A comparison of k-nearest neighbour algorithms with performance results on speech data

Mike Matton and Ronald Cools

Report TW 381, January 2004

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A comparison of k-nearest neighbour algorithms with performance results on speech data

Mike Matton and Ronald Cools

Report TW 381, January 2004

Department of Computer Science, K.U.Leuven

Abstract

The (k-)nearest neighbour problem is well known in a wide range of areas. Many algorithms to tackle this problem suffer from the "curse of dimensionality" which means that the execution time grows exponentially with increasing dimension. Therefore, it is important to have efficient algorithms for the problem.

In this report, some well known tree-based algorithms for the k-nearest neighbour are investigated and tested on speech data. We experimentally derive the time complexity as a function of the number of nearest neighbours k, the database size n and the bucket size b.

Keywords : nearest neighbours search complexity approximate kd-tree bbd-tree

Contents

1 Introduction **1**

2 Basics **1**
2.1 Tree Structures . . . 1
2.1.1 Kd-trees . . . 2
2.1.2 Bbd-trees . . . 3
2.2 Search Algorithms . . . 4
2.2.1 The Standard Kd-tree Search . . . 4
2.2.2 Priority Search . . . 4

3 Experiments **5**
3.1 Exact vs Approximate k-nearest neighbours . . . 6
3.2 The Bucket Size . . . 6
3.3 Scale-up . . . 7
3.4 Kd-trees vs Bbd-trees and k-nearest neighbours . . . 7
3.5 Standard vs Priority Search . . . 8
3.6 Splitting Rules and the Cell Aspect Ratio . . . 9
3.7 Empirical complexity of the search . . . 9
3.7.1 Time complexity in the number of datapoints . . . 9
3.7.2 Time complexity in the number of requested nearest neighbours . . . 9

4 Future Work **11**

5 Conclusion **12**

1 Introduction

The (k-) nearest neighbour problem occurs in a wide range of areas, including pattern recognition, speech and image processing, data compression, databases, computer graphics etc .... In many cases this problem has to be tackled in high-dimensional spaces (ranging from two to a few hundred of dimensions). However, the amount of computing time necessary to solve the problem grows exponentially with increasing dimension. Therefore, high-performance search structures and algorithms have to be developed to reduce the necessary computation time. Two such search structures are kd-trees [5] and bbd-trees [2], which are compared in this report.

2 Basics

The Curse of Dimensionality Many (if not all) k-nearest neighbour algorithms suffer from the curse of dimensionality [4]. In terms of the k-nearest neighbour problem, the curse means that the number of data points that has to be investigated increases exponentially with the dimension of these points. One possible explanation for this is that the variances of the distances of uniform data decrease with increasing dimension (this means that the more dimensions you have, the closer the nearest and the farthest neighbour are to each other).

Exact vs Approximate Nearest Neighbours Several researchers have tried to reduce the complexity of the nearest neighbour search by relaxing the query. Instead of finding the exact k-nearest neighbours, the ϵ-k-nearest neighbours suffice. A point p is an ϵ-k'th nearest neighbour of a query point q if the ratio of the distances between p and q and between the real k'th nearest neighbour and q is not larger than $1+\epsilon$. The first researcher who used this idea was Arya [3]. Later, several others proposed algorithms for the approximate nearest neighbour problem.

2.1 Tree Structures

Many practical k-nearest neighbour search algorithms use tree structures to index the data. Among those structures are Samet's quad-trees [15], kd-trees originally proposed by Bentley [5] with its different variants [9, 14] and bbd-trees proposed by Arya [2]. Good overviews of different indexing structures based on trees are written by Gaede et al. [10] and Böhm et al. [6].

There are a lot of tree structures described in the literature. Many of them use hyperplanes to split the data. In this section, two kinds of these trees, those that were used in the experiments, are explained, namely kd-trees and bbd-trees. A short overview is given on how such a tree is created and the complexity of tree creation is mentioned. But first, the distance measures between two points that are used by these tree algorithms are explained.

Distance measures.

Friedman uses the notion of a dissimilarity measure $D(\mathbf{x}, \mathbf{y})$ to capture the "distance" between two points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$. He defines a dissimilarity measure as:

$$D(\mathbf{x}, \mathbf{y}) = F(\sum_{i=1}^{k} f_i(x_i, y_i)). \tag{1}$$

where f_i is the coordinate distance along coordinate i. F and f_i have to satisfy the following properties:

1. symmetry
$$f_i(x, y) = f_i(y, x), i = 1, \ldots, d \tag{2}$$

2. monotonicity
$$F(x) \geq F(y) \text{ if } x > y \tag{3}$$
$$f_i(x, z) \geq f_i(x, y) \text{ if } z \geq y \geq x \text{ or } x \geq y \geq z, i = 1, \ldots, d \tag{4}$$

When constructing a kd-tree, one can use a variety of distance measures. In theory, every measure that satisfies the symmetry (2) and monotonicity properties (3) (4) can be used to construct a kd-tree. The triangle inequality is not strictly needed. However, it is important that the distance function can be split along the coordinates because the kd-tree algorithms also uses “partial distances” along some of the coordinates. One of the most used set of distance measures is the set of vector space p-norms, also called Minkowski metrics, defined as

$$D_p(\mathbf{x}, \mathbf{y}) = \left[\sum_{i=1}^{k} |x_i - y_i|^p \right]^{1/p}. \tag{5}$$

Specific well known instances of this formula are the Euclidean distance ($p = 2$), the maximum distance ($p = \infty$) and the Manhattan distance ($p = 1$). In the experiments described in this report, the Euclidean distance is used.

2.1.1 Kd-trees

Kd-trees were invented by Bentley [5] and proposed as a tool for fast nearest neighbour queries. At each node of the tree, the data space is split with a hyperplane along one of the basic dimensions. Originally, as proposed by Bentley, the space is split along consecutive dimensions of the search space at each level of the tree. So, in the root node, the space is split along the first dimension, on the second level, the split occurs along the second dimension etc .... A few years later, Friedman proposed, together with Bentley, another way to choose the splitting direction. He used the direction with the maximal variance of point coordinates [9]. An example of a point cloud with its space partitioning and the corresponding kd-tree is shown in fig. 1.

Another parameter in kd-trees is the so called “bucket size”. This is the maximal number of data points that can be put in a leaf node of the kd-tree. The standard tree as proposed by Bentley uses a bucket size of 1, but there are valid arguments to increase this bucket size. One of the arguments is that elements that are in the same bucket have a high probability to be close to each other, so if one of the points in the bucket is a k-nearest neighbour, some of the other points in the bucket are it probably too. In this case, the standard kd-tree algorithm with bucket size 1 will probably cause more overhead than the computation of the distance from the query point to all the points in the bucket.

One of the disadvantages of the kd-tree is that buckets can become very narrow. The result of this is that, when searching for nearest neighbours, more nodes have to be examined, increasing the computing time. To overcome this, another criterion called "balanced aspect ratio" was introduced. However, for certain data distributions, the kd-tree algorithm fails to meet this criterion [8]. On the other hand, the splitting in kd-trees occurs along the median of the coordinate values in the split direction. It is easy to see that in this case, the balanced aspect ratio is not always guaranteed. To overcome this, some alternative splitting rules have been proposed.

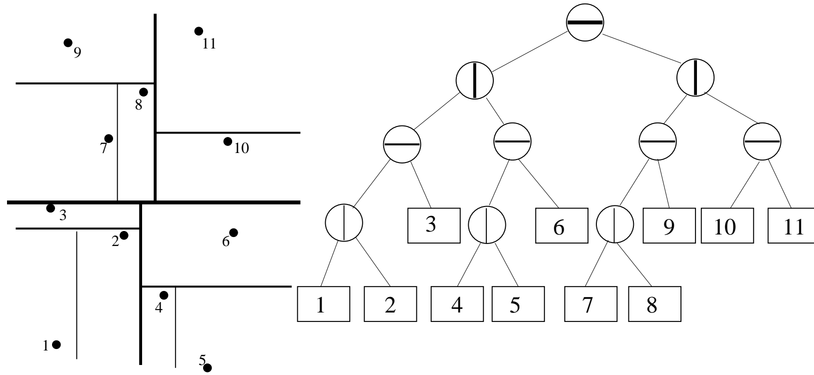

Figure 1: Point cloud and its corresponding kd-tree

Alternatives for the splitting rule. As mentioned above, the standard kd-tree algorithm uses a median splitting rule. It is easy to see that, for certain non-uniform data distributions, this leads to splits which violate the balanced aspect ratio criterion. Most alternative splitting rules guarantee the balanced aspect ratio, but to do this, they don't provide an equal distribution of the data over the tree any more. This leads to unbalanced trees. Some performance results on these splitting rules are shown in section 3.6 on p. 9.

Time complexity The expected time complexity of kd-tree construction is

$$O(dN \log N) \tag{6}$$

with d the dimension of the search space and N the number of data points. This is not that important because the tree creation has to occur only once. The complexity of the search algorithm is much more important.

2.1.2 Bbd-trees

Bbd-trees (balanced box decomposition) were introduced by Arya [2]. This type of trees uses two splitting criteria in the tree nodes. The first one is an iso-oriented hyperplane, similar to the ones used in kd-trees. The second one is a shrink operation which zooms in on the data (if the data is highly clustered). For this shrink-operation, different alternatives have been proposed by Ayra, details can be found in [2].

Another aspect of these trees is that they satisfy the "balanced aspect ratio" criterion. A consequence of this is that bbd-trees cannot guarantee that the points are evenly distributed and the resulting tree may not be balanced.

Time complexity The expected time complexity for bbd-tree time construction is equal to that of kd-tree construction, namely $O(dN \log N)$.

2.2 Search Algorithms

2.2.1 The Standard Kd-tree Search

An algorithm for searching the k nearest neighbours of a query point using optimized kd-trees was proposed by Friedman [9]. Summarized, this algorithm searches the kd-tree, starting with the node in which the query point is located until the hypersphere centered at the query point and with radius the distance to the k'th closest nearest neighbour found so far, fits entirely within the kd-node currently under investigation. In this case, all remaining points in the kd-tree are further away than that k'th closest nearest neighbour so these points don't need to be investigated.

Figure 2 shows an example of a kd-tree search for one nearest neighbour. The square-shaped point is the query point for which the nearest neighbour has to be found. The * indicates the currently investigated node and the highlighted point is the current point marked as nearest neighbour.

Complexity The complexity of the kd-tree search algorithm is not easily derivable. In his paper, Friedman derives that the search time is logarithmic in file size [9]. He also states that the number of records that is examined for the k nearest neighbours according to the ∞-distance in the ideal case is

$$\overline{R}_{\infty}(k, d) = (k^{1/d} + 1)^d, \tag{7}$$

which shows an exponential behaviour in the dimension (curse of dimensionality) and a sub-linear behaviour in the number of nearest neighbours. However, practical experiments show that the real behaviour of the kd-tree search algorithm is much worse (even super-linear). For an empirical derivation of the time complexity in the number of requested nearest neighbours, see section 3.7.2 on p. 9.

For the bbd-trees, Arya was not able to give bounds on the runtime other than a trivial one. His search algorithm also suffers from the curse of dimensionality: its complexity has exponential factors in the dimension of the search space too.

2.2.2 Priority Search

Priority search is a search technique that lists the nodes in the search tree in a somewhat more intelligent way compared to the standard kd-tree search. It visits the nodes in the tree in order of increasing distance to the query point q. This technique is described by Arya and Mount [3]. The technique can be implemented in an easy way using a priority queue. At each internal node in the search algorithm, the distance of the query point q to the farthest

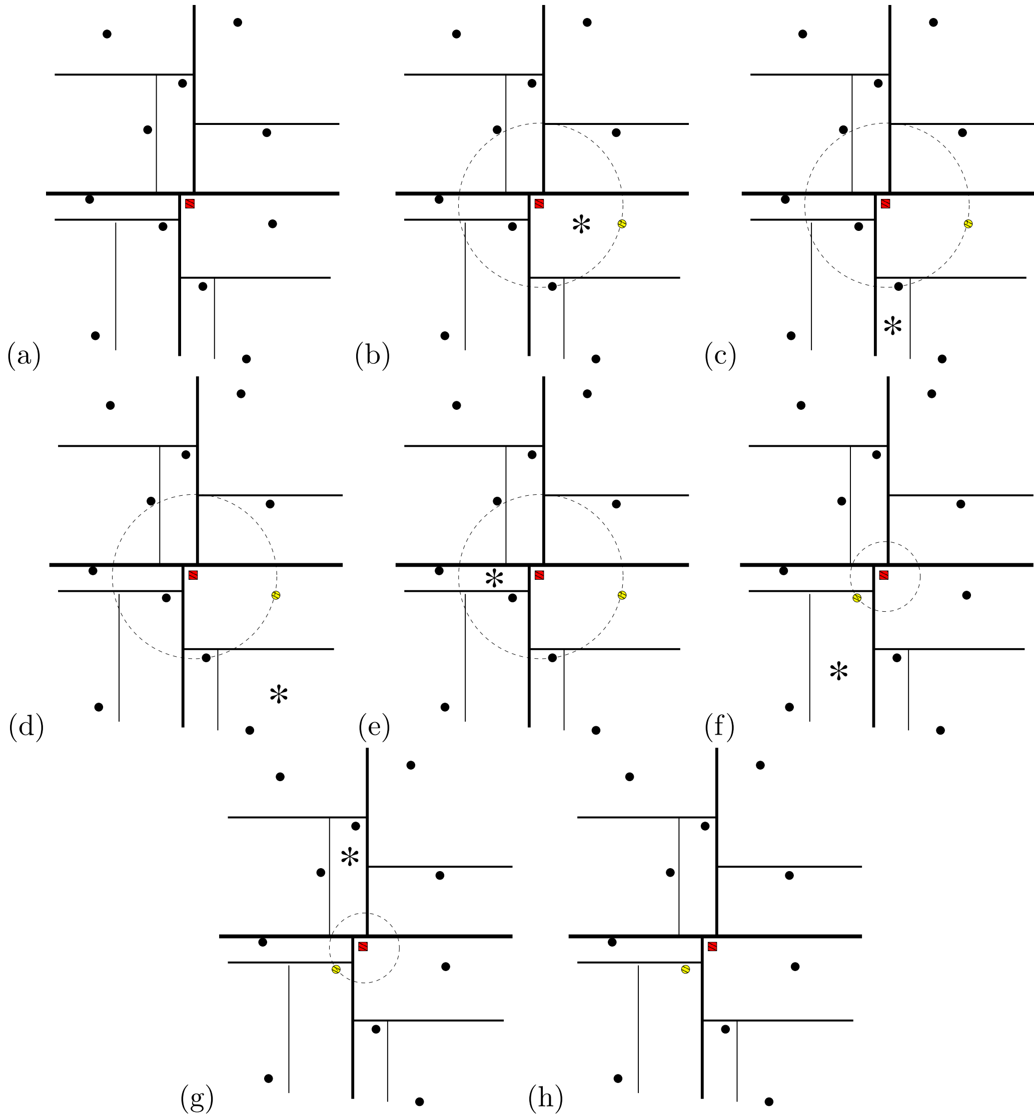

Figure 2: Kd-tree standard search algorithm

child node is inserted into the sorted priority queue. If the search arrives at a leaf node, the distances to the points in that leaf node are computed and then, if necessary, the next element in the priority queue is investigated by the search.

3 Experiments

To test the performance of these algorithms, the ANN (approximate nearest neighbours) toolbox of Arya [1] was used. The toolbox is used to compare kd-trees and bbd-trees, to experiment with different bucket sizes, to examine exact and approximate nearest neighbours, to compare standard search and priority search and to perform scale-up experiments. The database consists of a set of 15 dimensional feature vectors obtained from the TiDigits data set for speaker independent digit recognition [13].

3.1 Exact vs Approximate k-nearest neighbours

In a first experiment, it is checked in what way the introduction of the approximation factor ϵ improves the performance of the search. The measurements are summarized in fig. 3. It can be seen from fig. 3b that the relative gain in computing time decreases with increasing number of requested nearest neighbours. Further experiments point out that with an ϵ factor of 1, about 75-85% of the real nearest neighbours are part of the computed approximate nearest neighbours. With $\epsilon = 2$ this is about 50%. Details are given in table 1.

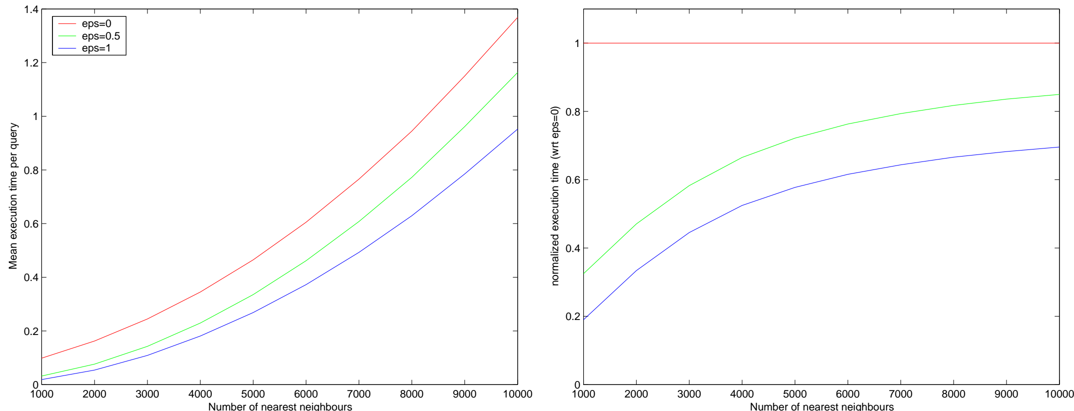

Figure 3: (a) searching 1000-10000 (approximate) nearest neighbours with a K-D-Tree in a 1516015 point TiDigits data set for varying ϵ. (b) same as (a) but with $\epsilon = 0$ execution times normalized to 1.

ϵ	100-nn	1000-nn	5000-nn	10000-nn
0	100%	100%	100%	100%
1	85.36%	81.07%	77.66%	75.95%
2	54.88%	51.17%	49.50%	49.12%

Table 1: Number of exact k-nearest neighbours computed as function of ϵ.

3.2 The Bucket Size

For determining the optimal bucket size, mean execution time per query is computed as a function of the bucket size. This relation is plotted in fig. 4 on p. 7. The behaviour shown in this figure will be typical for most databases. First, as the bucket size increases, the mean execution time will drop. The search algorithm will produce too much overhead when the bucket size is too low. At a certain point the curve will reach a minimum after which it will start to increase again because then the problem will converge to a brute force search (when the bucket size is bigger than the database size). The location of this minimum could be different for different problems. For this database, the optimal bucket size for computing 10000 nearest neighbours is around 20.

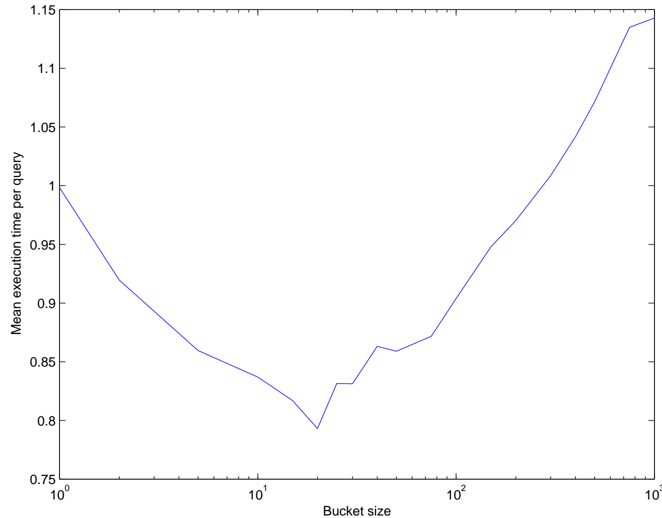

Figure 4: Query execution time as a function of the bucket size

3.3 Scale-up

In this experiment, some scale-up experiments were performed on the data set using kd-trees. According to Friedman, the complexity of the search should be logarithmic in file size N [9]. The measurements with the ANN toolbox are shown in fig. 5. The main problem with k nearest neighbours algorithms is the exponential factor in the dimension of the search space, which is a very large constant factor.

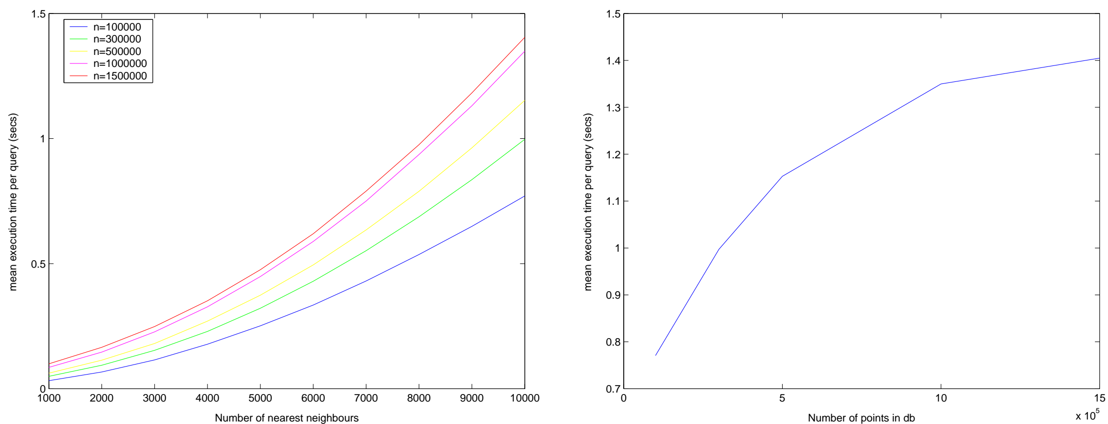

Figure 5: Scale-up experiment with kd-trees. (a) results for various file sizes and number of nearest neighbours. (b) results for 10000 nearest neighbours and varying file size

3.4 Kd-trees vs Bbd-trees and k-nearest neighbours

Fig. 6 shows a comparison between kd-trees and bbd-trees for the approximate nearest neighbours problem with $\epsilon = 1$. Strangely, the performance of the bbd-tree search is worse than the performance of the kd-tree search. A possible explanation for this is that the data

is not highly clustered and in that case the bbd-algorithm only produces overhead because the shrink operation will seldomly be used.

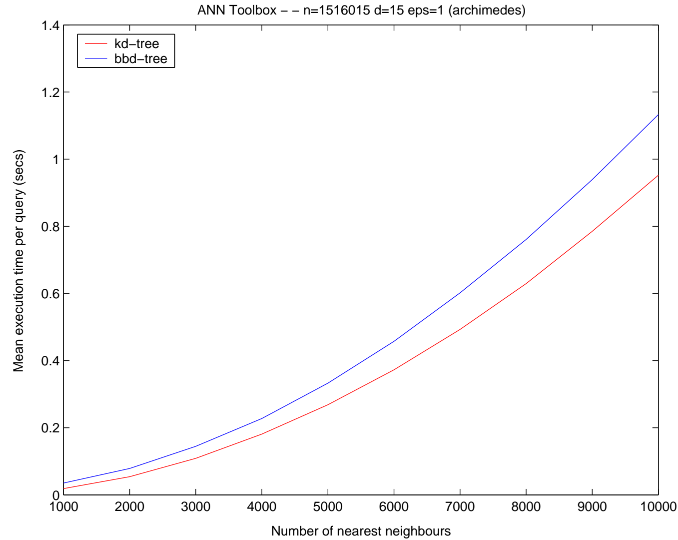

Figure 6: comparison searching 1000-10000 approximate nearest neighbours with kd-tree - bbd-tree in a 1516015 point TiDigits data set

3.5 Standard vs Priority Search

Another experiment compares the standard kd-search algorithm (section 2.2.1) and the priority search algorithm (section 2.2.2). The results of this comparison are displayed in fig. 7. As can be seen, the introduction of the priority search technique improves the performance of the search drastically for kd-trees as for bbd-trees. The behaviour it now shows is almost linear (only a slight curve) for the first 10000 nearest neighbours. As fig. 7.c shows, the execution time of the search is quadratic in the number of nearest neighbours.

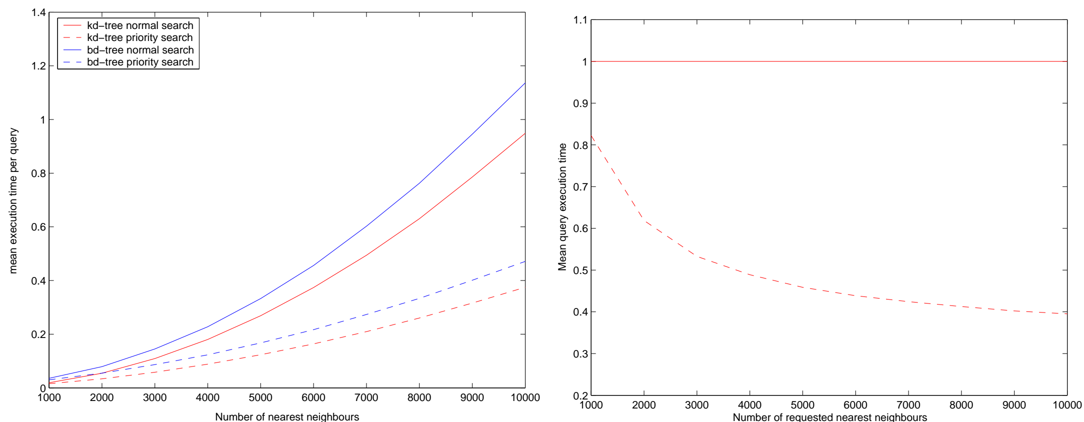

Figure 7: Comparison between standard kd-tree search and priority search (a) real execution times. (b) execution times for kd-tree normal search is normalized to 1.

3.6 Splitting Rules and the Cell Aspect Ratio

Experiments with the standard kd-tree split rule have shown that the mean aspect ratio of the cells is quite large (over 33). The aspect ratio is the ratio between the lengths of the longest and the shortest boundaries of the cell. If this ratio is large this means that the cell is quite "skinny". One can intuitively see that this degrades performance of the search. In this experiment another splitting rule is chosen (sliding midpoint splitting) which guarantees a lower aspect ratio. Some results are shown in fig. 8. The restriction of balanced aspect ratio provides only minor improvements to the search performance.

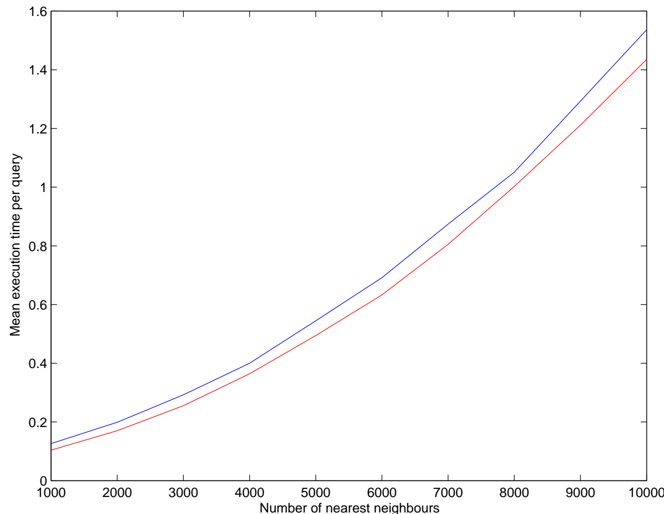

Figure 8: comparison splitting rules. blue = standard split, red = sliding midpoint split

3.7 Empirical complexity of the search

3.7.1 Time complexity in the number of datapoints

According to Friedman, the time complexity of the k nearest neighbour search with kd-trees is $O(\log n)$. This can be empirically verified by producing a logplot of the execution times of the scale-up experiment from section 3.3. If the complexity is indeed logarithmic, this plot should appear as a straight line. Such a logplot is shown in fig. 9 on p. 10. As expected, this plot indeed is approximately straight.

3.7.2 Time complexity in the number of requested nearest neighbours

The derivation of the time complexity as a function of the number of requested nearest neighbours is more difficult. Friedman, nor any researcher after him was able to give a time complexity as a function of the number of requested nearest neighbours. For ideal cases and the maximum distance, the complexity could be approximated using eq. 7 on p. 4. According to this equation, the time complexity should be sublinear in k. In practice this is not the case as can be derived from several plots in this report. Intuitively (from the plots) one would expect this complexity to be $O(k^2)$ or $O(k \log k)$.

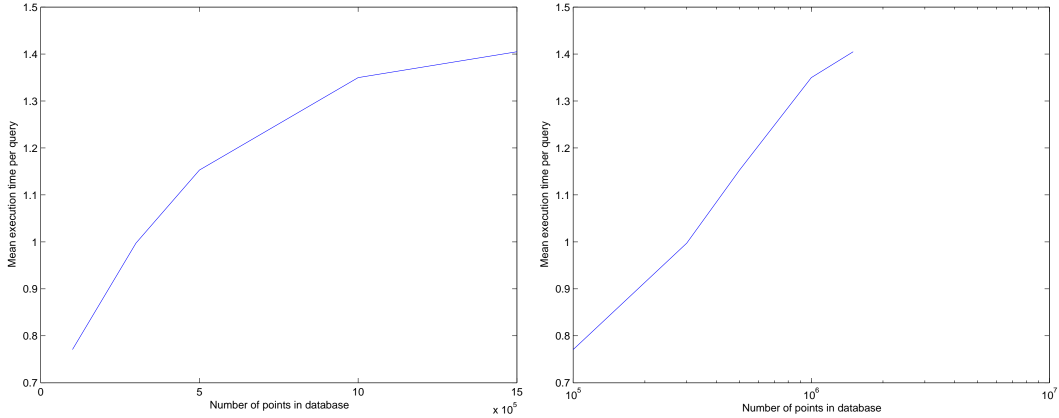

Figure 9: (a) mean execution time as function of number of requested nearest neighbours. (b) logplot of (a)

If the complexity is $O(k \log k)$ then a logplot of the function $t(k)/k$ should be a straight line for k large enough. Such a plot is shown in fig. 10 on p. 10. This is not at all a straight line, rejecting the hypothesis that the complexity is $O(k \log k)$.

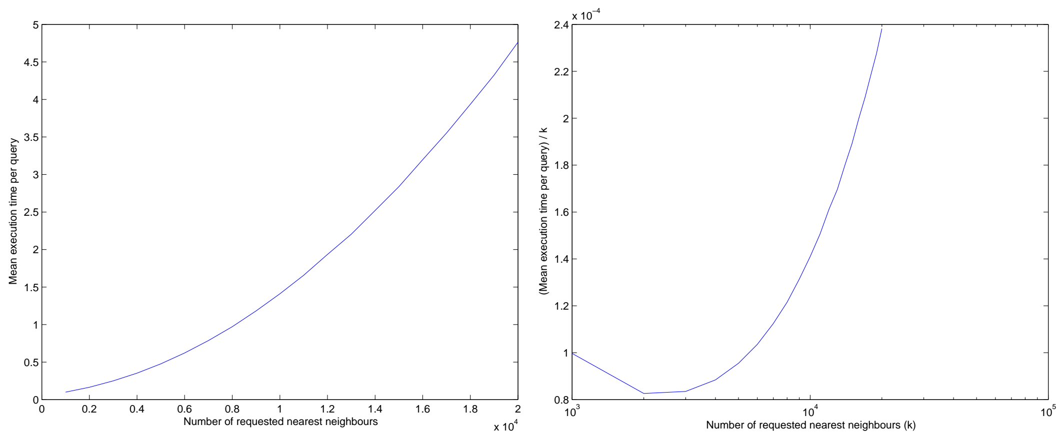

Figure 10: (a) Mean execution times for increasing k (b) Logplot showing k on the x-axis and $t(k)/k$ on the y-axis

Another possibility is that the complexity is polynomial in k, most probably quadratic. To check this, several polynomials of increasing order are estimated using the least squares criterion. The variance of the error is a good statistic to determine the optimal degree of the polynomial. The error variance for polynomials of increasing degree is shown in table 2 on p. 11, the polynomials in this table were made monic (a part from the sign of the highest order coefficient). From degree 2 on, this variance is small and it is smallest for a degree of 3. However, a closer look at the coefficiënts of the third order polynomial shows a highest order coefficiënt that is negative. This is very unlikely and therefore the second order approximation is preferred. This would mean that the time complexity in k is indeed quadratic and there is even more evidence for this: if we look at the plot of $t(k)/k$ on a standard scale shown in

fig. 11, a straight line can be seen (for big values of k). From all this, we can conclude that the time complexity of the search as function of the number of requested nearest neighbours is very likely to be $O(k^2)$

k	σ^2 (variance)	S (error)	polynomial
1	$2.1731 \cdot 10^{-1}$	$1.74 \cdot 10^{0}$	$k - 2.92$
2	$1.0918 \cdot 10^{-4}$	$7.64 \cdot 10^{-4}$	$k^2 + 3.70 \cdot k + 4.83$
3	$7.6782 \cdot 10^{-5}$	$4.61 \cdot 10^{-4}$	$-k^3 + 411 \cdot k^2 + 1138 \cdot k + 2362$
4	$9.2097 \cdot 10^{-5}$	$4.60 \cdot 10^{-4}$	$-k^4 - 147 \cdot k^3 + 7717 \cdot k^2 + 217425 \cdot k + 443432$
5	$9.5336 \cdot 10^{-5}$	$3.81 \cdot 10^{-4}$	$k^5 + 52.7 \cdot k^4 + 957 \cdot k^3 + 11049 \cdot k^2 + 81020 \cdot k + 84956$
6	$1.2393 \cdot 10^{-4}$	$3.72 \cdot 10^{-4}$	...

Table 2: Error variance for increasing polynomial degree

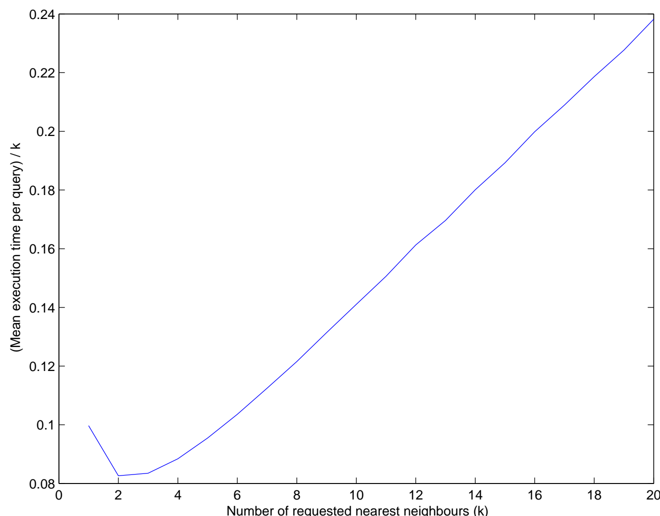

Figure 11: Plot showing k on the x-axis and $t(k)/k$ on the y-axis

4 Future Work

Many theoretical papers have been written about the (approximate) k nearest neighbours problem [7, 11, 12]. Many of them propose algorithms that can be used to solve the k-nearest neighbours problem. However, it is not always clear how to implement these theoretical algorithms. Further work on this topic could be done by trying to implement these theoretical algorithms and compare them with existing practical implementations of k-nearest neighbours search algorithms.

It is also possible to check for this TiDigits case study (or for speech data in general) if there aren't any "regularities" in the data that can be exploited with a problem specific algorithm. It is highly unlikely that a general algorithm for the k-nearest neighbours problem will overcome the curse of dimensionality, but when exploiting some regularities, the amount of computing

time needed could possibly be improved. However, it is not clear which regularities are easy exploitable and, more important, how they can be exploited. Some more work could be done on trying to identify such regularities in the data set (speech data in this case) and designing some ways to exploit these.

5 Conclusion

The k-nearest neighbour problem is a very time-consuming search problem and will probably remain this for quite a while. The main problem with k-nearest neighbours in high dimensions is the so-called "curse of dimensionality" that causes constant factors exponential in the dimension of the search space in the complexity of the search algorithms. Many attempts have been made to overcome this exponential dependence but none have really succeeded until now and the question is if it will in the near future. Even a relaxation of the problem: ϵ-approximate nearest neighbours did not improve performance very much (the exponential factor is still there).

On the other hand, the introduction of priority search has a significant impact on the performance of the search, at least for the database used in this case study. The cell aspect ratio seems a less important feature when comparing performance statistics.

References

[1] S. Arya. Ann: Library for approximate nearest neighbor searching v0.2, 1998.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, November 1998.

[3] Arya, S. and Mount, David M. Approximate Nearest Neighbor Queries in Fixed Dimensions. In *Proceedings of the 4th Annual ACM Symposium on Discrete Algorithms*, pages 271–280, Austin, January 1993.

[4] R. Bellman. *Adaptive Control Processes*. Princeton University Press, 1961.

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

[6] C. Böhm, S. Berchtold, and D.A. Keim. Searching in high-dimensional spaces - index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[7] Timothy M. Chan. Approximate nearest neighbours revisited. *Discrete and Computational Geometry*, 20(3):359–373, October 1998.

[8] A. Christian Duncan, Michael T. Goodrich, and Stephen Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–309, Baltimore, January 1999.

[9] J.H. Friedman, J.L. Bentley, and Finkel R.A. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[10] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[11] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, Dallas, May 1998.

[12] Jon M. Kleinberg. Two algorithms for nearest-neighbour search in high dimensions. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 599–608, El Paso, May 1997.

[13] Leonard, R. Gary. A Database for Speaker Independent Digit Recognition. In *Proceedings of the IEEE ICASSP'84*, pages 328–331, San Diego, March 1984.

[14] James McNames. A fast nearest-neighbour algorithm based on a principal axis search tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(9):964–976, September 2001.

[15] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.