Proceedings of the 2014 Federated Conference on
Computer Science and Information Systems pp. 93–100

DOI: 10.15439/2014F194
ACSIS, Vol. 2

Experimental evaluation of selected tree structures for exact and approximate *k*-nearest neighbor classification

Aleksander Cisłak
Technical University of Munich,
Department of Informatics,
Boltzmannstr. 3, D-85748 Garching, Germany
Email: a.cislak@tum.de

Szymon Grabowski
Lodz University of Technology,
Institute of Applied Computer Science,
Al. Politechniki 11, 90–924 Łódź, Poland
Email: sgrabow@kis.p.lodz.pl

***Abstract*—Spatial data structures, for vector or metric spaces, are a well-known means to speed-up proximity queries. One of the common uses of the found neighbors of the query object is in classification methods, e.g., the famous *k*-nearest neighbor algorithm. Still, most experimental works focus on providing attractive tradeoffs between neighbor search times and the neighborhood quality, but they ignore the impact of such tradeoffs on the classification accuracy.**

In this paper, we explore a few simple approximate and probabilistic variants of two popular spatial data structures, the k-d tree and the ball tree, with *k*-NN results on real data sets. The main difference between these two structures is the location of input data — in all nodes (k-d tree), or in the leaves (ball tree) — and for this reason they act as good representatives of other spatial structures. We show that in several cases significant speedups compared to the use of such structures in the exact *k*-NN classification are possible, with a moderate penalty in accuracy. We conclude that the usage of the k-d tree is a more promising approach.

I. Introduction

FINDING objects similar to a given one in a large database is a classic research topic, with applications in pattern recognition, multimedia processing, genomic analyses, and other fields. There exist many particular variants of the problem, but one of the most popular is: given object x, we wish to find its k nearest neighbors in a given database D of size n, according to the specified similarity measure. The parameter $k \geq 1$ is usually selected at query time. A naïve solution to this problem is to calculate the distances between the query x and all objects in the database and choose k nearest ones, but this approach requires computation of n distances. If database preprocessing is allowed, we can usually reduce the query time. One of major applications of the proximity search is *classification*, when the query sample is assigned a class label according to the known class labels of its neighbors, and the rest of this paper is focused on this application.

We assume a vector space, in which objects are identified with d real-valued vectors (tuples). The distance function in this space is usually a metric (i.e. it satisfies non-negativity, identity of indiscernibles, symmetry, and the triangle inequality), and the most common particular metrics used are the Euclidean or Manhattan (city-block) one. In vector spaces, the popular search structures include the k-d tree, R-tree, quadtree, X-tree, and their numerous variants. Their common trait is to cluster objects in space, to allow pruning the dataset during most queries. For example, the popular k-d trees partition the space along different coordinates while R-trees group objects in hyperrectangles.

As the (in)famous curse of dimensionality subdues the performance of practically any (however sophisticated) nearest-neighbor finding data structure in high dimensions, it is interesting to investigate how approximate or probabilistic variations of the true nearest neighborhood of the given query affect the classification accuracy. This question has met significant interest from both theoreticians and practitioners, see for instance Arya et al. [1], Indyk and Motwani [2], or Jones et al. [3].

In this work we introduce simple modifications to well-known spatial data structures: the k-d tree and the ball tree, in order to explore how approximate or probabilistic speedup idea (e.g., via more aggressive pruning than in the original method) affect the time-accuracy tradeoff. While our conclusions are hardly definite, we believe that experimentations with popular (and relatively easy to implement) data structures have their own, practice-oriented, value.

II. K-d tree

One of the oldest spatial data structures, the *k-d tree*, was introduced by Jon Louis Bentley in 1975 [4], and the name refers to k dimensions it operates on. To avoid confusion with the number of neighbors in the *k*-NN rule, from now on we will use the symbol d for the number of dimensions.

The k-d tree is a binary tree, where every instance of the indexed data corresponds to one node. The left child together with its descendants contain points whose values of the feature (coordinate) f are smaller than the f value of the splitting hyperplane H — analogously, the right child together with its descendants contain points with higher f values. As regards the selection of H, the most popular approach is to choose the point whose f is the median, and divide the points into

978-83-60810-58-3/$25.00

two parts of equal size (assuming that the number of points to divide is even).

A. Construction of the tree

During the construction, the current feature space is recursively divided into two subspaces, with half of the points lying in each subspace. This division is based on the current dimension, and the algorithm switches to the next dimension with each step as the recursion progresses. After all dimensions have been processed, it goes back to the first dimension in a circular manner. The recursion stops when there is only a single point left, and this point is stored in a leaf. The result is a binary tree, where inner nodes represent points situated on the splitting hyperplanes, and leaves represent the rest of the given data.

B. K-nearest neighbor search

When the *k*-NN search is performed, the tree is traversed from the root to the leaf. The algorithm goes left or right depending on feature values, and this can be represented by following relations, where Q is the queried point, N is the point corresponding to the current node, and d is this node's split dimension: $Q_d \leq N_d \rightarrow left$, $Q_d > N_d \rightarrow right$. Dimensions are switched in the same way as during the construction, so that the dimension which is checked at each level is always the one on which the space was split in halves.

After a leaf has been found, the search goes back towards the root, following the same path which was traversed downwards. From this moment, the algorithm maintains the list of k points with smallest distances to the queried point Q, and tries to update it every time a new node is visited.

At each step upwards, there is a possibility of inspecting a subtree whose root is the sibling of the current node N_{cur}. Such a subtree can be pruned if and only if k points have already been found, and all distances from Q to these k points are smaller than or equal to the distance between Q and the point P_{spl}. P_{spl} represents the point located on the splitting hyperplane, and it is associated with the node which is the parent of N_{cur}. This is demonstrated by the relation in Figure 1, where P represents the set of points found so far, and D refers to the distance.

$$prune \leftrightarrow |P| = k \wedge \bigvee_{p \in P} D(p, Q) \leq D(Q, P_{spl})$$

Fig. 1. Pruning condition in an exact k-d tree.

If the subtree S could not be pruned and it has been checked, the list of best points from S must be merged with the current list of best points. This is rather straightforward, because we simply select k points with lower distances from both lists, or if the size of the combined list would be smaller than k, all points are retained.

The whole *k*-NN search procedure can be summarized as follows:

1) Find the leaf.
2) Go to the parent and try to update the list of k best points.
3) Recursively check the subtree whose root is the sibling of a current node, unless the relation in Figure 1 is satisfied.
4) If checked the subtree, merge the lists of best points.
5) Repeat 2. until found the root of the whole tree.

C. Complexity

As regards search time complexity, the average case for the nearest neighbor lookup (1-NN), under favorable assumptions (discussed in the next sentences), is equal to $O(\log n)$ [4], and the worst case, where all points are checked, is clearly equal to $O(n)$. Performance degrades to linear time when, roughly speaking, the number of dimensions is large, and for this reason the number of visited nodes also tends to be large. In general, for optimal performance the relation $2^d \ll n$ should hold [5]. When it comes to *k*-NN, the average case expands into $O(\log n \cdot \log k)$, and the worst case expands into $O(n \log k)$, assuming a heap is used to maintain the list of best points. In practice, a k-d tree might turn out to be slower than a naïve method, due to the search procedure overhead.

The construction takes $O(n \log n)$ time, assuming the median required to split points in halves is found with a linear worst-case time algorithm [6, Ch. 9]. Since the number of nodes is proportional to the number of points, the space complexity is equal to $O(n)$.

III. Ball tree

The aim of the *ball tree* is akin to the one of the k-d tree, as it attempts to reduce time spent on a nearest-neighbor query by partitioning the feature space. Just as the name suggests, this is achieved by constructing closed balls, that is geometric objects containing a sphere S and the space inside S.

The ball tree is a binary tree, where each internal node N_I is associated with one ball, and this ball contains all balls of the descendants of N_I. Hence, the biggest ball is stored as a root and it contains all other balls in the tree. The training data are stored in the leaves, with one leaf corresponding to one training instance, and internal nodes act only as guidance during the search. Subspaces resulting from the partitioning are clearly overlapping, unlike in the case of other structures, such as the aforementioned k-d tree.

A. Construction of the tree

We opt for the bottom-up construction algorithm, which is the most efficient one with respect to the search time of *k*-NN queries performed on the resulting tree [7]. This efficiency results from the fact that we try to reduce the volume (the radius) of the balls.

At the beginning of the construction, we create a set of balls from the training data, with one ball corresponding to one instance. At each step, we search for a pair of balls, whose resulting ball R_B (one that contains the selected pair of balls) is the smallest. Subsequently, two selected balls are set as children of R_B, and R_B is inserted back into the set of

available balls. Thus, at each step we reduce the size of the set by one. When we are left with only one ball, this ball is associated with the root node and the algorithm terminates.

For more details on other construction algorithms, we refer the reader to the original article by Omohundro [7].

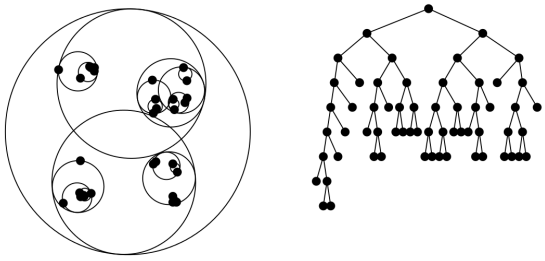

Fig. 2. Space partitioning with balls and the resulting binary tree using the bottom-up construction algorithm; reproduced from Omohundro [7].

B. K-nearest neighbor search

When the *k*-NN search is performed, the tree is traversed in the depth-first manner. At each step, there is a possibility of recursively inspecting two children of the current node, and each child can be pruned together with all of its descendants, if and only if the following condition is satisfied. For safe pruning, it is required that we have already found k points, and the ball that is centered at the query point and that contains all k points found so far does not intersect the ball of the child in question. This is demonstrated by the relation in Figure 3, where *P* represents the set of points found so far, and B_Q and B_C refer to the ball centered on the query and the ball centered on the child, respectively.

$$prune \leftrightarrow |P| = k \wedge intersects(B_Q, B_C) == false$$

Fig. 3. Pruning condition in an exact ball tree.

Whole *k*-NN search procedure can be summarized as follows:

1) If we are at the leaf, try to update the list of k nearest neighbors and terminate.
2) If k nearest neighbors have already been found, compute the ball that is centered at the query point and contains these k points.
3) Recursively inspect the left child, unless the relation in Figure 3 is satisfied.
4) If *k* nearest neighbors have already been found, recompute the ball from step 2).
5) Recursively inspect the right child, unless the relation in Figure 3 is satisfied.

C. Complexity

We construct a binary tree, whose bottom level contains n nodes, because all training data are stored as leaves. For this reason, the space complexity of the ball tree is equal to $O(n)$.

As regards the search time complexity, the worst case where all nodes are checked is clearly proportional to the size of the tree, that is $O(n)$. Because of the time overhead resulting from tree traversal, the ball tree can turn out to be slower than a brute-force algorithm. Assuming optimal space partition, where branches can be pruned, for a *k*-NN search it is possible to achieve the best case bound of $O(\log n + k)$.

Since our focus is solely on finding nearest neighbors, we ignore the preprocessing time complexity, however, it is worth noticing that it can be fairly expensive. For instance, a naïve bottom-up construction algorithm requires $O(n^3)$ time.

IV. Approximate algorithms

The objective of approximate search algorithms based on tree data structures introduced in previous sections is to decrease the time spent on classification, at the cost of an increase in the error rate. This is accomplished by limiting the visited area of a tree.

We utilize the notion of *bounds*, where after the specified bound has been crossed, current list of nearest neighbors is returned. This means that the bound should be chosen with care, since the list can actually contain less than k points when the algorithm terminates. For probabilistic pruning (Subsection IV-D), we ensure that branches are pruned only after k points have been found.

It is to be noted that time and space overheads presented in this section are relevant only to described modifications, and for a complete analysis, complexities of exact algorithms should be added.

A. CPU time bound

CPU time refers to the time spent by the processing unit on executing actual instructions, which means that it is not affected by context switches or time changes. The bound is checked for the first time after the leaf has been found, which is required for the ball tree, and allows us to concentrate the search in the bottom part of the k-d tree. Bounds checking introduces a small time overhead $O(v)$, where v refers to the number of nodes visited by the algorithm after the first leaf has been reached. The space overhead is constant.

B. Depth bound

The depth bound specifies a maximum depth of the tree that can be reached by the search algorithm. This is relevant only for the k-d tree, since in the ball tree all training data are situated in the leaf nodes. The depth is checked every time a new node is visited, and for this reason the time overhead is equal to $O(v_t)$, where v_t refers to the total number of nodes visited by the algorithm. The space overhead is equal to $O(n)$, because every node stores its depth.

C. Node bound

The idea of the node bound is simply to set a hard threshold t on the number of nodes, which can be checked by the algorithm. Analogically to the CPU time bound, this bound is checked for the first time after the leaf has been found, which

is required for the ball tree, and allows us to concentrate the search in the bottom part of the k-d tree. For this reason, the total number of nodes which have been traversed might turn out to be greater than t. Again, bounds checking introduces a small time overhead $O(v)$, where v refers to the number of nodes visited by the algorithm after the first leaf has been reached.

D. Probabilistic pruning

Similarly to approximate variants introduced in previous subsections, the aim of this algorithm is to limit the space that is inspected during the search procedure. This is achieved by introducing the pruning factor σ, which describes the probability that the subtree is pruned, even if the pruning condition presented in Figure 1 for the k-d tree or in Figure 3 for the ball tree is not satisfied. For instance, if $\sigma = 25\%$, every time a subtree should be inspected, there is a $1/4$ chance that it will be ignored instead. It should hold that $\sigma > 0 \wedge \sigma \leq 1$. It is worth noticing that in the case of $\sigma = 1$, the algorithm's behavior is in fact deterministic, as all possible branches are pruned.

Since pseudorandom number generation can be done in constant time, the time overhead is proportional to the number of pruning decisions which have to be taken. These decisions are made only when the subtree cannot be safely pruned, and the complexity is equal to $O(1)$ in the best case, since then all subtrees can be safely pruned. As regards the worst case, a decision has to be made every time a new subtree is encountered, and for this reason the time overhead expands into $O(n)$. The space overhead is constant.

E. Best bin first (BBF)

The *best bin first* (BBF) algorithm [8] is relevant only to the k-d tree and it aims to increase the accuracy of an approximate search. Since an inexact algorithm does not visit all nodes which would be required to provide an exact answer, the order in which the nodes are visited is crucial to the performance in terms of an error rate. After the leaf has been found, instead of following the path to the root from the bottom, going up one level per step, the algorithm selects an optimal node lying on this path. Subsequently, it continues to choose an optimal node from the remaining ones, until the path is exhausted, or some limit (such as the node bound) is exceeded. We choose a straightforward method to determine node's optimality, which selects the node whose splitting hyperplane is closest to the queried point [8].

The time overhead depends on the kind of priority queue that is used for selecting the smallest distance. We have $O(V_T)$ inserts and $O(t)$ delete-min operations, where V_T represents the total number of nodes traversed by the search procedure, and t is the number of nodes visited after the first leaf has been found. For instance, if the Fibonacci heap [9] were used, the complexity would be equal to $O(V_T + t \log V_T)$ amortized time. As regards the space overhead, it is possible to achieve a bound of $O(V_T)$. These complexities refer only to maintaining a priority queue and not to bounds checking.

V. Experimental results

The error rates presented in this section were calculated using the *leave-one-out* method for the k-d tree, and 5-fold *cross validation* for the ball tree, the reason for the second method being computational demands. We used the *Manhattan* metric for the similarity measure. Classification times presented in the diagrams refer to CPU time in milliseconds spent on classifying one sample, and the preprocessing time is not taken into account. CPU time values are arithmetic mean values obtained in the course of three runs, in order to minimize an influence of external factors such as cache utilization. Error rates presented for probabilistic variants are arithmetic mean values obtained in the course of five runs. The value $k = 5$ was selected arbitrarily. The machine used for experiments was equipped with Intel e2160 processor running at 2.9 GHz and 4 GB DDR2 memory. The code was compiled using the GCC suite and run on Ubuntu 12.04 64-bit operating system.

Only selected results are presented due to space constraints, nevertheless, results reported for different data sets (for a short description of the sets, see Appendix A) were consistent to a satisfactory degree. Unexpected or particularly unusual behavior was rare, and it was most probably caused by a unique structure of specific data set in question.

Selected diagrams were published in the Bachelor's Thesis of the first author [10].

A. CPU time bound

CPU time bound values are significantly smaller than the times spent on classification shown in other diagrams. This results from the fact that the execution time spent on classifying one sample is calculated as the total time used by all procedures in the *k*-NN algorithm, such as allocating the memory or comparing class counts. On the other hand, the CPU time bound refers only to the internal approximate search procedure.

Just as expected, we observed a growth in the error rate as the time bound decreased. Above certain higher bound value there was no change with respect to exact algorithms, and as the bound approached zero, there was a dramatic increase in the error rate. For the k-d tree with Ferrites data set, there was only a marginal increase in the error rate until the bound value of about 0.35 ms, as demonstrated in Figure 4. Other data sets and the ball tree behaved similarly, although relative increases in the error rate were more significant.

B. Depth bound

The depth bound introduced a rather moderate increase in the error rate, although associated time decreases were lower than in the case of other bounds. For instance, for Banknotes data set, there was almost no increase in the error rate up to the bound value of 7, with a roughly 3-fold decrease in classification time (see Figure 5). At the other extreme was the Isolet data set, for which both time decrease and error rate increase were more substantial than for other sets (see Figure 6).

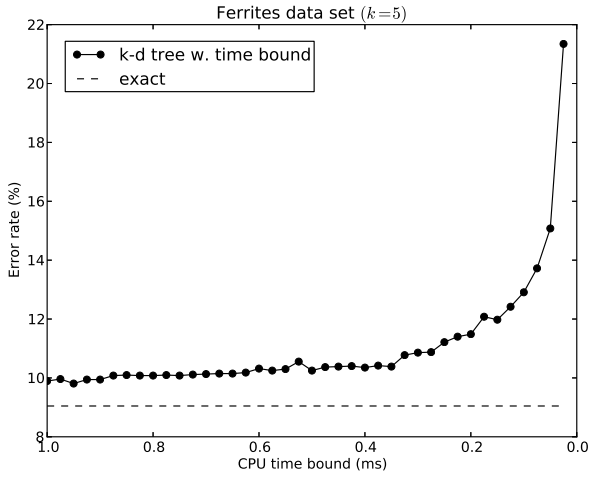

Fig. 4. Error rate vs CPU time bound for the k-d tree with Ferrites data set.

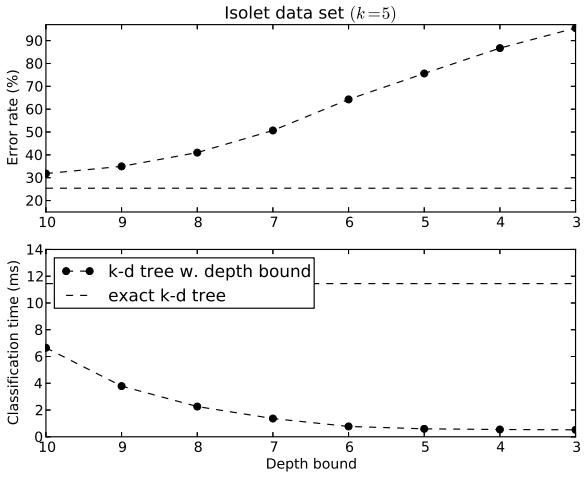

Fig. 6. Error rate and classification time vs depth bound for the k-d tree with Isolet data set.

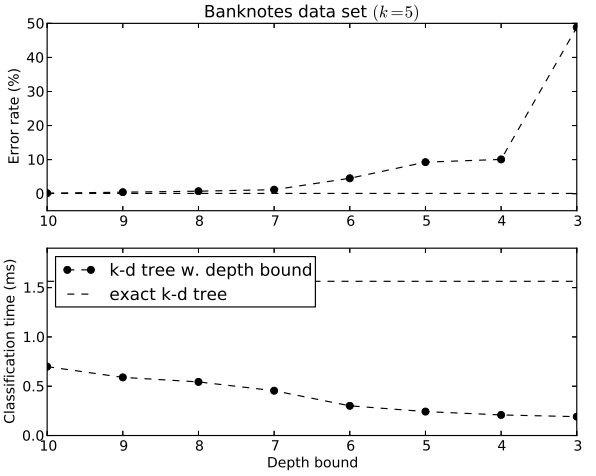

Fig. 5. Error rate and classification time vs depth bound for the k-d tree with Banknotes data set.

C. Node bound

Similarly to other algorithms with bounds, as the node bound value decreased, expected increase in the error rate and a decrease in classification time were observed. Lower limit for the node bound t was set so that the relation $t \geq k$ was satisfied, and above higher limits there was no change in the error rate with respect to an exact result.

For the k-d tree, all data sets demonstrated roughly similar behavior, showing that the node bound is indeed a promising approach. For instance, in the case of Ferrites data set, for $t = 9$ the time was reduced approximately 5-fold, with the absolute increase in the error rate of around 1.2% (see Figure 7).

The *best bin first* algorithm achieved mostly a slight improvement over the regular search procedure (e.g., for Banknotes data set demonstrated in Figure 8), and it turned out to be most effective for the Isolet data set with 617 dimensions (see Figure 9). Nonetheless, very optimistic results reported by Lowe [11] were not reproduced for data sets used in this article. The difference between classification time for BBF and the regular node bound approach was negligible, and thus the former is omitted.

As regards the ball tree, the error rate behaved rather strangely. For Banknotes and Iris data sets (presented in Figure 10 and Figure 11, respectively), we can see unexpected spikes in the error rate, although there remained a steady decrease in classification time.

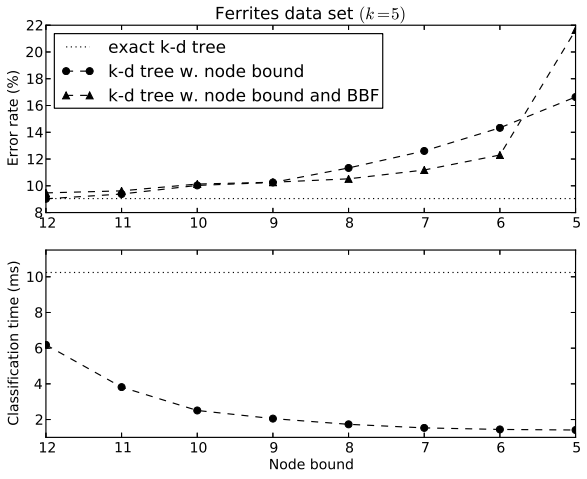

Fig. 7. Error rate and classification time vs node bound for the k-d tree with Ferrites data set, with and without BBF.

D. Probabilistic pruning

Since the pruning probability (σ) is equal for all subtrees, it might be the case that the pruned subtree contains one node, just as well as it might be the half of the entire tree. For this

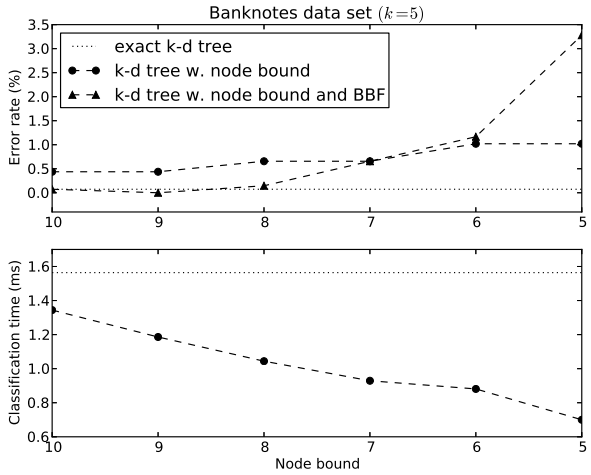

Fig. 8. Error rate and classification time vs node bound for the k-d tree with Banknotes data set, with and without BBF.

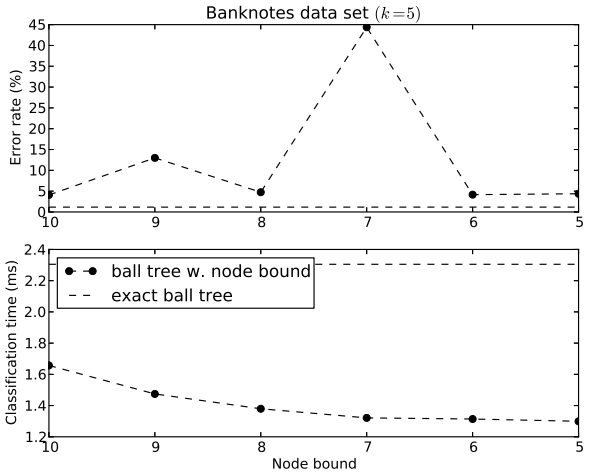

Fig. 10. Error rate and classification time vs node bound for the ball tree with Banknotes data set.

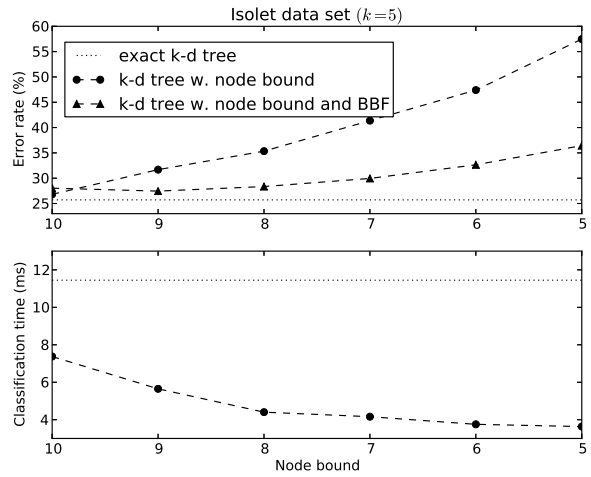

Fig. 9. Error rate and classification time vs node bound for the k-d tree with Isolet data set, with and without BBF.

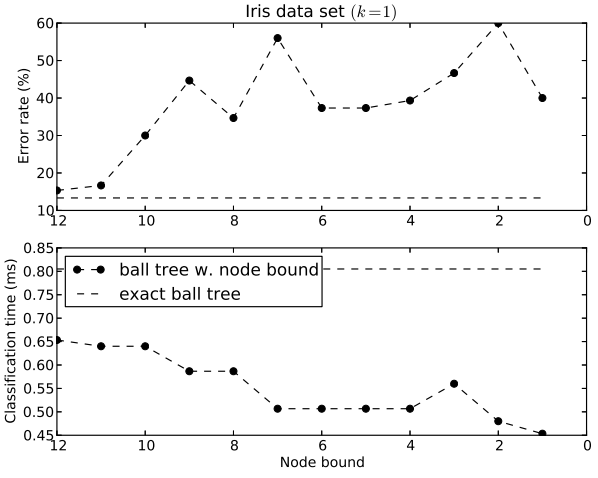

Fig. 11. Error rate and classification time vs node bound for the ball tree with Iris data set.

reason, effectiveness of this approach is clearly subject to a substantial amount of chance. Nevertheless, similar tendencies were observed for all data sets.

For the k-d tree, the ratio of error rate increase to classification time decrease tended to be less favorable than in the case of bounds presented in previous subsections — compare Figure 5 with Figure 13 to see how depth bound performed better for Banknotes data set. Still, for the Isolet data set (Figure 12), there was only a marginal increase in the error rate for $\sigma \leq 0.4$, and for these values the time was reduced by up to 35%.

Error rate increases for the ball tree were sharp (e.g., see Figure 14), and we observed once again unexpected results when the error rate for Banknotes data set actually decreased as more branches were pruned (see Figure 15). This can be most probably ascribed to simple luck resulting from the particular structure of this data set. Influence of the probabilistic nature of this algorithm was minimized by the fact that it was run five times.

VI. Conclusion

We have investigated two spatial data structures with identical applications, but different mechanics. The main difference between the k-d tree and the ball tree is the location of nodes associated with training data. The k-d tree does not consist of any redundant nodes, and each node corresponds to one instance from the training data. On the other hand, all training data in the ball tree are stored in the leaves, and internal nodes are utilized only in order to speed up the search procedure.

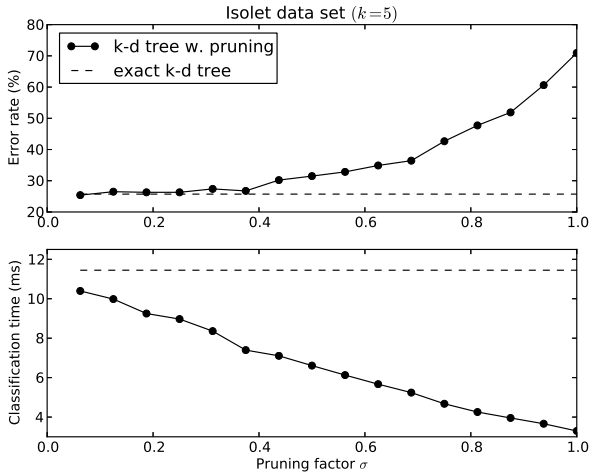

Fig. 12. Error rate and classification time vs pruning factor σ for the k-d tree with Isolet data set.

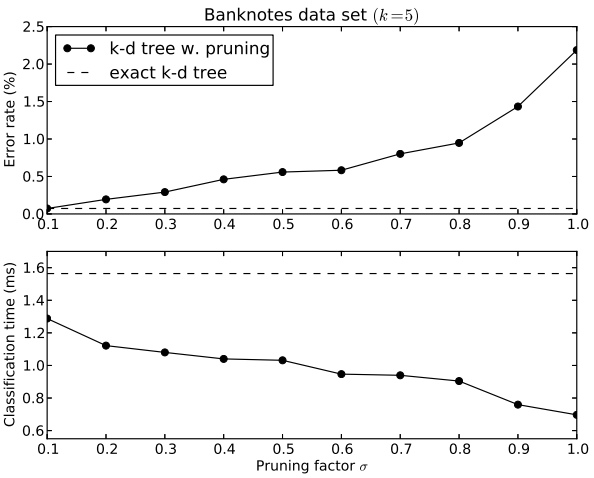

Fig. 13. Error rate and classification time vs pruning factor σ for the k-d tree with Banknotes data set.

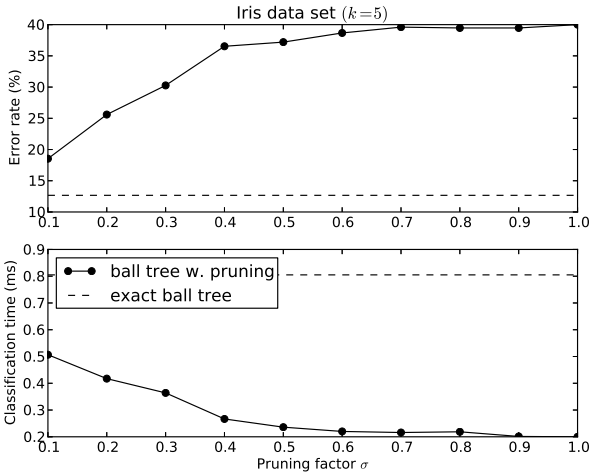

Fig. 14. Error rate and classification time vs pruning factor σ for the ball tree with Iris data set.

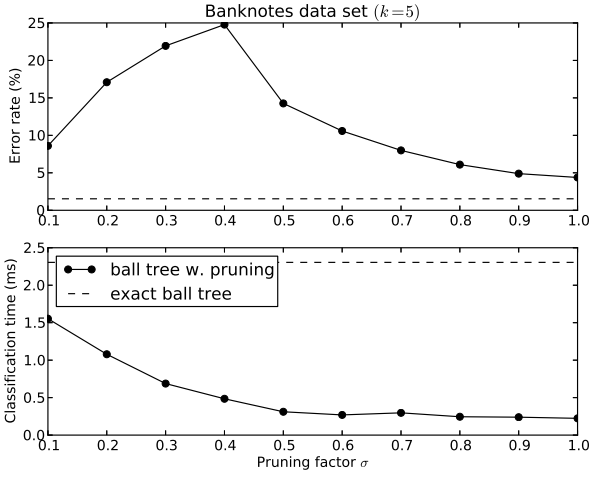

Fig. 15. Error rate and classification time vs pruning factor σ for the ball tree with Banknotes data set.

Approximate algorithms based on both trees turned out to perform rather well. The ratio of the increase in the error rate to the decrease in classification time was favorable, and the best results were reported for the k-d tree with node bound and best bin first (BBF) priority search. In the case of the ball tree, increases in the error rate were more rapid and unpredictable, however, this can be partially explained by the use of 5-fold cross validation instead of the leave-one-out method. Overall, the k-d tree was faster than the ball tree for both exact and approximate variants, which is consistent with exact performance measures presented by Munaga and Jarugumalli [12], and Kibriya and Frank [13].

Results depended chiefly on the data set that was used. No particular relation between the structure or size of the input data and results was observed, and it can be concluded that empirical findings remain the most valuable indicator, in spite of general tendencies.

We conclude that approximate variants of the *k*-nearest neighbor classification rule are indeed a very promising approach, and they are often indispensable when it comes to real-world massive data sets. Other data structures, which are based on the notion of partitioning the feature space, could also be adapted to use aforementioned bounds (CPU time, depth, node) and probabilistic pruning.

APPENDIX A

We list the data sets that were used for obtaining experimental results, along with their properties: class count, attribute count, and instance count.

- Banknotes — 2, 4, 1372
- Ferrites — 8, 30, 5903
- Iris — 3, 4, 150
- Isolet — 26, 617, 1559

REFERENCES

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998. doi: 10.1145/293347.293348

[2] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998. doi: 10.1145/276698.276876 pp. 604–613.

[3] P. W. Jones, A. Osipov, and V. Rokhlin, "Randomized approximate nearest neighbors algorithm," *Proceedings of the National Academy of Sciences*, vol. 108, no. 38, pp. 15 679–15 686, 2011. doi: 10.1073/pnas.1107769108

[4] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975. doi: 10.1145/361002.361007

[5] S. Arya, D. M. Mount, and O. Narayan, "Accounting for boundary effects in nearest-neighbor searching," *Discrete & Computational Geometry*, vol. 16, no. 2, pp. 155–176, 1996. doi: 10.1007/BF02716805

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. ISBN 9780262033848

[7] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[8] J. S. Beis and D. G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," in *CVPR*, 1997. doi: 10.1109/CVPR.1997.609451 pp. 1000–1006.

[9] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, 1987. doi: 10.1145/28869.28874

[10] A. Cisłak, "Approximate and probabilistic variants of the k-nearest neighbor classification rule," Bachelor's Thesis, Lodz University of Technology, 2014.

[11] D. G. Lowe, "Object recognition from local scale-invariant features," in *ICCV*, 1999. doi: 10.1109/ICCV.1999.790410 pp. 1150–1157.

[12] H. Munaga and V. Jarugumalli, "Performance evaluation: Ball-tree and kd-tree in the context of mst," *CoRR*, vol. abs/1210.6122, 2012. doi: 10.1007/978-3-642-32573-1_38

[13] A. M. Kibriya and E. Frank, "An empirical comparison of exact nearest neighbour algorithms," pp. 140–151, 2007. doi: 10.1007/978-3-540-74976-9_16