A Comparison of Nearest Neighbor Search Algorithms for Generic Object Recognition

Ferid Bajramovic[1], Frank Mattern[1,⋆], Nicholas Butko[2], and Joachim Denzler[1]

[1] Chair for Computer Vision, Friedrich-Schiller-University Jena
{bajramov, mattern, denzler}@informatik.uni-jena.de
http://www4.informatik.uni-jena.de
[2] Department of Cognitive Science, University of California at San Diego
nbutko@cogsci.ucsd.edu
http://mplab.ucsd.edu

Abstract. The nearest neighbor (NN) classifier is well suited for generic object recognition. However, it requires storing the complete training data, and classification time is linear in the amount of data. There are several approaches to improve runtime and/or memory requirements of nearest neighbor methods: Thinning methods select and store only part of the training data for the classifier. Efficient query structures reduce query times. In this paper, we present an experimental comparison and analysis of such methods using the ETH-80 database. We evaluate the following algorithms. Thinning: condensed nearest neighbor, reduced nearest neighbor, Baram's algorithm, the Baram-RNN hybrid algorithm, Gabriel and GSASH thinning. Query structures: kd-tree and approximate nearest neighbor. For the first four thinning algorithms, we also present an extension to k-NN which allows tuning the trade-off between data reduction and classifier degradation. The experiments show that most of the above methods are well suited for generic object recognition.

1 Introduction

As shown in [1], the nearest neighbor classifier works well for generic object recognition. However, a naive implementation requires storing the complete training set, and classification takes time proportional to the size of the training data times the dimension of the feature vectors. Both aspects can be improved: efficient query structures greatly reduce classification time and thinning methods reduce the amount of data which has to be stored for the classifier. In this paper, we evaluate the performance of several such methods: for classification, we use kd-trees and approximate nearest neighbor (ANN), and for thinning, condensed nearest neighbor (CNN), reduced nearest neighbor (RNN), Baram's algorithm, Baram-RNN hybrid algorithm, Gabriel and GSASH thinning. For CNN, RNN, Baram and Baram-RNN, we propose and evaluate an extension to k nearest neighbors, which allows tuning the extent of thinning and thus the trade-off between data reduction and degradation of classification rates.

⋆ This work was financially supported by the German Science Foundation (DFG), grant no. DE 732/2-1.

J. Blanc-Talon et al. (Eds.): ACIVS 2006, LNCS 4179, pp. 1186–1197, 2006.

The remainder of the paper is organized as follows: In section 2 we give a short repetition of the k nearest neighbor classifier. Sections 3 and 4 describe the efficient query structures and the thinning algorithms respectively. In section 5 we present our experimental results. Section 6 gives final conclusions.

2 Nearest Neighbor Classifier

The k nearest neighbors (k-NN) classifier requires a labeled training data set $\{\mathcal{X}, \mathcal{Y}\} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$ consisting of d dimensional feature vectors $\boldsymbol{x}_i$ and their class labels y_i. For $k = 1$, in order to classify a new feature vector $\boldsymbol{x}$, find the closest element $\boldsymbol{x}_i$ in $\mathcal{X}$ and assign the label y_i to $\boldsymbol{x}$. The misclassification error of the 1-NN classifier converges (for $n \to \infty$) to at most twice the Bayes-optimal error [2].

For $k > 1$, find the k nearest neighbors $(\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_k})$ of $\boldsymbol{x}$ in $\mathcal{X}$. Then perform a voting amongst the class labels $(y_{i_1}, \ldots, y_{i_k})$ of these neighbors. The classic rule is to choose the class with the most votes within the set of neighbors, breaking ties arbitrarily. For $k > 1$, the asymptotic ($n \to \infty$) misclassification error of the k-NN classifier is as low as the Bayes-optimal error [2]. The voting can be modified to include a rejection rule. There are several possibilities: reject ties, reject if majority is too small, reject if not all neighbors are in the same class (unanimous voting). In general, the stricter the voting rule is, the more rejections there will be, but also the lower the misclassification rate will be.

3 Efficient Query Structures

There are several approaches to improve the running time of brute force nearest neighbor search [3,4,5]. In higher dimensions, however, these algorithms have an exponentially growing space requirement. Besides the small asymptotic improvement in time which was achieved by Yao and Yao [6] there exists no exact algorithm which can improve both time and space requirements in the worst case.

3.1 kd-Tree

The practically most relevant approach known for higher dimensions is the kd-tree introduced by Friedman, Bentley and Finkel [7]. The idea of the kd-tree is to partition the space using hyperplanes orthogonal to the coordinate axes. Each leaf node contains a bucket with a number of vectors, the other nodes in the binary kd-tree consist of a splitting dimension d and a splitting value v.

A query only has to look at one dimension of the query point at each node to decide into which subtree to descend. After the closest vector $\boldsymbol{x}$ in the bucket is found, one also has to search all buckets which are closer to the query vector than $\boldsymbol{x}$. In order to keep the tree small and to avoid searching in many buckets, one can stop splitting the tree if the bucket has a reasonably small size and search in the bucket linearly. It is shown in section 5 that this can reduce query times.

If the data is organized in a balanced binary tree, running time in the expected case is logarithmic. Unfortunately, the running time depends on the distribution of the training

data. In the worst case, the running time is linear. To improve the running time, several splitting rules where defined by [8].

The *standard kd-tree splitting rule* chooses the dimension as splitting dimension in which the data $\mathcal{X}$ have maximum spread. The splitting threshold is the median of the coordinates of $\mathcal{X}$ along this dimension. The depths of the tree is ensured to be $\lceil log_2 n \rceil$. But theoretically, the bucket cells can have arbitrarily high aspect ratio.

The *midpoint splitting rule* guarantees cells with bounded aspect ratio. It cuts the cells through the mean of its longest side breaking ties by choosing the dimension with maximum spread. *Trivial splits*, where all vectors of $\mathcal{X}$ lie on one side of the splitting plane, can occur and possibly cause the tree to have a larger depth than n.

The *sliding-midpoint splitting rule* is defined as the midpoint splitting rule, but omits trivial splits by replacing such a split with a split which contains at least one vector on each side. This is achieved by moving the splitting plane from the actual position up to the first vector of the dataset. This ensures that the maximum possible tree depth is n.

The *fair-split rule* is a compromise between the standard and midpoint splitting rules. The splitting plane is chosen from the possible coordinates in which a midpoint split can be done that does not exceed a certain aspect ratio of longest to shortest side. Among these, the coordinate with the maximum spread is chosen. The two extreme splitting planes which fullfill the aspect ratio will be compared with the median of the coordinates. If the median is on the smaller side, the cut will be done. Otherwise, a cut will be done at the median. Again, trivial splits can cause the tree depth to exceed n.

The *sliding fair-split rule* works as the fair-split rule but omits empty buckets by considering the extreme cut which just does not exceed a certain aspect ratio and which is closer to the median if the median does not fulfill the aspect ratio criterion. If this extreme cut is a trivial one, it is moved up to the position such that one vector lies on the other side. Again, this ensures that the maximum depth of the tree is n.

3.2 kd-Tree for Approximate Nearest Neighbor

Applying NN classification to generic object recognition, it is not important to really find the nearest neighbor. The classification is correct if a datapoint of the same class is found. So we consider doing generic object recognition with an approximate nearest neighbor approach developed by Arya and Mount [9]. A $(1+\epsilon)$ approximate nearest neighbor is defined as follows:

Definition 1. *A vector $\boldsymbol{q}$ is called* $(1+\epsilon)$ approximate nearest neighbor *of $\boldsymbol{x} \in \mathcal{X}$ if for all $\boldsymbol{y} \in \mathcal{X}$: $d(\boldsymbol{x}, \boldsymbol{q}) \leq (1+\epsilon) d(\boldsymbol{y}, \boldsymbol{q})$.*

The value ϵ is also called the error bound. If $\epsilon = 0$, the query is equivalent to the exact nearest neighbor classification. Otherwise, the minimum distance to the real nearest neighbor is at least $1/(1+\epsilon)$ of the found distance.

To find a given query vector $\boldsymbol{q}$, the leaf cell in the tree is found by descending the tree. Only those neighboring cells which are in the range of $d(\boldsymbol{x}, \boldsymbol{q})/(1+\epsilon)$ are searched for a closer training vector. Arya [9,10] has shown that the algorithm has polylogarithmic query time and needs nearly linear space which can be made quite independent of the vector distribution.

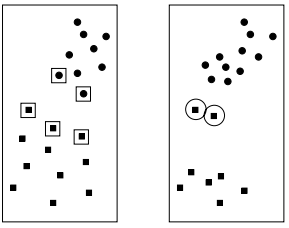

Fig. 1. Both images show the training data of two class problems. In the left image, the subset indicated by the boxes is a 2-consistent subset. The set in the right image is 2-consistent, but not 3-consistent, because the vectors indicated by the circles are 3-inconsistent.

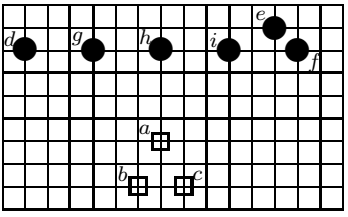

Fig. 2. The image shows a 3-consistent two class training set, which can be thinned by a 3-NN condensed nearest neighbor thinning to a 3-inconsistent set. The example uses the manhatten distance. If the vectors are visited in the order $a, b, c, d, e, f, g, h, i$, all vectors except for i will be added to the thinned set. In this set, h is 3-inconsistent.

4 Thinning

Thinning means reducing the training data set $\{\mathcal{X}, \mathcal{Y}\}$ to a smaller subset $\{\mathcal{X}', \mathcal{Y}'\}$. The classifier then only uses $\{\mathcal{X}', \mathcal{Y}'\}$. This results in reduced memory requirements and query times. There is an important property of thinned data sets $\{\mathcal{X}', \mathcal{Y}'\}$ [2]:

Definition 2. *A set $\{\mathcal{X}', \mathcal{Y}'\} \subseteq \{\mathcal{X}, \mathcal{Y}\}$ is called* consistent subset of $\{\mathcal{X}, \mathcal{Y}\}$ *if the 1-NN classifier for $\{\mathcal{X}', \mathcal{Y}'\}$ correctly classifies all members of the original set $\{\mathcal{X}, \mathcal{Y}\}$.*

This property is very desireable, as it guarantees perfect recognition of the 1-NN classifier for $\{\mathcal{X}', \mathcal{Y}'\}$ applied to the whole training set $\{\mathcal{X}, \mathcal{Y}\}$. We extend the definition with respect to the k-NN classifier:

Definition 3. *A vector $\boldsymbol{x} \in \mathcal{X}$ is called* k-consistent with respect to $\{\mathcal{X}, \mathcal{Y}\}$ *if the unanimous k-NN classifier for $\{\mathcal{X}, \mathcal{Y}\}$ classifies it correctly. Otherwise it is called* k-inconsistent with respect to $\{\mathcal{X}, \mathcal{Y}\}$. *A set $\{\mathcal{X}, \mathcal{Y}\}$ is called* k-consistent set *if it has no elements which are k-inconsistent with respect to $\{\mathcal{X}, \mathcal{Y}\}$. A subset $\{\mathcal{X}', \mathcal{Y}'\} \subseteq \{\mathcal{X}, \mathcal{Y}\}$ is called* k-consistent subset of $\{\mathcal{X}, \mathcal{Y}\}$ *if all members of $\{\mathcal{X}, \mathcal{Y}\}$ are k-consistent with respect to $\{\mathcal{X}', \mathcal{Y}'\}$.*

Clearly, the terms consistent subset and 1-consistent subset are equivalent. As for the 1-NN case, the property k-consistent subset guarantees perfect recognition of the k-NN classifier for $\{\mathcal{X}', \mathcal{Y}'\}$ applied to the whole training set $\{\mathcal{X}, \mathcal{Y}\}$. Fig. 1 shows an example of a 2-consistent subset for a given training set. Next, we proof three theorems:

Theorem 1. *A vector which is k-consistent with respect to a set is also k'-consistent with respect to the same set, for all $k' \leq k$.*

Proof: The k nearest neighbors of a labeled vector $(\boldsymbol{x}, c)$, which is k-consistent with respect to $\{\mathcal{X}, \mathcal{Y}\}$, are all in class c. Thus, also its k' nearest neighbors are in class c. Thus, $(\boldsymbol{x}, c)$ is k'-consistent with respect to $\{\mathcal{X}, \mathcal{Y}\}$. □

Input: $\{\mathcal{X}, \mathcal{Y}\}$		
Initialize R with one random element of $\{\mathcal{X}, \mathcal{Y}\}$		
FOR EACH $(\boldsymbol{x}, c) \in \{\mathcal{X}, \mathcal{Y}\} \setminus R$		
	IF	$\boldsymbol{x}$ is k'-inconsistent with respect to R
	THEN	Set $R = R \cup (\boldsymbol{x}, c)$
UNTIL R has not changed during the previous FOR EACH loop		
Result: $\{\mathcal{X}', \mathcal{Y}'\} = R$		

Fig. 3. Hart's thinning algorithm: condensed nearest neighbor

Theorem 2. *A k-consistent subset of a set is a k-consistent set.*

Proof: Given a k-consistent subset $\{\mathcal{X}', \mathcal{Y}'\}$ of $\{\mathcal{X}, \mathcal{Y}\}$, all elements of $\{\mathcal{X}, \mathcal{Y}\}$ are k-consistent with respect to $\{\mathcal{X}', \mathcal{Y}'\}$. As $\{\mathcal{X}', \mathcal{Y}'\} \subseteq \{\mathcal{X}, \mathcal{Y}\}$, all elements of $\{\mathcal{X}', \mathcal{Y}'\}$ are k-consistent with respect to $\{\mathcal{X}', \mathcal{Y}'\}$. Thus, $\{\mathcal{X}', \mathcal{Y}'\}$ is a k-consistent set. □

Theorem 3. *A k-consistent subset of a set is a k'-consistent subset of the same set, for all $k' \leq k$.*

Proof: Let $\{\mathcal{X}', \mathcal{Y}'\}$ be a k-consistent subset of $\{\mathcal{X}, \mathcal{Y}\}$. All labeled vectors in $\{\mathcal{X}, \mathcal{Y}\}$ are by definition k-consistent with respect to $\{\mathcal{X}', \mathcal{Y}'\}$ and thus k'-consistent with respect to $\{\mathcal{X}', \mathcal{Y}'\}$ (theorem 1). Thus, $\{\mathcal{X}', \mathcal{Y}'\}$ is a k'-consistent subset of $\{\mathcal{X}, \mathcal{Y}\}$. □

4.1 Condensed Nearest Neighbor

Hart [2,11] proposed a thinning algorithm called condensed nearest neighbor (CNN). First, one element of the training set is chosen arbitrarily. Then, a scan over all remaining elements is performed. During the scan, all elements which are 1-inconsistent with respect to the new growing set are added to the new set. Additional scans are performed until the new set does not change during a complete scan. The thinned subset is guaranteed to be a 1-consistent subset of the training set [2].

While Hart's algorithm reduces the size of the data and thus improves memory requirements and query times, it typically also reduces the recognition rate [2]. Depending on the structure of the training data and the application, the degradation of the classifier may be unacceptable. Hence, we propose an extension to the algorithm. The only change is that we require vectors to be k'-consistent instead of only 1-consistent. The complete algorithm is given in Fig. 3. The runtime of a naive implementation is $O((d + k')n^3)$ in the worst case.

While the thinned set is not guaranteed to be a k'-consistent set, as can be seen from the counter example in Fig. 2, it is obviously guaranteed to be a 1-consistent subset of the training set. It is quite obvious from theorem 3 in combination with the growing nature of the algorithm that in general, a greater value of parameter k' will result in a greater thinned set. The second part of our proposal is to choose $k' \geq k$. This means that we use a greater (or equal) parameter k' for thinning than for the application of the k-NN classifier. On the one hand, this makes sense, because even for a k'-consistent training set, the thinned subset is not guaranteed to be k'-consistent, but with increasing k', the chances of the thinned subset being at least k-consistent increase. On the other

Input: training data $\{\mathcal{X}, \mathcal{Y}\}$ and thinned data $\{\mathcal{X}', \mathcal{Y}'\}$		
Set $R = \{\mathcal{X}', \mathcal{Y}'\}$		
FOR EACH $(\boldsymbol{x}, c) \in R$		
	IF	All $(\boldsymbol{x}', c') \in \{\mathcal{X}, \mathcal{Y}\}$ are k'-consistent with respect to $R \setminus \{(\boldsymbol{x}, c)\}$
	THEN	Set $R = R \setminus \{(\boldsymbol{x}, c)\}$
Result: $\{\mathcal{X}'', \mathcal{Y}''\} = R$		

Fig. 4. Postprocessing algorithm for reduced nearest neighbor

hand, while it is desireable to have a k-consistent set (or even better, a k-consistent subset of the training set), what is more important is the classification rate of the k-NN classifier for the thinned set on a separate test set. Thus, it makes perfect sense to choose $k' > 1$ for a 1-NN classifier, even though the thinned set is already guaranteed to be a 1-consistent subset of the training set for $k' = 1$. To summarize, the parameter k' can be used to tune the trade-off between data reduction and classifier degradation.

4.2 Reduced Nearest Neighbor

Gates [2,12] proposed a postprocessing step for the CNN thinning algorithm. As the initial members of the thinned set are chosen arbitrarily and as additional members are added, it may be possible to remove some vectors and still retain a 1-NN consistent subset of the training set. The postprocessing algorithm simply checks for each vector of the thinned set if the thinned set without that vector is still a 1-NN consistent subset of the training set. If it is, the vector is removed. Of course this algorithm can also be extended to a k'-NN version as described in the previous subsection. The postprocessing algorithm is given in Fig. 4. The runtime of a naive implementation is $O((d + k')n^3)$ in the worst case. The complete reduced nearest neighbor (RNN) thinning algorithm performs CNN thinning followed by the postprocessing algorithm. As the CNN part may produce a k'-inconsistent set and the postprocessing will not remove any k'-inconsistent vectors, the RNN thinning algorithm can also produce a k'-inconsistent set.

4.3 Baram's Method

Baram [2,13] proposed a thinning algorithm that thins each class individually. For each class, a new set for the thinned class is initialized with an arbitrary member of that class. Then, each vector of that class, which is 1-inconsistent with respect to a modified training set in which the current class is replaced by the growing thinned version of that class, is added. Naturally, also this algorithm can be extended to a k'-NN version. Fig. 5 shows the complete algorithm. The k'-NN version of Baram's algorithm can also produce a k'-inconsistent set, as the same counter example as for CNN applies (Fig. 2). The runtime of a naive implementation is $O((d + k')n^2)$ in the worst case. In an unpublished paper, Olorunleke [14] proposed combining Baram's algorithms with the postprocessing step of RNN and calls it Baram-RNN hybrid algorithm.

Input: $\{\mathcal{X}, \mathcal{Y}\}$		
FOR EACH class $c \in \mathcal{Y}$		
	Remove class c: $\{\mathcal{X}^*, \mathcal{Y}^*\} = \{\mathcal{X}, \mathcal{Y}\} \setminus \{\mathcal{X}_c, \mathcal{Y}_c\}$	
	Set $R_c = \emptyset$	
	FOR EACH vector $\boldsymbol{x} \in \mathcal{X}_c$	
		IF $(\boldsymbol{x}, c)$ is k'-inconsistent with respect to $\{\mathcal{X}^*, \mathcal{Y}^*\} \cup R_c$
		THEN Set $R_c = R_c \cup (\boldsymbol{x}, c)$
Result: $\{\mathcal{X}', \mathcal{Y}'\} = \bigcup_{c \in \mathcal{Y}} R_c$		

Fig. 5. Baram's thinning algorithm ($\{\mathcal{X}_c, \mathcal{Y}_c\} \subseteq \{\mathcal{X}, \mathcal{Y}\}$ contains the members of class c)

4.4 Proximity Graph Based Thinning

The thinning algorithms in the previous sections all exhibit the property that different thinned-sets will result from considering the datapoints in a different order. As this is undesirable, we also consider order-independent, graph-based thinning algorithms.

The starting place for these order-independent algorithms is the *Delaunay graph* [15], which is constructed by connecting nodes in adjacent Voronoi cells. A Voronoi cell is the region of space around a point that is closer to that point than to any other point. If we remove a point from our set, all points falling in its Voronoi cell will now fall in a cell belonging to one of its neighbors in the Delauny graph. This suggests a thinning algorithm: by removing all points that are surrounded by Delauny neighbors of the same class, we are left with a thinned set that has exactly the same classification properties as the original set in a 1-NN classification scheme.

Despite its desirable properties, Delaunay Graph thinning has two critical drawbacks: the algorithm is exponential in the dimensionality of the data, and empirically removes very few points for real datasets [15]. It seems that tolerating some shift in the decision boundary can (greatly) increase the number of points removed in thinning.

Two points $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathcal{X}$ are Gabriel neighbors if there is no third point $\boldsymbol{x}_3 \in \mathcal{X}$ inside the hypersphere centered on the midpoint of $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, and with diameter equal to the distance between them. Mathematically, we say that $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are Gabriel neighbors iff $\forall \boldsymbol{x}_3 \in \mathcal{X},\ d(\boldsymbol{x}_1, \boldsymbol{x}_2)^2 \leq d(\boldsymbol{x}_1, \boldsymbol{x}_3)^2 + d(\boldsymbol{x}_2, \boldsymbol{x}_3)^2$. A *Gabriel graph* is an undirected graph built by connecting each node to all of its Gabriel neighbors. As with Delaunay graphs, we will consider a thinning algorithm in which all points that are only neighbors with points of the same class are removed from the dataset. Since the Gabriel graph is a subset of the Delaunay graph, Gabriel thinning will remove all of the points that Delaunay thinning removes, and possibly more. This may change the decision boundary (and possibly even leads to a 1-inconsistent thinned subset), but in practice, Sánchez *et al.* found that Gabriel thinning leads to better classification (at the cost of keeping more points) than traditional CNN methods [16].

There is a quadratic cost to finding a given point's Gabriel neighbors. To build an entire graph so that we can do filtering, we incur this cost for every point in the data set. This means that building an exact Gabriel graph is cubic in the number of data points, and so is very costly. Using Mukherjee's *GSASH* data structure [17], the cost becomes $O(n \log_2 n)$, though with potentially large constants.

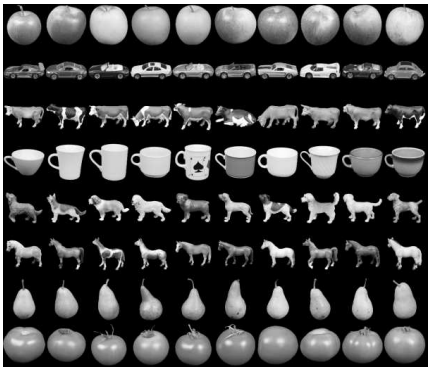

Fig. 6. Sample images of the objects of the ETH-80 database [18]

5 Results

5.1 Dataset

To evaluate the nearest neighbor classification for generic objects, we use the ETH-80 database [18], which contains 80 objects from 8 categories: `apple`, `car`, `cow`, `cup`, `dog`, `horse`, `pear` and `tomato` (see Fig. 6). Each object is represented by 41 images of views from the upper hemisphere. The experiments are performed using 128×128 pixel images, with each image cropped close to the object boundaries. The grayvalues of the image will be transformed to a feature vector by a PCA transformation with the eigenvectors of the 100 largest eigenvalues. The 100 dimensional feature vectors will be used for classification. The test is performed by cross-validation with a leave-one-object-out strategy. One of the 80 objects is used for testing and the 79 other objects are used to learn the PCA transformation and build the k-NN query structure. This "unknown" object must accordingly be classified into the correct object category.

5.2 Experiments

We examined the presented methods with respect to query time, error / rejection / recognition rate and used data size. In many applications, NN is not applied because it is too slow. This can be improved with the kd-tree. Arya [10] has shown the dependency on the splitting rule.

One other important parameter of the kd-tree is the bucket size. If it is too small, the tree becomes very large and the search for the bucket which has to be taken into consideration takes long. If the bucket size is too large, it takes too much time to search the bucket linearly. To get a fast kd-tree query, the optimal bucket size for the generic ETH-80 dataset should be medium size. In our example, bucket size 32 with the standard kd-tree splitting rule is the best choice. The splitting rule is not very important if the bucket size is chosen well. For this bucket size, query times vary by about 8% from 88.9μs with the standard splitting rule to 96.1μs with the midpoint splitting rule,

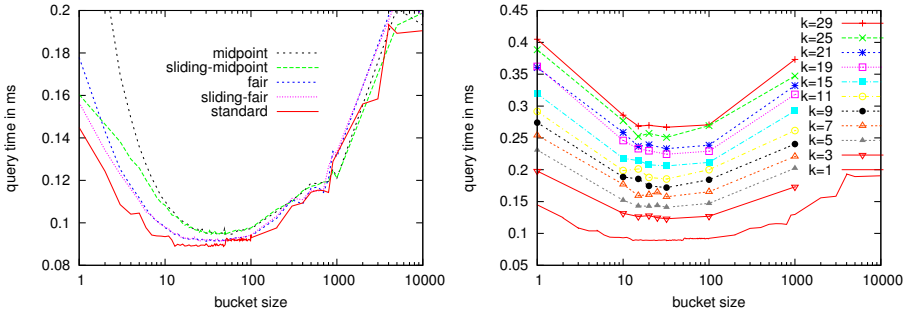

Fig. 7. Dependency of query time on a Intel Pentium 4 with 3.4GHz on a kd-tree with different bucket sizes using the generic ETH-80 dataset. Different splitting rules for $k = 1$ (left) and different parameters k for the k-NN classifier with standard splitting rule (right) are examined. Best query times are achieved with bucket size between 20 and 32.

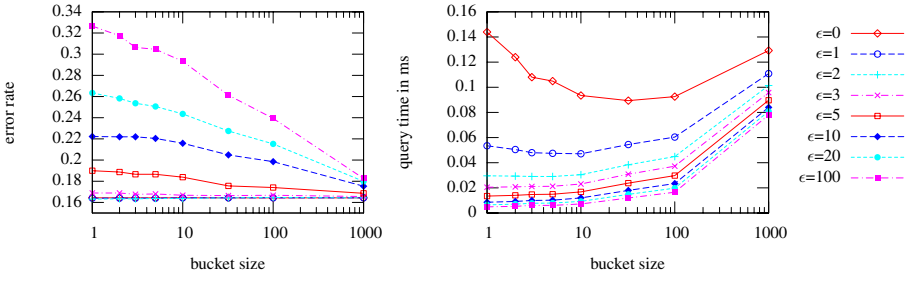

Fig. 8. Performance on different approximation error bounds ϵ. The left image shows that the error rate on smaller bucket sizes increases, but as the right image shows the query time can become amazingly fast. For $\epsilon = 100$ the query time is 5μs but causing an error rate of 32.7% with a 1-NN classifier on the generic ETH-80 dataset.

whereas with bucket size one, query times vary by a factor of about two. Using a larger parameter k for the k-NN classification, the query time increases, but the best query time is still attained at the same bucket sizes. So this is independent of k (see Fig. 7).

The query time can further be decreased by using approximate nearest neighbor classification. In general, the query time decreases with larger error bounds and also with lower bucket sizes if the error bound is large enough. In our experiments, the best query time (5μs) is obtained using $\epsilon = 100$ and bucket size one, but at the cost of a strong rise of the error rate to 32.7%. Useful values of ϵ are about 1–3 (see Fig. 8). Using an error bound $\epsilon = 2$, the query time can be improved by the factor of 3 to 29.0μs without losing any recognition rate in our 80 test sets and with $\epsilon = 3$ to 20.5μs with an increased error rate of 0.46 percentage points, which is quite acceptable.

Gabriel thinning reduces the data set only to 96.8%. The fastest and least precise GSASH approximation with one parent and one child reduces the data set to 93.8% and with 6 children and 6 parents to 94.6%. So the results are similar to those using the

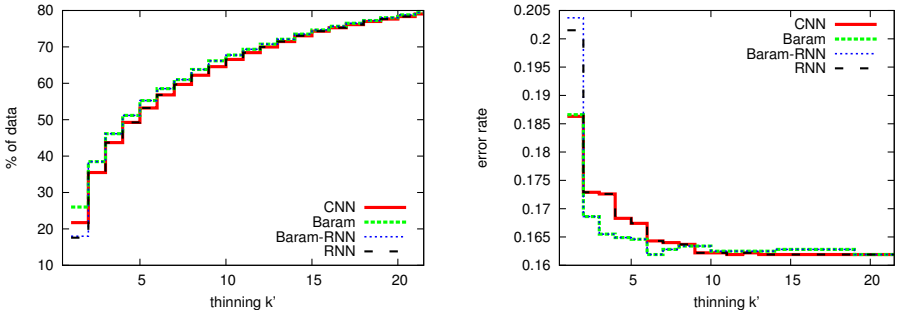

Fig. 9. In the images, the dependency of different thinning algorithms on the parameter k' is presented. The left image shows the proportion of the full dataset which remains after thinning. In the right image, the error rate using 1-NN on the reduced dataset is shown.

original data set, but can also improve the query time to error rate ratio (see Fig. 12). A major disadvantage of the approach is the time requirement for thinning. Gabriel thinning needs about 27 minutes for 3239 vectors and the fastest approximation about 11 minutes, whereas e.g. Baram or CNN need only about 2.5 seconds. The reason for the small reduction is an indication of the bad distribution of the data in the 100 dimensional space. A reduction of the dataset can still be done with CNN or Baram but at the cost of recognition rate. Our extension of the NN thinning algorithms can adjust the reduction of the dataset. This effect can be observed in Fig. 9 (left).

As expected, the greater the value of k' is, the more data is retained after thinning. Accordingly, as can be seen in Fig. 9 (right), the error rate decreases with increasing k'. Furthermore, Fig. 10 shows the influence of k' on the recognition and rejection rates for Baram and several values of k (unanimous voting). Independent of k and in accordance with Fig. 9, the recognition properties of the classifier improve with growing k'. This shows that the trade-off between data size and error rate can be tuned.

Considering Fig. 9 again, a comparison between the four thinning algorithms shows that for $k' > 1$, a) there is no difference between CNN and RNN as well as between Baram and Baram-RNN and b) Baram keeps a bit more data than CNN. While it is not suprising that hence the error rate of Baram is lower, it is actually lower than the sheer amount of data would suggest: Baram with $k' = 6$ keeps 58.5% of the data while CNN with $k' = 9$ retains 64.6%. The error rate is 16.2% in both cases. On this data set, for $1 < k' < 9$, Baram clearly outperforms CNN.

Varying parameter k for k-NN classification with unanimous voting lets us choose a specific error rate versus rejection rate ratio (see Fig. 11). If k becomes larger, the error rate decrease, but the rejection rate increase. Thinning with $k' < k$ does not make sense, because the k-NN rejection rate strongly increases, as Fig. 10 shows. Thinning with $k' = 1$ is, with respect to recognition rate, bad in general. The error rate versus rejection rate ratio is vitally better for k-NN trained with the full data set. Using Baram thinning with $k' = 6$ reaches the best possible ratio – even better than without thinning.

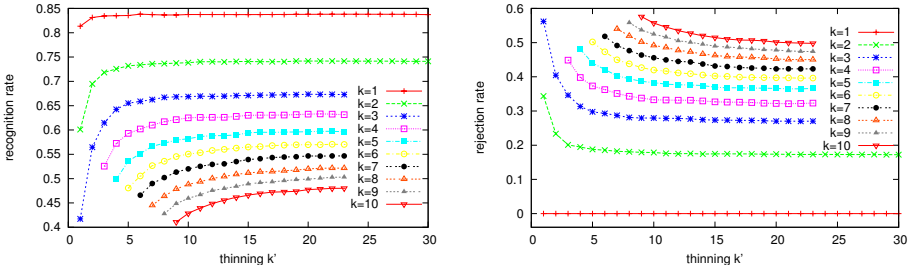

Fig. 10. Error versus rejection rate of Baram using different parameter k' for thinning and different parameter k for classification with unanimous voting

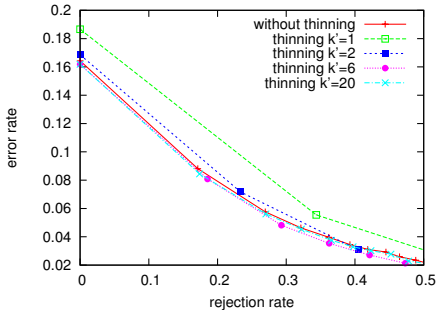

Fig. 11. Rejection versus error rate of k-NN classification (varying k) with unanimous voting on training data thinned by Baram using several parameters k'

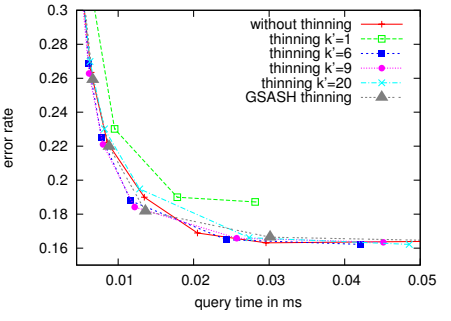

Fig. 12. Query time versus error rate of 1-ANN classification for different parameters ϵ and training data thinned by Baram with different parameters k' and also GSASH thinning

Using higher approximation or smaller training data for nearest neighbor classification leads to a higher error rate. Which ratio between query time and error should be chosen highly depends on the application. The best methods at a given query time form an optimal ratio curve. As shown in Fig. 12, 1-ANN classification trained with data thinned by Baram with $k' = 1$ is in general worse than 1-ANN trained with the original data. A smaller error rate with respect to a given query time can be obtained using Baram thinning with e.g. $k' = 9$. Using these Baram thinned data, which are reduced to 66.2% of the original data, the ANN classification with $\epsilon = 5$ attains a query time of 12.2μs with an error rate of 18.4%. For thinning parameter $k' > 9$, the methods lie on the ratio curve of the original data (see Fig. 12).

6 Conclusions

We showed that thinning methods and query structures for k-NN are well suited to reduce memory requirements and/or classification times for generic object recognition. The experiments showed that, for optimal speed of exact queries, the bucket size of the

kd-tree is important and independent of k. For ANN, a small bucket size and a large error bound ϵ yield the fastest queries. Furthermore, we developed k'-NN extensions of CNN, RNN and Baram and showed that they allow to tune the trade-off between data reduction and classifier degradation. As expected, the classical versions of the algorithms ($k' = 1$) yield maximum degradation. The best trade-off between query time and error rate was reached for a combination of k'-NN Baram and ANN. Gabriel and GSASH thinning turned out not to work well on the high-dimensional ETH-80 data.

References

1. Mattern, F., Denzler, J.: Comparison of appearance based methods for generic object recognition. Pattern Recognition and Image Analysis **14** (2004) 255–261
2. Toussaint, G.: Geometric proximity graphs for improving nearest neighbor methods in instance-based learning and data mining. Int. J. of Comp. Geom. & Appl. **15** (2005) 101–150
3. Clarkson, K.: A randomized algorithm for closest-point queries. SIAM Journal of Computing **17** (1988) 830–847
4. Dobkin, D., Lipton, R.: Multidimensional searching problems. SIAM Journal of Computing **2** (1976) 181–186
5. Meisner, S.: Point location in arrangements of hyperplanes. Information and Computation **2** (1993) 286–303
6. Yao, A., Yao, F.: A general approach to d-dimension geometric queries. In: 17th Symposium on Theory of Computing. (1985) 163–168
7. Friedman, J., Bentley, J., Finkel, R.: An algorithm for finding best matches in logarithmic expected time. ACM Transactions on Mathematical Software **3** (1977) 209–226
8. Maneewongvatana, S., Mount, D.: Analysis of approximate nearest neighbor searching with clustered point sets. In: The DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Volume 59. (2002) 105–123
9. Arya, S., Mount, D.: Approximate nearest neighbor queries in fixed dimensions. In: Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. (1993) 271–280
10. Arya, S., Mount, D., Netanyahu, N., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor searching. Journal of the ACM **45** (1998) 891–923
11. Hart, P.E.: The condensed nearest neighbor rule. IEEE Transactions on Information Theory **14** (1968) 515–516
12. Gates, W.: The reduced nearest neighbor rule. IEEE Transactions on Information Theory **18** (1972) 431–433
13. Baram, Y.: A geometric approach to consistent classification. Pattern Recognition **13** (2000) 177–184
14. Olorunleke, O.: Decision Rules for Classification: Classifying Cars into City-Cycle Miles per Gallon Groups. Dep. of Computer Science, University of Saskatchewan, Canada (2003)
15. Toussaint, G., Bhattacharya, B., Poulsen, R.: The application of voronoi diagrams to non-parametric decision rules. In: 16th Symp. on Comp. Science and Statistics. (1984) 97–108
16. Sánchez, J., Pla, F., Ferri, F.: Prototype selection for the nearest neighbor rule through proximity graphs. Pattern Recognition Letters **18** (1997) 507–513
17. Mukherjee, K.: Application of the Gabriel graph to instance based learning algorithms. PhD thesis, Simon Fraser University (2004)
18. Leibe, B., Schiele, B.: Analyzing Appearance and Contour Based Methods for Object Categorization. In: Int. Conf. on Comp. Vision and Pattern Recog. Volume 2. (2003) 409–415