Speeding up Relief algorithms with k-d trees

Corrected version (Table 1) of the paper published in Proceedings of the Electrotechnical and Computer Science

Conference ERK'98, Portorož, Slovenia, 1998

Marko Robnik Šikonja
Faculty of Computer and Information Science
University of Ljubljana
Tržaška 25, 1001 Ljubljana, Slovenia
Marko.Robnik@fri.uni-lj.si

Abstract

There are certain problems in machine learning which desire special attention when we scale up the size of the data or move towards data mining. One of them is the problem of searching nearest neighbours of a given point in k dimensional space. If the space is $\Re^k$ than k-d trees can solve the problem in asymptotically optimal time under certain conditions. We investigate the use of k-d trees in nearest neighbour search in the family of attribute estimation algorithms Relief on typical machine learning databases and examine their performance under various conditions.

1 Introduction

In recent years we have seen in machine learning an increasing demand for algorithms that can scale up to huge amounts of data in terms of number of learning examples as well as in the dimensionality of the data. Eventually a new scientific field emerged called data mining [4] which contains elements of machine learning, statistics and databases. In the scope of this "upsizing" we have investigated the possibilities to speed up the attribute estimation algorithms Relief [6], ReliefF [7], and RReliefF [11]. The main idea of increasing speed of these algorithms was to improve the nearest neighbour search which lies at the core of them.

If we have N records in the space $\Re^k$ k-d trees [1] based search algorithm [5] can find t nearest neighbours in asymptotically optimal time $O(logN)$ under certain conditions. In machine learning some of the conditions are typically not met. We are rarely dealing with continuous attributes only and the distribution of attribute values is not necessary normal. In most cases our problem is described with a mix of discrete and continuous attributes. Also it has been empirically indicated that the performance of k-d trees seriously degrades as the dimensionality of the data increases. This could be a serious problem in machine learning and especially in attribute estimation where we are often facing hundreds of attributes.

We investigate the use of k-d trees in attribute estimation algorithms Relief, ReliefF and RReliefF, but our conclusions are valid for pure nearest neighbour search (which is the basis of all instance-based (lazy) learning) as well, since the main computational burden of Relief algorithms is in the nearest neighbour search as we will show.

There are other uses of k-d trees in machine learning and pattern recognition. Nearest neighbour search algorithm on k-d trees can be slightly modified to allow incremental nearest neighbour search [2]. Multiresolution instance-based learning [3] uses k-d trees to structure datapoints into groups of similar weights. The same technique is applied to locally weighted polynomial regression [8].

In Section 2 we describe a k-d tree building algorithm and a nearest neighbour search algorithm based on k-d trees. In Section 3 we describe some problems with k-d trees in machine learning and show improvements in time complexity for Relief family of algorithms. In Section 4 we experiment with some typical machine learning datasets and measure performance of nearest neighbour search in Relief algorithms. Section 5 concludes and gives some guidelines of when to apply k-d trees.

2 K-d trees and NN search

The k-d tree [1, 13] is a generalization of the simple binary tree, which uses k keys instead of a single key. Let we suppose that elements are presented in a vector notation, each vector consisting of k dimensions (keys), $\mathbf{x} = (x_1, x_2, ..., x_k)$. Each interior node has two successor nodes and defines a split of the elements on one of the dimensions, e.g. i^{th}. The elements with the value of the i^{th} dimension smaller than the split value are part of the left subtree and other elements are part of the right subtree. The root of the tree presents all the elements

and the splitting is done recursively in each of the successor until the node contains less than a predefined number of elements (bucket size). The terminal nodes (called leaves) represent mutually exclusive small subsets of elements which collectively form a partition of the record space.

The k-d tree data structure provides an efficient mechanism for examining only those elements closest to the query point thereby reducing the computation required to find the best matches. In the application of the k-d trees to finding nearest neighbours to a query elements [5, 2] it is important to pay attention to the following:

- the splitting dimension is chosen so to maximize the variance in that dimension and
- the split value should split the elements into two equal sized partitions.

The computational complexity of building such optimized k-d tree is in order of $O(k \cdot N \cdot logN)$ where N is the number of elements.

The nearest neighbour search algorithm [5] which finds t elements closest to the query point q can be described as a recursive procedure. The argument to the procedure is the node under investigation. We begin the search by calling it with the root node. If the node under investigation is leaf then all the elements in the bucket are examined. A list of t closest elements so far encountered and their distances to the query point is always maintained as a priority queue during the search. Whenever we examine an element and find it to be closer than the most distant member of the list, the list is updated. If the node under investigation is not terminal the recursive procedure is called for the node representing the elements on the same side of partition as the query point. When we return from recursive call a test is made to determine if it is necessary to consider the elements on the side of partition opposite to the query point. This is only necessary if the geometric boundaries delimiting these elements overlap the ball centred at the query point with radius equal to the distance to the t^{th} closest element so far encountered.

The expected search time for t nearest neighbours to a prespecified query record is proportional to $logN$. To understand the effectiveness of the search procedure we must take into account that at each node the partition divides the current elements and reduces the multidimensional key space. The volume of this space gets smaller and smaller as we delve deeper into the tree.

3 K-d trees in machine learning

When using k-d trees in machine learning we first face the problem that datasets we use are typical not collections of records in $\Re^k$. The problems are mostly described by a mix of discrete and continuous attributes. We are of course not satisfied to use only continuous attributes to determine the distance to the query point (sometimes the problem is described with discrete attributes only), since this reduces the information available and skews the space. There are many ways how to incorporate all the attributes into the distance computation [12, 14], here we describe and use only the most common one. The distance function is unweighted sum of distances by the attributes

$$d(x,q) = \sum_{i=1}^{k} \delta(x_i, q_i), \tag{1}$$

$$\delta(x_i, q_i) = \begin{cases} |x_i - q_i| & ;\ i \text{ is continuous} \\ 0 & ;\ i \text{ is discrete and } x_i = q_i \\ 1 & ;\ i \text{ is discrete and } x_i \neq q_i \end{cases} \tag{2}$$

To insure that contribution of each attribute (dimension) is in $[0,1]$ range the continuous attributes are normalized by subtracting minimum and dividing by their observed range.

Now let we see how we can lower time complexity in the Relief family of algorithms (Relief [6], ReliefF [7], and RReliefF [11]). The key idea of the original Relief algorithm [6], given in Figure 1, is to estimate the quality of attributes according to how well their values distinguish between the instances that are near to each other. For that purpose, given a randomly selected instance R (line 3), Relief searches for its two nearest neighbours: one from the same class, called *nearest hit* H, and the other from a different class, called *nearest miss* M (line 4). It updates the quality estimation W[A] for all the attributes A depending on their values for R, M, and H (lines 5 and 6). The process is repeated for m times, where m is a user-defined parameter.

Algorithm Relief
Input: for each training instance a vector of attribute values and the class value
Output: the vector W of estimations of the qualities of attributes

```
1.  set all weights W[A] := 0.0;
2.  for i := 1 to m do begin
3.      randomly select an instance R;
4.      find nearest hit H and nearest miss M;
5.      for A := 1 to #all_attributes do
6.          W[A] := W[A] - diff(A,R,H)/m
7.                       + diff(A,R,M)/m;
8.  end;
```

Figure 1: The basic Relief algorithm

We can see (lines 2 and 4) that we are searching for nearest neighbours inside a loop. The time complexity of the algorithm for N training instances and A attributes is $O(m \cdot A \cdot N)$.

If we use k-d trees we first build the optimal k-d tree (outside the loop) in $O(A \cdot N \cdot logN)$ steps so we need

only $O(m \cdot A)$ steps in the loop and the total complexity of the algorithm is now the complexity of the preprocessing which is $O(A \cdot N \cdot logN)$. The required sample size m is related to the problem complexity (and not the sample size), and is typically much more than $logN$ so we have reduced the complexity of the algorithm.

ReliefF and RReliefF are searching for several nearest neighbours but their computational complexity is the same as that of Relief. They also need $O(A \cdot N \cdot logN)$ steps to build k-d tree, and in the main loop they select t nearest neighbours in $logN$ steps, update weights in $O(t \cdot A)$ but $O(m(tA + \log N))$ is asymptotically less than the preprocessing which means that the complexity has reduced to $O(A \cdot N \cdot logN)$ for ReliefF and RReliefF, too.

Our analysis shows that ReliefF family of algorithms is actually in the same order of complexity as multikey sort algorithms.

4 Experiments

To test the performance of k-d tree based nearest neighbour search algorithm (kdSearch) in machine learning we have compared it with priority queue (implemented with partially ordered tree) based implementation of nearest neighbour search (pqSearch). We have chosen 6 typical machine learning domains with the following characteristics:

Abalone: predicting the age of the abalone, 1 nominal and 7 continuous attributes, 4177 instances.

Autoprice: prices of the vehicles, 10 nominal , 15 continuous attributes, 201 instances.

Wisconsin: time to recur in Wisconsin breast cancer database, 32 continuous attributes, 198 instances.

Modulo-8-2: an integer generalization of the parity concept of order 2. Value of each attribute is an integer value in the range 0-7. Half of the attributes are treated as discrete and half as continuous; each continuous attribute is exact match of one of the discrete attributes. We have chosen 5 discrete and 5 continuous attributes and 1000 examples for this database.

Parity-2: continuously randomized parity concept of order 2. Data set consists of 1000 examples which are described by 10 discrete, Boolean attributes.

Cosinus: non-linear dependency with cosinus. There are 10 continuous attributes with values from 0 to 1 and 1000 examples.

Since in this paper we have tested kdSearch in the framework of RReliefF we have built it into the CORE learning system [10]. This choice also conditioned the datasets we have chosen. The first three are representative function learning datasets from UCI [9] and the next three are artificial datasets from RReliefF paper [11].

Table 1: Comparison of running times (in seconds) of kdSearch and pqSearch inside RReliefF algorithm. The two estimation columns give results when estimating attributes and the two construction columns present performance of RReliefF inside constructive induction algorithm.

	estimation		construction	
dataset	k-d	pq	k-d	pq
Abalone	30.5	66.3	183.5	283.2
Autoprice	0.5	0.4	1.8	1.8
Wisconsin	0.8	0.7	5.2	5.2
Modulo-8-2	3.8	3.0	111.9	94.8
Parity-2	0.3	1.1	0.6	1.4
Cosinus	6.6	5.1	232.9.8	188.1

First we checked how well k-d trees perform inside RReliefF algorithm. Default settings were used for RReliefF algorithm (searching for 200 nearest neighbours) and the bucket size was set to 10 (as recommended by [2]).

On left hand side of Table 1 we compare average times of 10 runs for RReliefF with kdSearch and RReliefF with pqSearch. We can see that in estimation it is not always worth to use k-d trees, especially in datasets where the number of examples is small (however in small datasets the absolute differences are small). In bigger datasets the differences become greater in favour of k-d trees and can reduce the computation time by more than 50%.

Next we have used RReliefF to estimate attributes inside constructive induction algorithm which tries to combine attributes in various ways (conjunction, sum, and product) and then estimates the combinations. Best combinations are retained and further combined with other attributes. It turned out that kdSearch was useful in the same datasets as in estimation, so the fact that k-d tree is built only once in each construction phase and subsequently only kdSearch procedure is called did not have enough influence on the overall performance.

In the third set of experiments we were systematically checking the robustness of kdSearch against large number of attributes (we know from previous studies e.g., [5] that kdSearch works well for small number of attributes). For this experiment we have constructed artificial dataset with one half of discrete attributes with 4 values and one half of continuous attributes with values assigned randomly following the Gaussian distribution. We have fixed the number of examples to 2000 and varying the number of attributes from 10 to 200. Figure 2 presents the results.

The advantage of kdSearch is relatively small even with smaller number of attributes while with many attributes the pqSearch is better. The reason for this is called the curse of dimensionality and was observed also in [3].

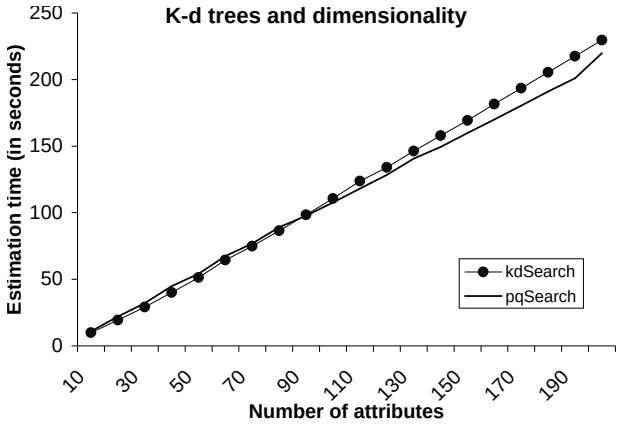

Figure 2: Estimation time of RReliefF with kdSearch and pqSearch when varying the number of attributes.

With large number of dimensions the k-d tree is too shallow (its depth is $\log N$) for all the attributes to occur on the path from the root to the leaves. This forces the kdSearch to investigate almost all the leaves and its power based on ability to cut off many branches disappears.

5 Conclusions

We have investigated the use of k-d tree based nearest neighbour search in the family of attribute estimation algorithms Relief and showed that the asymptotical complexity of these algorithms can be reduced to $O(A \cdot N \cdot \log N)$ which is the same order of complexity as that of multikey search algorithms.

Our empirical testings have revealed that with some typical machine learning databases the kdSearch might be worth the effort, since it can reduce the estimation and construction time by up to 50% on larger datasets. Unfortunately the curse of dimensionality is preventing the use of kdSearch in data mining applications where there are many attributes. The performance of k-d trees under these circumstances is not worth additional effort and we recommend the use of simpler data structures.

References

[1] Jon Luis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 15(9):509–517, 1975.

[2] Alan J. Broder. Strategies for efficient incremental nearest neighbor search. *Pattern Recognition*, 23(1/2):171–178, 1990.

[3] Kan Deng and Andrew W. Moore. Multiresolution instance-based learning. In *Proceedings of the IJCAI-95*, pages 1233–1239. Morgan Kaufmann, 1995.

[4] Usama M. Fayad, George Piatetsky-Shapiro, and Padraic Smith. From data mining to knowledge discovery: An overview. In Usama M. Fayad, George Piatetsky-Shapiro, and Padraic Smith, editors, *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.

[5] Jerome H. Friedman, Jon Luis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. Technical Report STAN-CS-75-482, Stanford University, 1975.

[6] Kenji Kira and Larry A. Rendell. A practical approach to feature selection. In D.Sleeman and P.Edwards, editors, *Proceedings of International Conference on Machine Learning*, pages 249–256. Morgan Kaufmann, 1992.

[7] Igor Kononenko. Estimating attributes: analysis and extensions of Relief. In L. De Raedt and F. Bergadano, editors, *Machine Learning: ECML-94*, pages 171–182. Springer Verlag, 1994.

[8] Andrew W. Moore, Jeff Schneider, and Kan Deng. Efficient locally weighted polynomial regression predictions. In Dough Fisher, editor, *Machine Learning: Proceedings of the Fourteenth International Conference (ICML'97)*, pages 236–244. Morgan Kaufmann Publishers, 1997.

[9] P.M. Murphy and D.W. Aha. UCI repository of machine learning databases, 1995. (http://www.ics.uci.edu/ mlearn/-MLRepository.html).

[10] Marko Robnik Šikonja. Core - a system that predicts continuous variables. In *Proceedings of Electrotechnical and Computer Science Conference*, pages 145–148, Portorož, Slovenia, 1997.

[11] Marko Robnik Šikonja and Igor Kononenko. An adaptation of Relief for attribute estimation in regression. In Dough Fisher, editor, *Machine Learning: Proceedings of the Fourteenth International Conference (ICML'97)*, pages 296–304. Morgan Kaufmann Publishers, 1997.

[12] Craig Stanfill and David Waltz. Toward memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228, 1986.

[13] Mark Allan Weiss. *Data Structures and Algorithm Analysis*. The Benjamin/Cummings, 1995.

[14] Dietrich Wettschereck, David W. Aha, and Takao Mohri. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11:273–314, 1997.