are hard to tune, further complicating its use for kNNG construction. In contrast, VRLSH shows consistent high accuracy and small scan rate regardless of k for all datasets, while also presenting the important advantage of having no hyperparameters to tune by the user. Moreover, the values of the error shown in Table 5 indicate that the inaccuracies contained in the approximate kNNG computed by VRLSH are small, with exact neighbors being replaced only by points that are close nearby.

Figure 3 enables the comparison of all methods. It is worth noting that the scan rate is a measure relative to the square of the number of elements in the dataset, so a percent point in scan rate represents an amount of computation that depends on the dataset size. This means that for large datasets, a difference of a single percent point can represent significant time, while for small datasets the scan rate can approach 1, even for approximate methods. This effect can be noticed on the scan rates measured, which are significantly larger in the case of the smallest dataset (*Shape*) compared to the largest dataset (*Corel*). Moreover, in small datasets the scan rate of an approximate method can become larger than 1, which implies that it would be advisable to use the naïve method of computing the exact graph. For such small datasets we recommend calculating the exact graph and we provided a multithreaded implementation of the naïve method in our code. Conversely, for the large datasets that our proposal is intended for, the advantage in terms of scan rate that our method offers becomes very significant, since it represents a great amount of calculations.

Data	Method	k					
		2	4	8	16	32	64
Audio	VRLSH	0.848	0.821	0.847	0.883	0.924	0.960
	NNDES	0.002	0.429	0.892	0.982	0.998	1.000
	FastKNN	0.698	0.717	0.778	0.849	0.915	0.959
	EFANNA	0.997	0.996	0.995	0.993	0.962	-[4]
Shape	VRLSH	0.908	0.886	0.894	0.905	0.954	0.935
	NNDES	0.003	0.641	0.958	0.994	0.998	0.999
	FastKNN	0.796	0.821	0.865	0.909	0.954	0.976
	EFANNA	0.993	0.994	0.995	0.994	0.961	-[4]
Corel	VRLSH	0.896	0.932	0.922	0.946	0.972	0.989
	NNDES	0.000	0.419	0.950	0.996	0.999	0.999
	FastKNN	0.545	0.574	0.648	0.739	0.813	0.859
	EFANNA	0.078	0.098	0.121	0.148	0.174	-[4]

Table 4. Recall of the approximate kNNG calculated by the compared methods with different values of k on the studied datasets.

Data	Method	k					
		2	4	8	16	32	64
Audio	VRLSH	0.012	0.020	0.017	0.013	0.009	0.004
	NNDES	0.852	0.109	0.009	0.001	0.000	0.000
	FastKNN	0.204	0.194	0.179	0.161	0.140	0.118
	EFANNA	0.000	0.000	0.000	0.000	0.176	-[4]
Shape	VRLSH	0.001	0.001	0.000	0.000	0.000	0.000
	NNDES	0.102	0.007	0.000	0.000	0.000	0.000
	FastKNN	0.104	0.118	0.130	0.141	0.151	0.161
	EFANNA	0.000	0.000	0.000	0.000	0.008	-[4]
Corel	VRLSH	0.001	0.001	0.001	0.001	0.000	0.000
	NNDES	0.892	0.022	0.000	0.000	0.000	0.000
	FastKNN	0.190	0.197	0.203	0.208	0.214	0.221
	EFANNA	0.222	0.218	0.212	0.206	0.384	-[4]

Table 5. Mean distance error of the approximate kNNG calculated by the compared methods with different values of k on the studied datasets.

4.3 Scalability of the method

Finally, to measure the scalability of the method and the provided distributed implementation in Apache Spark, we performed the same computation by varying the number of computing nodes. These experiments were run in a computer cluster formed by 8 machines with 12 computing cores each. The technical specifications of each node are listed on Table 6. The Spark version used was 2.4.0, on Hadoop 3.0.0-cdh6.1.0. The operating system of the machines was CentOS Linux release 7.4.1708.

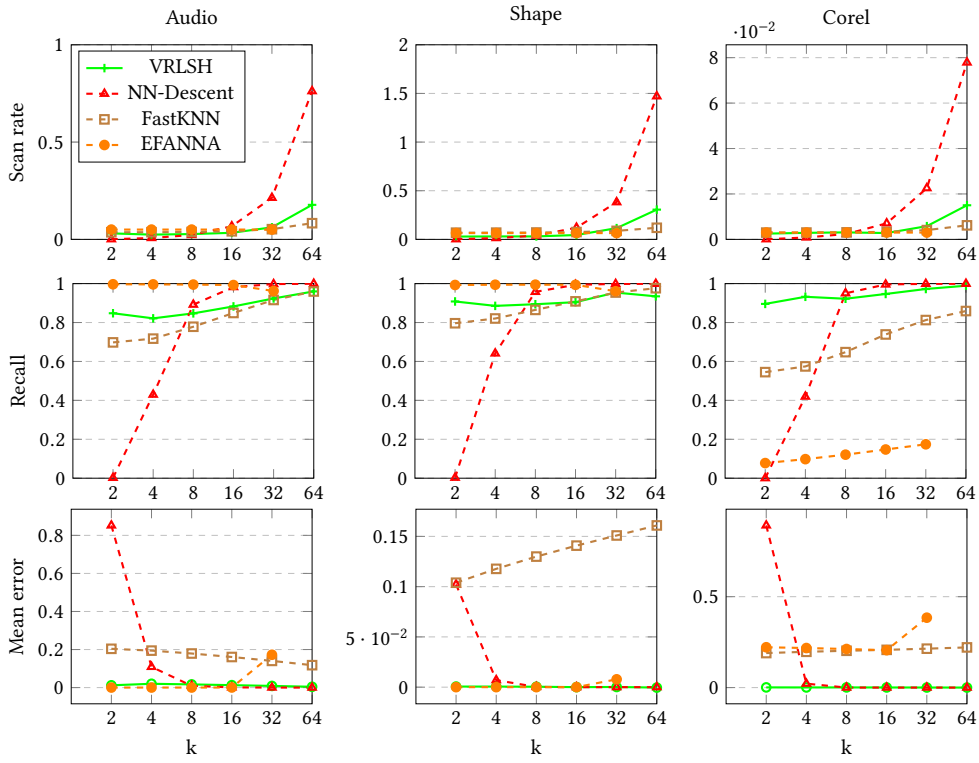

Fig. 3. Scan rate / Recall / Mean error vs number of neighbors plots for *Audio*, *Shape* and *Corel* datasets.

Table 6. Computer cluster overview:

8 nodes with the following characteristics:	
Processor:	2 × Intel Xeon E5-2620 v3 at 2.40Ghz
Cores:	6 per processor (12 per node)
Threads:	2 per core (24 total per node)
Storage:	12 × 2TB NL SATA 6Gbps 3.5" G2HS
RAM:	64 GB
Network:	1x10Gbps + 2x1Gbps

To ascertain the suitability of the method for processing large datasets, we used a dataset with more examples for this experiment. In particular, we selected the Higgs dataset[5] [5], which describes measurements of particle collisions and consists of 11 million examples with 28 attributes. With such a large number of elements, calculating the exact kNNG is completely out of reach, and calculating an approximation is the only option. To measure the scalability of our algorithm, we calculated the approximate 4NNG for the Higgs dataset several times using a growing number of computing nodes, and recorded the execution time invested in the calculation. For all these experiments we used $C_{MAX} = 32$. The results, listed on Table 7, show that the distributed implementation provided manages to harness the computational power of the available machines, obtaining almost

[5]Available for download at https://archive.ics.uci.edu/ml/datasets/HIGGS

	VRLSH			
# units	Time (s)	Scan rate	Ops/s	Speed-up
1	7390	$1.98 \cdot 10^{-4}$	$1.62 \cdot 10^{6}$	1.00
2	3972	$1.93 \cdot 10^{-4}$	$2.94 \cdot 10^{6}$	1.82
4	2045	$2.09 \cdot 10^{-4}$	$6.17 \cdot 10^{6}$	3.81

Table 7. Scalability vs number of computational units for the computation of the approximate 4NNG for the Higgs dataset. The speed-up listed is the ratio between the operations per minute obtained and the operations per minute performed with a single computational unit (12 cores).

linear speed-up, that is, accelerating the execution in proportion to the number of cores available. This feature enables the user to analyze very large datasets in a reasonable time as long as enough computational units are available.

5 CONCLUSIONS AND FUTURE WORK

In this paper we present VRLSH, a *k*NNG approximation algorithm which produces high recall graphs using a low scan rate irrespective of the number of neighbors selected, thus obtaining a good approximation of the exact graph at a much lower computational cost. The *k*NNG can be obtained without adjusting any hyperparameters, thanks to the automatic tuning procedure built into the method. This automatic feature can handle the hyperparameters of the algorithm and, additionally, those required by the Euclidean distance similarity measure, which is very frequently used. This solves the problem of hyperparameter tuning common to other LSH-based solutions. Additionally, we provide a distributed implementation of this algorithm in Apache Spark (available for download at https://github.com/eirasf/KNiNe), which exploits the structure of the algorithm to provide a distributed solution that can handle datasets with a large number of elements by using several computational units. This enables practitioners to tackle large datasets that are out of reach for other state-of-the-art methods. Our experimentation shows that our proposed method offers significant advantages over the previously available alternatives for *k*NNG computation.

In the future we will explore the possibility of using memory-efficient registers such as Bloom filters [9] to keep track of the pairwise computations that have been performed, thus helping to avoid the repetition of computations. Adapting this algorithm to similar problems such as *nearest neighbor search* and *spherical range reporting* is also a research avenue of great interest. Lastly, advances in automated parameter tuning over a pareto frontier by implementing multi objective genetic algorithms [6, 34] can be used to further optimize the parameters of the proposed algorithms to improve both accuracy and speed.

ACKNOWLEDGMENTS

This research has been supported in part by the Spanish Ministerio de Economía y Competitividad (project TIN 2015-65069-C2-1-R and 2-R), partially funded by FEDER funds of the EU and by the Xunta de Galicia (projects ED431C 2018/34 and Centro singular de investigación de Galicia, accreditation 2016-2019).

REFERENCES

[1] Thomas D Ahle, Martin Aumüller, and Rasmus Pagh. 2017. Parameter-free locality sensitive hashing for spherical range reporting. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 239–256.

[2] David C Anastasiu and George Karypis. 2015. L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM,

791–800.

[3] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 459–468.

[4] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2017. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International Conference on Similarity Search and Applications*. Springer, 34–49.

[5] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature communications* 5 (2014), 4308.

[6] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T Barr. 2018. Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 118–128.

[7] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*. ACM, 651–660.

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

[9] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[10] Brankica Bratić, Michael E Houle, Vladimir Kurbalija, Vincent Oria, and Miloš Radovanović. 2018. NN-Descent on High-Dimensional Data. In *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*. ACM, 20.

[11] Jeremy Buhler. 2001. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 17, 5 (2001), 419–428.

[12] Jie Chen, Haw-ren Fang, and Yousef Saad. 2009. Fast approximate kNN graph construction for high dimensional data via recursive Lanczos bisection. *Journal of Machine Learning Research* 10, Sep (2009), 1989–2012.

[13] Graham Cormode, Anirban Dasgupta, Amit Goyal, and Chi Hoon Lee. 2018. An evaluation of multi-probe locality sensitive hashing for computing similarities over web-scale query logs. *PloS one* 13, 1 (2018), e0191175.

[14] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.

[15] Belur V Dasarathy. 2002. Data mining tasks and methods: Classification: nearest-neighbor approaches. In *Handbook of data mining and knowledge discovery*. Oxford University Press, Inc., 288–298.

[16] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[17] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*. ACM, 577–586.

[18] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. 2008. Modeling LSH for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM, 669–678.

[19] Carlos Eiras-Franco, Leslie Kanthan, Amparo Alonso-Betanzos, and David Martınez-Rego. 2017. Scalable approximate k-NN Graph construction based on Locality Sensitive Hashing. In *ESANN 2017 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*.

[20] Apache Foundation. 2019. Apache Hadoop Project. http://hadoop.apache.org/. (2019). Accessed: 2019-09-01.

[21] Apache Foundation. 2019. Apache Mahout Project. http://mahout.apache.org/. (2019). Accessed: 2019-09-01.

[22] Cong Fu and Deng Cai. 2016. Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv preprint arXiv:1609.07228* (2016).

[23] Taher Haveliwala, Aristides Gionis, and Piotr Indyk. 2000. Scalable techniques for clustering the web. (2000).

[24] Shiyuan He, Bokun Wang, Zheng Wang, Yang Yang, Fumin Shen, Zi Huang, and Heng Tao Shen. 2019. Bidirectional Discrete Matrix Factorization Hashing for Image Search. *IEEE transactions on cybernetics* (2019).

[25] Mengqiu Hu, Yang Yang, Fumin Shen, Ning Xie, Richang Hong, and Heng Tao Shen. 2018. Collective reconstructive embeddings for cross-modal hashing. *IEEE Transactions on Image Processing* 28, 6 (2018), 2770–2784.

[26] K Ismo et al. 2004. Outlier detection using k-nearest neighbour graph. In *null*. IEEE, 430–433.

[27] Igor Kononenko. 1994. Estimating attributes: analysis and extensions of RELIEF. In *European conference on machine learning*. Springer, 171–182.

[28] Viktor Mayer-Schönberger and Kenneth Cukier. 2013. *Big data: A revolution that will transform how we live, work, and think*. Houghton Mifflin Harcourt.

[29] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[30] Ninh Pham. 2016. Hybrid LSH: Faster near neighbors reporting in high-dimensional space. *arXiv preprint arXiv:1607.06179* (2016).

[31] Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. 2005. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *Proceedings of the 43rd annual meeting on association for computational linguistics*. Association for Computational Linguistics, 622–629.

[32] Jagan Sankaranarayanan, Hanan Samet, and Amitabh Varshney. 2007. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics* 31, 2 (2007), 157–174.

[33] Jing Wang, Jingdong Wang, Gang Zeng, Zhuowen Tu, Rui Gan, and Shipeng Li. 2012. Scalable k-nn graph construction for visual descriptors. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 1106–1113.

[34] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 1375–1382.

[35] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[36] Yan-Ming Zhang, Kaizhu Huang, Guanggang Geng, and Cheng-Lin Liu. 2013. Fast kNN graph construction with locality sensitive hashing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 660–674.