A Progressive k-d tree for Approximate k-Nearest Neighbors

Jaemin Jo, Jinwook Seo, Jean-Daniel Fekete

A Progressive k-d tree for Approximate k-Nearest Neighbors

Jaemin Jo*
Seoul National University, Republic of Korea

Jinwook Seo†
Seoul National University, Republic of Korea

Jean-Daniel Fekete‡
Inria, France

ABSTRACT

We present a progressive algorithm for approximate k-nearest neighbor search. Although the use of k-nearest neighbor libraries (KNN) is common in many data analysis methods, most KNN algorithms can only be run when the whole dataset has been indexed, i.e., they are not *online*. Even the few online implementations are not progressive in the sense that the time to index incoming data is not bounded and can exceed the latency required by progressive systems. Exceeding this latency significantly impacts the interactivity of visualization systems especially when dealing with large-scale data. We improve traditional k-d trees for progressive approximate k-nearest neighbor search, enabling fast KNN queries while continuously indexing new batches of data when necessary. Following the progressive computation paradigm, our progressive k-d tree is bounded in time, allowing analysts to access ongoing results within an interactive latency. We also present performance benchmarks to compare online and progressive k-d trees.

Index Terms: H.3.m [Information Systems]: Information Storage and Retrieval—Miscellaneous;

1 INTRODUCTION

Progressive data analysis has recently gained in popularity due to its ability to deliver ongoing results before the whole computation is completed [9, 30]. In contrast to previous computation paradigms such as online computation [1], progressive algorithms deliver estimates at a bounded rate: they are guaranteed to return a partial result in a specified delay to comply with human attention constraints.

However, despite the advantages of progressive computation, it is not always simple or even possible to convert a sequential algorithm to a progressive one, and such a hurdle hinders the applicability of progressive computation to a wider range of data analyses.

In this paper, we address one important problem: computing k-nearest neighbors (KNN) progressively. KNN is an optimization problem of finding the k closest points to a query point in a multidimensional metric space. The KNN problem is a building block of many computer vision and machine learning methods such as feature matching [20], clustering [31], classification [23], projection [25], and non-parametric density estimation [10]. Thus, designing an efficient progressive algorithm for the KNN problem is an important step towards extending the applicability of progressive systems.

This article presents a progressive k-d tree algorithm which can process KNN queries while continuously indexing new batches of data. It is based on previous work on improved k-d trees [21,28]. We first review previous approaches made for KNN search and discuss new challenges and requirements in interactive scenarios. Then, we improve the sequential k-d tree algorithm to first become online and then progressive. Finally, we report on performance benchmarks to compare a popular open-source k-d tree implementation (i.e., FLANN [21]) and ours.

*e-mail: jmjo@hcil.snu.ac.kr
†e-mail: jseo@snu.ac.kr
‡e-mail: Jean-Daniel.Fekete@inria.fr

2 RELATED WORK

In this section, we first introduce previous approaches to the k-nearest neighbor problem in parallel with progressive systems for interactive analysis. Then, we present the challenges and opportunities for designing a progressive algorithm for the KNN problem.

Formally, given N points in $P = \{p_1, \cdots, p_N\}$ and a query point p, a KNN search finds (the indices of) the k nearest points of the query point in P. Formally, this operation can be stated as follows:

$$KNN_k(p) \mapsto \{i_1, i_2, \cdots, i_j, \cdots i_k\} \text{ where } i_j \in [1,N]$$

$KNN_k(p)$ is a set of indices that satisfy the following condition:

$$\forall i \in KNN_k(p)\ \forall j \in [1,N] - KNN_k(p), ||p,p_i|| \leq ||p,p_j||$$

where $||p,p_i||$ is the distance between p and p_i.

One straightforward approach is to calculate the distances from the query point p to every point in the dataset and take the k closest. However, this method is very inefficient since it has to iterate over all points and thus has a time complexity of $\theta(N)$. A more efficient approach is to use a search data structure or an indexing method. In the recent years, there has been important advances in such data structures and algorithms to speed-up KNN queries. However, these advances have mostly focused on optimizing the query time, considering that the indexing was done once for all and thus the indexing time was less important than the query time [7]. However, for progressive systems, both times are important because data can be loaded progressively, the KNN queries can be done progressively, and therefore the index should be updated progressively too.

A popular approach to improve the query time is to compute approximate k-nearest-neighbors instead of exact ones. Approximate k-nearest neighbor search (AKNN) techniques are more efficient than exact KNN but all of them also require building an index. For example, the most efficient method to date, the hierarchical navigable small-world graph (HNSW) [17], needs all data point to be loaded upfront and a special graph structure to be built before querying. From the visual analytics point of view, such a precomputation leads to long loading times, hampering the interactivity of the entire system. Only a few AKNN techniques, such as FLANN [21], support online updates; they allow inserting new points after an index is built. However, this is not sufficient for interactive visual analytics because the insertion time is not bounded. Indeed, we observed that the FLANN algorithm pauses longer than ten seconds to update the indexes for a few hundred thousand points, exceeding the time limit to keep the user's attention [22].

The simplest data structures for AKNN are space-partitioning trees. They recursively divide a multidimensional space and build a tree structure that can be used to accelerate searching. Initially designed for exact KNN matches, k-d trees [5] have been one of the most widely used methods for KNN queries. A k-d tree iteratively splits the space with hyperplanes and builds a binary tree, allowing a logarithmic time complexity for KNN search. At each level in the binary tree, data is divided into two groups by the dimension in which the data has the highest variance.

Later, variants of k-d trees have been proposed to further reduce the query time. Beis and Lowe [4] showed that limiting the number of visited nodes in a k-d tree could bring a large speedup in the query time with a small loss in accuracy. For KNN search in higher

dimensional spaces, Silpa-Anan and Hartley [28] presented the idea of multiple randomized k-d trees where data is recursively split with a dimension that is randomly chosen from a small set of candidate dimensions with the highest variance. Muja and Lowe [20] identified two best algorithms for KNN querying: randomized k-d trees and hierarchical k-means trees, and presented an algorithm that selects optimum parameters for the algorithms in terms of speed and accuracy criteria. Later, they extended their work to perform distributed nearest neighbor search on multiple machines [21].

Another body of research adopted partitioning strategies with hyperplanes not aligned with axes. Examples include non-axis-aligned hyperplanes [29], random projection trees [8], trinary projection tree [12], ball tree [14], and several open-source implementations [6, 19, 23, 26].

Hash-based techniques use a set of *locality-sensitive hashing (LSH) functions* [2]. The core idea is that a pair of two close points is more likely to fall into the same bucket after hashing than a pair of two distant points. Therefore, hash-based techniques can efficiently search for neighbors by looking up the buckets that a query point falls into. The strength of hash-based techniques is that they can provide a theoretical base on the search quality. Examples include LSH forest [3], multi-probe LSH [15], and kernelized LSH [13].

Graph-based techniques model multidimensional data points as a graph by mapping the points to vertices and the neighborhood relationships to edges. Once the graph is built, AKNN search can be done by exploring the graph. From the KNN graph, Sebastian and Kimia [27] selected a few well-separated vertices (i.e., *seeds*) and iteratively moved the seeds to points that are closer to the query point until satisfactory neighbors were found. Hajebi et al. [11] provided theoretical guarantees for the accuracy and the computational complexity of such a greedy method. Wang et al. [32] proposed a new approach to construct approximate KNN graphs by building exact neighborhood graphs for hierarchically divided data and combining the graphs. Recently, more sophisticated graph structures such as navigable small world graphs are used for KNN queries. In addition to short-range links in a traditional neighbor graph, navigable small world graphs have long-range links that connect two distant points. Malkov et al. [16] showed that these long-range links can be used for logarithmic scaling of neighbor exploration. Later, Malkov and Yashunin [17] further improved the performance by introducing hierarchical structures to navigable small world graphs. Yet, the construction of the graphs is more costly than the other methods and cannot easily be done online.

Throughout a few decades of KNN research, query time (i.e., time taken to perform a KNN search) has been the key measure for evaluating the performance of various techniques. Indeed, in most studies mentioned in this section, authors assumed that data points had already been inserted to an index and measured the time taken to process queries. This is also the case with benchmarks in the public domain [7, 18]. However, such benchmarks are meaningful only when the data is kept constant. In more interactive scenarios, the data can be changed dynamically by loading new data (e.g., streaming) or deleting a subset of data through user interaction. Thus, it is necessary to keep the whole process of KNN queries, including building, and querying the index, interactive. In this paper, inspired by Progressive Visual Analytics [30], we introduce a progressive k-d tree for approximate k-nearest neighbor algorithm that can keep the latency for building, maintaining, and querying the index within a specified time bound. We chose to start with the multiple randomized k-d tree algorithm which is simple yet one of the most efficient algorithms for AKNN queries [20].

3 Approaches for the Approximate k-Nearest Neighbor Problem

In this section, we first describe a sequential algorithm using randomized k-d trees or a k-d forest for approximate k-nearest neighbor (AKNN) search, and then improve it first to be online and then progressive. Among many algorithms mentioned in the related work section, we chose the k-d forest because 1) it is known to be efficient and yet easy to implement [20] and 2) an online version of the algorithm is available in open-source [19] so we could directly compare our progressive version to the online version.

3.1 A Sequential Algorithm

A k-d tree is a binary tree built by recursively partitioning a multidimensional space using axis-aligned hyperplanes [5] and used thereafter to search, guaranteeing $\log_2 N$ search time and $N \times \log_2 N$ build time. At the root node, the algorithm chooses a *cutting* dimension which has the largest variance, and assigns points to child nodes: the points whose value on the cutting dimension is less than the median are assigned to the left node and the remaining points are assigned to the right node. This procedure repeats until only one point remains in a node. In the randomized k-d tree forest, we randomly choose a cutting dimension among the top n (e.g., $n = 5$) dimensions with the largest variance. This allows us to build multiple randomized trees and represent high-dimensional spaces more effectively. The algorithm can be described as follows:

Algorithm 1 A sequential algorithm for building a randomized k-d tree of the given l points in L

```
 1: procedure BUILDSEQUENTIAL(L)
Input: L is a list of l points of D dimensions.
Output: A randomized k-d tree
 2:     if L has only one point then
 3:         node ← a new leaf node
 4:         node.point ← L[0]
 5:         return node
 6:     end if
 7:     node ← a new internal node
 8:     calculate the variance of each dimension in L
 9:     node.cutdim ← a random dimension with large variance
10:     node.cutval ← the median of values of node.cutdim in L
11:
12:     left ← [p for p in L if p[node.cutdim] ≤ node.cutval]
13:     right ← [p for p in L if p[node.cutdim] > node.cutval]
14:
15:     node.left ← BuildSequential(left)
16:     node.right ← BuildSequential(right)
17:     return node
18: end procedure
```

Algorithm 1 has three strong limitations: First, it requires all points to be already loaded in main memory (L) before building the randomized k-d trees. Such a constraint forces analysts to wait until all data is read from a disk before performing any analysis. Second, once the data structures are built, the algorithm does not allow any modification on them (e.g., by inserting new points). Finally, the running time of the algorithm solely depends on the size of input (i.e., l) and thus latency cannot be controlled.

3.2 An Online Algorithm

In contrast to a sequential algorithm, an online algorithm allows adding new points to trees even after the trees are built. This benefits interactive systems in that analysts do not have to wait until all data is loaded. Rather, the data is split into mini-batches, loaded onto the system incrementally, and can be used for further online algorithms. Analysts can access the running result between the mini-batches, obtaining an approximation of the final results.

However, we found such online data structures for AKNN search was rare; the FLANN library [21] is the only one that supports online updates. It builds a small k-d tree of the points in the first mini-batch using the same algorithm as the sequential one (i.e., Algorithm 1). Then, other points can be added into the tree thereafter when needed.

The insertion procedure of the FLANN library is very akin to that of a binary tree; starting from the root node, each point moves to

either the left or the right child by comparing its value at the *cutdim* dimension and *cutval* of an internal node until it reaches a leaf node. Then, the leaf node becomes an internal node, and the two points become the children of the node. Algorithm 2 describes the insertion procedure in more detail.

Algorithm 2 An algorithm for inserting a new point p into a randomized k-d tree with a root node *node*

```
1:  procedure INSERT(node, p)
Input: node is the root of a k-d tree and
2:  p is a new D-dimensional point
Output: p is inserted as one of the leaf nodes in the tree.
3:      if node is a leaf node then
4:          mark node as an internal node.
5:          calculate the absolute difference between p and
6:          node.point at each dimension
7:          choose a cutdim dimension with the largest difference
8:          cutval ← (p[cutdim] + node.point[cutdim])/2
9:          if p[cutdim] ≤ cutval then
10:             node.left ← a new leaf node with a point p
11:             node.right ← a new leaf node with a point node.point
12:         else
13:             node.left ← a new leaf node with a point node.point
14:             node.right ← a new leaf node with a point p
15:         end if
16:         return
17:     end if
18:     if p[node.cutdim] ≤ node.cutval then
19:         Insert(node.left, p)
20:     else
21:         Insert(node.right, p)
22:     end if
23: end procedure
```

As more points are inserted to a k-d tree, the tree can become unbalanced, deteriorating the query time. In the FLANN library, the distribution of the points in the first mini-batch heavily affects the overall performance since they are used to build a "skeleton" of the tree. At worst, if all the updates after the first mini-batch are skewed to one side of the k-d tree, all the remaining points are inserted in a linked list and the search time becomes linear with the number of points. This implies the need to rebalance the tree when possible. In real cases, the unbalance is never that extreme, but can vary substantially if the data added has a different distribution than the original tree. The imbalance leads to a slower query time with little degradation of the quality. On the other side, when updating a tree for a large dataset, assuming the data is stationary, the distribution of incoming data will at some point converge to the distribution of the whole dataset and the tree will remain balanced after inserting new points.

FLANN's implementation of k-d trees uses a simple strategy for rebalancing the trees: it re-constructs all trees each time the dataset doubles in size from the initial dataset (i.e., the first mini-batch). Therefore, the k-d trees can become unbalanced as new data is loaded but eventually will be re-constructed. When loading a large dataset progressively, even if the incoming distribution matches the current k-d tree structure, FLANN will always re-construct its k-d trees when the dataset doubles in size. To sum up, the FLANN implementation suffers from three problems:

1. the k-d tree may become unbalanced when data is added, leading to longer KNN searches,
2. the k-d tree is always re-constructed when the data doubles in size, leading to very long interruptions in the KNN search at unpredictable moments,
3. the k-d tree is always re-created when the data doubles in size, even when it remains balanced.

3.3 A Progressive Algorithm

To overcome the limitations of online k-d trees, we made three main changes to the FLANN algorithm:

1. we maintain a quality measure for each k-d tree and trigger a reconstruction process when the measure satisfies a certain criterion,
2. when needed, we construct a fresh and balanced k-d tree with all the points
3. the construction is done in a parallel/interleaved task using a build queue, thus spreading the load and avoiding brutal changes in query times. When a new k-d tree is built, we drop the most unbalanced one and replace it with the new one,

To estimate the quality of a k-d tree, we use the following method: assume a k-d tree of size N is balanced; its depth is $\lceil \log_2 N \rceil$. The points are stored as leaves so accessing a point will require $\log_2 N$ operations. When a k-d tree becomes unbalanced, its depth will vary and the query time for a point P will be proportional to the depth of P. On average, the query time for accessing P is: $\phi_P \times \text{depth}(P)$ where ϕ_P is the probability to search P and $\text{depth}(P)$ is the depth of P in the tree. On a balanced k-d tree, the cost of querying for an arbitrary point is $\log_2 N$ whereas for a specific k-d tree T, the cost $c(T)$ is:

$$c(T) = \sum_{p \in T} \phi_p \times \text{depth}(p) \tag{1}$$

If the tree is perfectly balanced, the cost equals to $\log_2 N$, otherwise, it becomes higher. The quality of a tree is thus the difference between the actual cost and the lower bound is the number of additional operations we should perform to search a point on average. To decide when we should trigger the computation of a fresh tree, this loss should be compared to the cost of rebuilding the whole tree: $N \log_2 N$. We accumulate the loss in the query time (i.e., $c(T) - \log_2 N$) for each query and once the accumulated loss exceeds a threshold, or a specific proportion of the rebuilding cost (i.e., $\alpha \times N \log_2 N$, where α is a reconstruction weight), we start the reconstruction procedure. In practice, we do not compute Equation 1 for every update but maintain the cost incrementally. For each point p, we maintain the depth of the point, $\text{depth}(p)$, and the number of times the point is searched, $\text{freq}(p)$. Let's define $\sum \text{freq} = \sum_{p \in Q} \text{freq}(p)$, then ϕ_p can be calculated by $\phi_p = \frac{\text{freq}(p)}{\sum \text{freq}}$. When a point p is searched, $\text{freq}(p)$ will increase by one and the updated cost C' is computed from the the current cost C:

$$C' = \frac{\sum \text{freq} \times C + \text{depth}(p)}{\sum \text{freq} + 1}$$

After updating the cost, we increment $\text{freq}(p)$ and $\sum \text{freq}$ by one.

When we need to re-construct a k-d tree (i.e., the accumulated loss exceeds the threshold), we distribute the reconstruction load across multiple iterations by building the tree *incrementally*. To this end, we implement a non-recursive version of Algorithm 1 using a build queue, allowing the whole procedure to be interleaved between iterations. The progressive reconstruction algorithm (Algorithm 3) is similar to Algorithm 1 except that recursive calls are replaced with insertion on the queue.

To achieve progressiveness, the algorithm should work only for a given number of operations and stop, allowing the user or the system to access the ongoing results. For each iteration, a certain number of operations are given to the algorithm and the algorithm assigns the operations to insertion tasks and reconstruction tasks. An insertion task reads one point from data and inserts it to the k-d trees as described in Algorithm 2. If reconstruction is needed after insertion, the algorithm builds a new k-d tree incrementally by calling the function *Initialize* first and the function *ProcessQueue* in the following iterations, as described in Algorithm 3. When the new k-d tree is built, the algorithm replaces the most unbalanced tree with the new one.

The number of allowed operations is a parameter specified by the user. The algorithm can freely use this number to perform either insertion or reconstruction tasks. In our benchmark, we used a

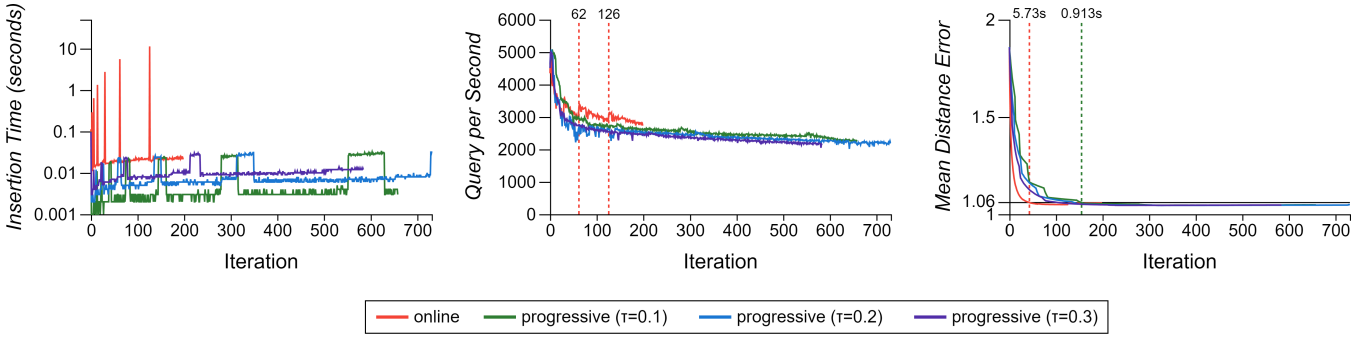

Figure 1: **Benchmark results** with one million points from the GloVe dataset [24]. FLANN's online algorithm (the red line) rebuilds the k-d trees each time the data size doubles. When inserting a new mini-batches of points (the leftmost chart), the online algorithm produces a delay longer than 10 seconds which hampers the interactivity of visualization systems. Our progressive algorithms are bounded in time, yielding gains in insertion time with a small loss in query time.

Algorithm 3 A progressive algorithm for building a new k-d tree

```
1:  procedure INITIALIZE(L)
Input: L is a list of l points of D dimensions.
2:      queue ← a new work queue
3:      root ← a new node
4:      queue.push((root, L))
5:  end procedure
6:
7:  procedure PROCESSQUEUE(ops)
Input: ops is the number of operations allowed for reconstruction
Output: returns true if reconstruction is done
8:      count ← 0
9:      while count < ops and queue is not empty do
10:         node, L ← queue.pop()
11:         count ← count + 1
12:         if L has only one point then
13:             mark node as a leaf node.
14:             node.point ← L[0]
15:             continue
16:         end if
17:         calculate the variance of each dimension in L
18:         node.cutdim ← a random dimension with large variance
19:         node.cutval ← the median value of node.cutdim in L
20:
21:         left ← [p for p in L if p[node.cutdim] ≤ node.cutval]
22:         right ← [p for p in L if p[node.cutdim] > node.cutval]
23:
24:         node.left ← a new internal node
25:         node.right ← a new internal node
26:
27:         queue.push((node.left, left))
28:         queue.push((node.right, right))
29:     end while
30:     return true if queue is empty
31: end procedure
```

simple strategy: we assigned a specific proportion (τ) of the allowed operations to insertion tasks and $1-\tau$ to reconstruction tasks. For example, when $\tau = 0.5$, half of the operations are used to insert new points and the other half to re-construct a tree.

4 Benchmark

We conducted benchamarks to compare the performance of the online and progressive k-d tree algorithms. We used the GloVe [24] dataset that had 100 dimensions. We randomly took 1M points from the dataset as train data (i.e., points that were inserted to k-d trees) and 1K points as test data (i.e., points that were queried). The order of points was kept as in the original dataset. We computed the exact 20 neighbors (i.e., $k = 20$) before the benchmark to measure the quality of answers. For each iteration, we gave 5,000 operations to both online and progressive k-d trees. The online version used up all operations to add new points (i.e., 5,000 points were inserted to k-d trees during one iteration). For the progressive k-d tree, we used three different values for τ: 0.1, 0.2, and 0.3. We set the value of α (i.e., a reconstruction weight) to 0.25.

To assess the quality of answers, we computed the mean distance error (MDE) which is the mean ratio between the distances from each query point to its exact k-th nearest neighbor and to its approximate k-th nearest neighbor. An MDE of one means that the exact neighbors were found, and an MDE of two means on average the algorithm found neighbors that are twice farther than the exact ones.

Fig. 1 shows the results of the benchmark. Since the online version used all 5K operations to insert new points, the corresponding red line ends at the 200th iteration (1M / 5K = 200). The results revealed the limitation of the online tree: at the 128th iteration, the online tree produced a peak latency in insertion time which was longer than 10 seconds. In contrast, the progressive trees showed more consistent insertion time always shorter than 0.1 second.

In the online tree, we could see a performance gain in querying speed after tree reconstruction (i.e., at the 62th and 126th iterations in the middle chart of Fig. 1). The progressive trees showed lower performance but the gap could narrow by adjusting the value of α (i.e., the reconstruction weight). In terms of accuracy, the MDE converged to near 1.06 for all algorithms. The online tree took the smallest number of iterations to reach the final MDE because it used all operations to insert new points from data so exact neighbors were more likely to be in the trees and searched. However, due to its longer insertion time, the online tree took the longest to reach the final MDE (marked with dotted lines in Fig. 1), which suggests the effectiveness of our progressive k-d trees.

5 Conclusion

In this article, we presented a progressive k-d tree algorithm for approximate k-nearest neighbor search. We showed three major changes to the previous k-d trees: maintaining a quality measure to determine when to reconstruct trees, triggering the reconstruction when really needed, and introducing a build queue to spread the reconstruction load. In our benchmark, our progressive k-d tree alleviated brutal changes in query time while keeping the speed and accuracy comparable to those of online k-d trees. Due to the limited paper length, we focused on adding points, but deletions and filtering can also be done progressively. The implementation of our progressive k-d tree is available at `github.com/e-/PANENE`.

As future work, we will integrate our progressive k-d tree to the ProgressiVis toolkit [9]. Furthermore, we will investigate the ones based on locality sensitive hashing (LSH) since they are very fast at building their index.

REFERENCES

[1] S. Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1):3–26, 2003. doi: 10.1007/s10107-003-0436-0 1

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pp. 459–468. IEEE, 2006. doi: 10.1145/1327452.1327494 2

[3] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pp. 651–660. ACM, 2005. doi: 10.1145/1060745.1060840 2

[4] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pp. 1000–1006. IEEE, 1997. doi: 10.1109/CVPR.1997.609451 1

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. doi: 10.1145/361002.361007 1, 2

[6] E. Bernhardsson. Annoy. `https://github.com/spotify/annoy`. Last accessed: 2017-07-22. 2

[7] E. Bernhardsson. Benchmarking nearest neighbors. `https://github.com/erikbern/ann-benchmarks`. Last accessed: 2017-07-22. 1, 2

[8] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pp. 537–546. ACM, 2008. doi: 10.1145/1374376.1374452 2

[9] J.-D. Fekete and R. Primet. Progressive analytics: A computation paradigm for exploratory data analysis. *ArXiv e-prints*, July 2016. 1, 4

[10] K. Fukunaga. *Introduction to statistical pattern recognition*. Academic press, 2013. 1

[11] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, p. 1312, 2011. doi: 10.5591/978-1-57735-516-8/IJCAI11-222 2

[12] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 3392–3399. IEEE, 2010. doi: 10.1109/CVPR.2010.5540006 2

[13] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2130–2137. IEEE, 2009. doi: 10.1109/ICCV.2009.5459466 2

[14] B. Leibe, K. Mikolajczyk, and B. Schiele. Efficient clustering and matching for object class recognition. In *Proceedings of the British Machine Vision Conference*, pp. 789–798. BMVA Press, 2006. doi: 10.5244/C.20.81 2

[15] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pp. 950–961. VLDB Endowment, 2007. 2

[16] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014. doi: 10.1016/j.is.2013.10.006 2

[17] Y. A. Malkov and D. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2016. 1, 2

[18] Mrpt performance comparison. `https://github.com/ejaasaari/mrpt-comparison`. Last accessed: 2017-07-22. 2

[19] M. Muja. Flann - fast library for approximate nearest neighbors. `https://github.com/mariusmuja/flann`. Last accessed: 2017-07-22. 2

[20] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009. 1, 2

[21] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014. doi: 10.1109/TPAMI.2014.2321376 1, 2

[22] J. Nielsen. *Usability engineering*. Elsevier, 1994. 1

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 1, 2

[24] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014. 4

[25] N. Pezzotti, B. P. F. Lelieveldt, L. van der Maaten, T. Höllt, E. Eisemann, and A. Vilanova. Approximated and user steerable tsne for progressive visual analytics. *IEEE Trans. Vis. Comput. Graphics*, 23(7):1739–1752, July 2017. doi: 10.1109/TVCG.2016.2570755 1

[26] rpforest. `https://github.com/lyst/rpforest`. Last accessed: 2017-07-22. 2

[27] T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 3, pp. 291–296. IEEE, 2002. doi: 10.1109/ICPR.2002.1047852 2

[28] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2008. doi: 10.1109/CVPR.2008.4587638 1, 2

[29] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1):579–589, 1991. doi: 10.1007/BF01759061 2

[30] C. D. Stolper, A. Perer, and D. Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. *IEEE Trans. Vis. Comput. Graphics*, 20(12):1653–1662, Dec 2014. doi: 10.1109/TVCG.2014.2346574 1, 2

[31] T. N. Tran, R. Wehrens, and L. M. Buydens. KNN-kernel density-based clustering for high-dimensional multivariate data. *Computational Statistics & Data Analysis*, 51(2):513–525, 2006. doi: 10.1016/j.csda.2005.10.001 1

[32] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-NN graph construction for visual descriptors. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1106–1113, June 2012. doi: 10.1109/CVPR.2012.6247790 2