Chapter 5

Efficient Memory Information Retrieval

In this chapter, we will talk about two topics: (1) What is a kd-tree? (2) How can we use kd-trees to speed up the memory-based learning algorithms? Since there are many details in the second topic, we only discuss how to improve the efficiency of Kernel regression in this chapter, to demonstrate the approach in principle. In next chapter, we will explain the details of applying kd-tree techniques to improve the efficiency of locally linear regression and locally weighted logistic regression.

5.1 Efficient information retrieval

Suppose there are a set of memory data points whose input space is 2-dimensional, shown in Figure 5-1. Given a query (x_q, y_q), a task of information retrieval is to find this query's neighboring memory data points. The brute force approach is to measure the distances from this query to each of the memory data points. Then based on these distances, it is straightforward to decide which memory data points are the query's neighbors. The distance may be Euclidean or another metric depending on the specific domain. The drawback of the brute force method is obvious: since its computational cost is $O(N \times d)$, where N is the memory size and d is the dimensionality of the input space. When the memory size N becomes very large, its costs will increase, too.

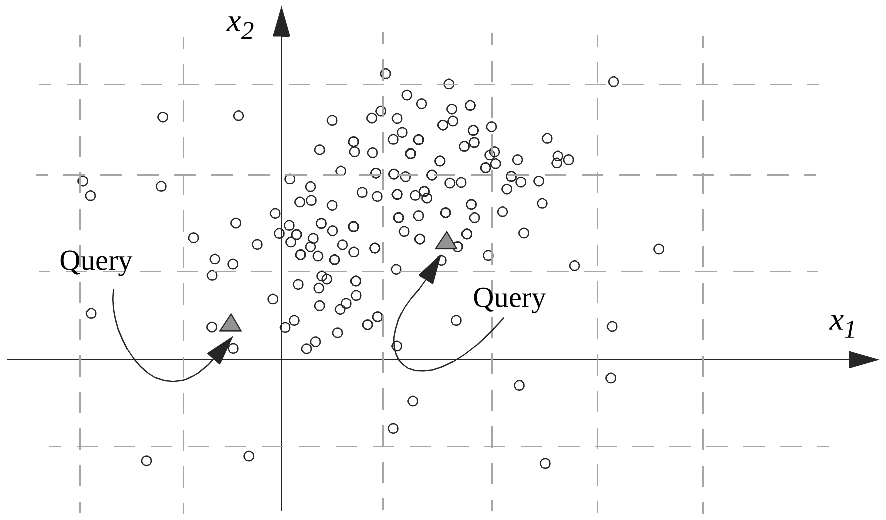


Figure 5-1: Grid for efficiency information retrieval.

To improve the efficiency of finding the neighbors, we can partition the input space of the memory data points into many cells by means of a grid. When a query arrives, we can consult the cell where the query locates and its neighboring cells, instead of visiting all the memory data points individually. In this way, the computational cost shrink from $O(N \times d)$ to $O(n \times d^2)$, where *n* is the number of memory data points in the concerned cell(s). (If we neglect the cost of finding the cell where the query resides.) The grid method performs the best when the memory data points distribute uniformly, so that *n* tends to be *N / G*, in which *G* is the number of grids in the whole input space. However, there is no guarantee that the memory data points distribute uniformly forever and wherever. Sometimes most of the memory data points are packed in only a limited number of cells, while the other cells are almost vacant. Therefore, the contribution of the grid method to the efficiency is not reliable.

The kd-tree technique [Preparata et al, 85] is similar to the grid method in the sense that it also partitions the input space into many cells. However, the partition is flexible with respect to the density of the data points in the input space. Wherever, the density is high in the input space, the resolution of the kd-tree's partition at that region is also high, so that the cells tend to be small. Otherwise, for those regions where there are only a limited number of memory data points, the partition resolutions are low, and the cells are large.

5.2 Kd-tree Construction and Information Retrieval

A kd-tree is a binary tree that recursively splits the whole input space into partitions, in a manner similar to a decision tree [Quinlan, 93] acting on real-valued inputs. Each node in the kd-tree represents a certain hyper-rectangular partition of the input space; the children of this node denote subsets of the partition. Hence, the root of the kd-tree is the whole input space, while the leaves are the smallest possible partitions this kd-tree offers. And each leaf explicitly records the data points that reside in the leaf. The tree is built in a manner that adapts to the local density of input points and so the sizes of partitions at the same level are not necessarily equal to each other.

In our formulation of the kd-tree structure, each node records the hyper-rectangle covered by it. This is defined as the smallest bounding box that contains all the data points owned by this node of the tree. Each non-leaf node has two children representing two disjoint subregions of the parent node. The break between the children is defined by two values: **`split_d`** is the splitting dimension, which determines which component of input space the children will be split upon; **`split_v`** determines the numerical value at which each split occurs. The data points owned by the left child of a node are those data points owned by the node which are less than value **`split_v`** in input component **`split_d`**. The right child contains the other data points. A sample kd-tree is shown in Figure 5-2.

To construct a tree from a batch of training data points in memory, we use a top-down recursive procedure. This is the most standard way of constructing kd-trees, described, for example, in [Preparata et al., 85] [Omohundro, 91]. In our work, we use the common variation of splitting a hypercube in the center of the widest dimension instead of at the median point. This method of splitting does not guarantee a balanced tree, but leads to generally more cubic hyper-rectan-

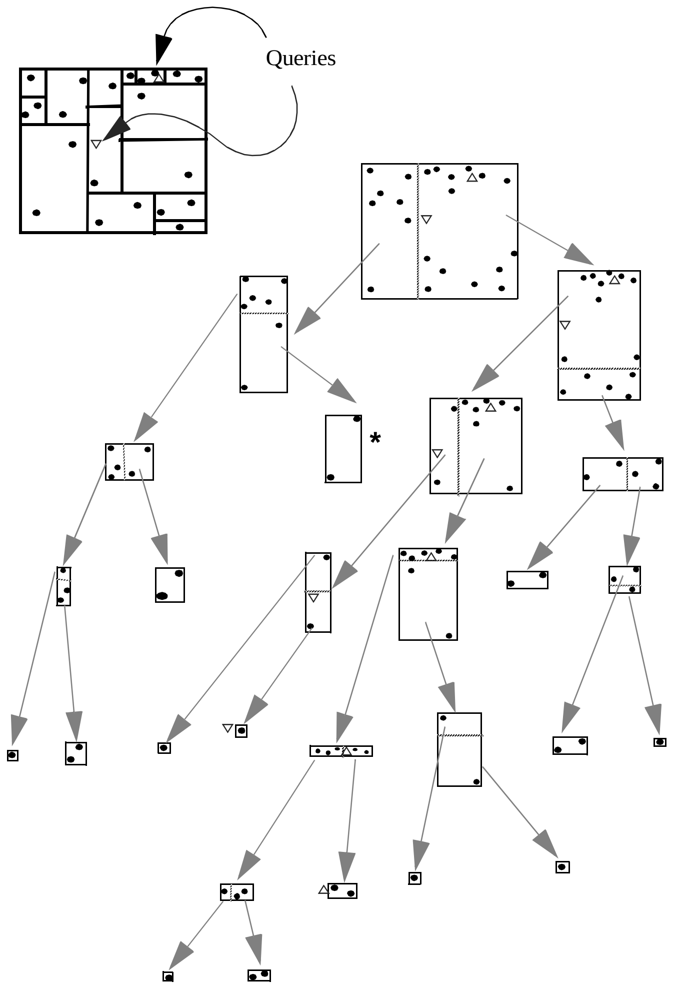


Figure 5-2: To implement the grouping idea, we use hyper-rectangles with *k*d-tree. To find the neighborhood of a certain query (triangle), we can recursively search the tree from the root towards to the leave where the query resides. For different query (reversed triangle), we can use the same kd-tree but choose different nodes.

gles, which has empirically proved better than other schemes (pathologically imbalances are conceivable, but trivial modifications to the algorithm prevent that.) The cost of making a tree from N data points is $O(Nd \log N)$.

The base case of the recursion occurs when a node is created with N_{min} or fewer data points. Then those data points are explicitly stored in the leaf node. In our experiments, $N_{min} = 2$.

To incrementally add a new data point to the tree, the leaf node containing the point is determined (*O(logN)* cost). The data point is inserted there (and a new subtree is recursively built if the number of nodes exceeds N_{min}).

Given a query (x_q, y_q), to find those memory data points whose input vectors are close to x_q, we can recursively search the tree from the root towards to leaves, referring to Figure 5-2, with the triangle query. According to the pre-defined range of "neighborhood", it is straightforward to find those branches of the kd-tree, which are close to the branches where the query resides. Two issues to be noticed:

1. With different ranges of the "neighborhood", the "neighboring" branches can be different. The neighboring branches with respect to a strict defined neighborhood is a subset of those neighboring branches corresponding to a loose definition. This characteristic is desirable, because it allows us to find those neighboring data points corresponding to any definition of the neighborhood along the local-global spectrum.

2. Although we will use the kd-tree to find a *set* of neighboring data points, it is also possible to find the "exact" nearest neighboring data point. For the example in Figure 5-2 with the reversed triangle query, to find its nearest neighbor data point, we wish we could search from the root of the tree down towards to the leaf where the query locates, so that the cost is $O(\log N)$, where N is the memory size. Unfortunately, it is possible that its nearest neighboring data point is in another leaf of a remote branch of the kd-tree, marked with "*" in the diagram. More theoretical analysis refers to [Kleinberg, 97]. The standard nearest neighbor algorithm, [Preparata et al, 85] [Moore, 90], avoids this problem while still only requiring $O(\log N)$ time.

5.3 Cached Kd-tree for Memory-based Learning

The goal of our exploring kd-trees is not to find the nearest neighbor, and not only to find a set of nearest neighbors, but mainly to enhance the efficiency of the memory-based learning methods. The basic principle is to cache useful statistical information into the kd-tree nodes, so that when we do the memory-based learning process, instead of visiting every relevant memory data point, we mainly rely on the statistical information in the tree nodes. In this chapter, we focus on using this cached kd-tree to speed up Kernel regression, to demonstrate the approach in general.

Kernel regression

In Chapter 2, we discussed using Kernel regression's idea to approximate $P(y_q \mid S_p, x_q)$, i.e. the probability that a given query data point (x_q, y_q) belongs to a system S_p, where the knowledge of S_p comes from a set of memory data point, (x_1, y_1) ..., (x_N, y_N), which is the observations of S_p's previous behavior. Cached kd-trees can improve the efficiency of Kernel regression (for example, [Franke, 82]), not only for the approximation of $P(y_q \mid S_p, x_q)$, but also for the general purpose use. As a popular machine learning method, Kernel regression is often used to do prediction: given an input vector x_q, which is called query, Kernel regression predicts its output, $\hat{y}_q(x_q)$, based on the memory data points (x_1, y_1), ..., (x_N, y_N). We assume all the memory data points were generated by an identical system.

Kernel regression use the weighted average of the outputs of all the memory data points to predict $\hat{y}_q(x_q)$:

$$\hat{y}_q(x_q) = \left(\sum_{i=1}^{N} w_i y_i\right) \Big/ \left(\sum_{i=1}^{N} w_i\right) \qquad (5\text{-}1)$$

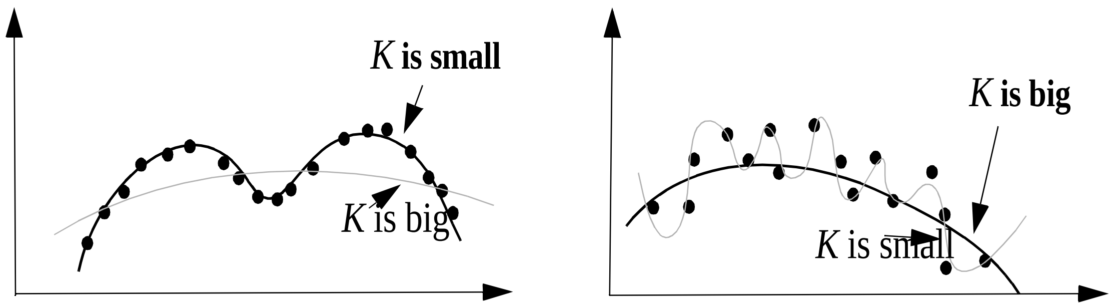


Figure 5-3: For the noiseless data in the top example, a small K gives the best regression (in terms of future predictive accuracy). For the noisy data in the bottom example, a large K is preferable.

where w_i is the weight assigned to the i'th datapoint in our memory, and is large for points close to the query and almost zero for points far from the query. It is usually calculated as a decreasing function of Euclidean distance, for example by Gaussian:

$$w_i = \text{Const} \times \exp\left(-\frac{\|x_q, x_i\|^2}{2K_w^2}\right)$$

As we have mentioned previously, K_w is the Kernel width. The bigger the parameter K_w is, the flatter the weight function curve is, which means that many memory points contribute quite evenly to the regression. As K_w tends to infinity the predictions approach the global average of all points in the database. If the K_w is very small, only closely neighboring data points make a significant contribution. K_w is an important smoothing parameter for kernel regression. If the data is noise free then a small K_w will avoid smearing away fine details in the function. If the data is relatively noisy, we expect to obtain smaller prediction errors with a relatively large K_w. This is illustrated in Figure 5-3.

The drawback of kernel regression is the expense of enumerating all the distances and weights from the memory points to the query. This expense is incurred every time a prediction is required. Several methods have been proposed to address this problem, reviewed as following:

1. [Preparata *et al*, 85] proposed a range-search solution. Similar to our cached kd-tree method, the range-search solution finds all points in the kd-tree that have significant weights, and then only sum together the weighted components of those points. This is only practical if the kernel width K_w is small. If it is large, all the memory data points may have significant weights, but with only small local variations, thus range searching would sum all the points individually. Even in cases of small kernel widths, but if there are many data points in the neighborhood, the range search method will need to search all the data points individually and may still end up with a large computational cost.

2. Another solution to the cost of conventional Kernel regression is *editing* (or *prototypes*): most data points are forgotten and only particularly representative ones are used (e.g. [Kibler and Aha, 88] [Skalak, 94]). Kibler and Aha extended this idea further by allowing data points to represent local averages of sets of previously-observed data points. This can be effective, and unlike range-searching can be applicable even for wide kernel widths. However, the degree of local averaging must be decided in advance, and queries cannot occur with different kernel widths without rebuilding the prototypes. A second occasional problem is that if we require very local predictions, the prototypes must either lose local details by averaging, or else all the data points are stored as prototypes.

3. Decision trees and *k*d-trees have been previously used to cache local mappings in the tree leaves [Grosse, 89], [Moore, 90], [Omohundro, 91], [Quinlan, 93]. These algorithms provide fast access once the tree is built, but a new structure needs to be built each time new learning parameters, such as Kernel width, are required. Furthermore, the resulting predictions from the tree have substantial discontinuities between boundaries. Only in [Grosse, 89] is continuity enforced, but at the cost of tree-size, tree-building-cost and prediction-cost all being exponential in the number of input variables.

Computing the kernel regression sums

Now it is time for us to use the cached kd-tree to improve the efficiency of Kernel regression, and at the same time avoid the drawbacks of the other competing methods.

Recall that each kd-tree node represents a hyper-rectangle sub-region of the input space, which covers a set of memory data points. Assume in one node there are *n* data points, and corresponding to a certain query, these *n* data points' weights are all close to a value *w*; in other words, the weight of the *i*'th data point in this node is $w_i = w + \xi_i$, where all ξ_i's are small. Referring to Equation 5-1, when performing Kernel regression, we need to accumulate two sums over all data points in memory, including these *n* data points in this node,

$$\sum w_i y_i \quad \text{and} \quad \sum w_i$$

Restricting our attention to summations over the *n* data points in the concerned kd-tree node, we have,

$$\sum w_i y_i = \sum (\bar{w} + \varepsilon_i) y_i = \bar{w} \sum y_i + \sum \varepsilon_i y_i \quad \text{and}$$
$$\sum w_i = \sum (\bar{w} + \varepsilon_i) = n\bar{w} + \sum \varepsilon_i$$

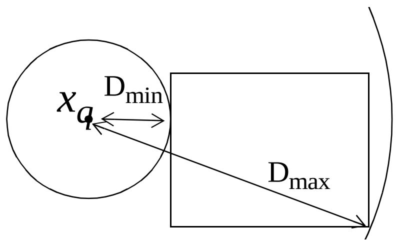
Providing we know *n*, *w* and Σy_i for the current node, we can therefore compute an approximation to $\Sigma w_i y_i$ and Σw_i in constant time without needing to sum individual data points contained in the node. This approximation to the partial sums is good to the extent that $\Sigma \varepsilon_i y_i$ is small with respect to $w \Sigma y_i$ and $\Sigma \varepsilon_i$ is small with respect to *nw*.

Therefore, we should cache two other pieces of information into each kd-tree node in conjunction with **`split_v`** and **`split_d`**: the number of data points below the current node, **`n_below`**, and the sum Σy_i of all output values of the data points contained in the node, **`sum`**. These are two of the three values needed to compute the contribution of a kd-tree node to the partial sums in Kernel regression. The third component, *w*, depends upon the location of the query and is determined dynamically in a manner described shortly.

With such cached information in each kd-tree node, we can efficiently approximate $\Sigma w_i y_i$ and Σw_i, summed over all data points in the kd-tree, so as to speed up the process of Kernel regression. This is performed by a top-down search over the tree. At each node we make a decision between:

1.(Cutoff) Treat all the points in this node as one group (a cheap operation) or
2.(Recurse) search the children.

We will use the cutoff option if we are confident that all weights inside the node are similar. Given the current query x_q and the hyper-rectangle of the current node it is an easy matter to compute $D_{\min}$ and $D_{\max}$: the minimum and maximum possible distances of any datapoint in this node to the query (computational cost is linear in the number of dimensions). From these



values one can then compute the maximum and minimum possible weights $w_{\max}$ and $w_{\min}$ of any data points owned by this node, since the weight of a point is a decreasing function of distance to the query. We thus decide if $w_{\max}$ and $w_{\min}$ are close enough to warrant the cut-off option.

The search is thus a recursive procedure which returns two values: *sum-weights* and *sum-wy*. If the cutoff option is taken, then estimate the weight of all data points as $\bar{w} = (w_{min} + w_{max})/2$ and return:

$$\begin{aligned} \textit{sum-weights} &= \mathbf{n\_below} \times \bar{w} \\ \textit{sum-wy} &= \mathbf{sum} \times \bar{w} \end{aligned}$$

If the cutoff option is not taken, recursively compute *sum-weights* and *sum-wy* for the left and right children, and then return:

sum-weights = sum-weights(left) + sum-weights(right)

sum-wy = sum-wy(left) + sum-wy(right)

Search cutoffs

Last section described how we can make our approximation arbitrarily accurate by bounding the maximum deviation we will permit from the true weight estimate with a value $\varepsilon_{\max} > 0$ and then making $\varepsilon_{\max}$ arbitrarily small. Thus the simplest cutoff rule in the kd-tree search would be to cutoff if $w_{\max} - w_{\min} < \varepsilon_{\max}$. It is easy to show that this guarantees that the total sum of absolute deviations $|\Sigma\varepsilon_i|$ is less than $N_T\varepsilon_{\max} / 2$ where N_T is the number of points in the tree. There are, however, other possible cutoff criteria which provide arbitrary accuracy in the limit, but which, when used as an approximation, have more satisfactory properties.

The simple cutoff rule does not take into account that a larger total error will occur if the node contains very many points than if the node contains only a few points. It does also not account for the fact that in a practical case we are less concerned about the absolute value of the sum of deviations $|\Sigma\varepsilon_i|$ but rather the size of $|\Sigma\varepsilon_i|$ relative to the sum of the weights Σw_i. Some simple analysis reveals a cutoff criterion to satisfy both of these intuitions. Cutoff only if

$$(w_{\max} - w_{\min})\, N_B < \tau\, \Sigma \mathrm{w}_i$$

where N_B is the number of data points below the current node. Simple algebra reveals that this guarantees

$$|\,\Sigma\varepsilon_i\,| < 0.5\ \mathrm{G}\ \tau\ \Sigma \mathrm{w}_i$$

where G is the number of groups finally used in the search (and thus $\mathrm{G} < N_T$, hopefully considerably less). Notice that this cutoff rule requires us to know $\Sigma \mathrm{w}_i$ in advance, which of course we do not. Fortunately the sum of weights obtained so far in the search can be used as a valid lower bound, and so the real algorithm makes a cutoff if

$$\frac{(w_{\max} - w_{\min})N_B}{\text{weight so far in search}} < \tau$$

where τ is a system constant.

5.4 Experiments and Results

Let us review the performance of the Kernel regression with the help of cached kd-tree in comparison to the conventional Kernel regression. In the first experiment we use a trigonometric function of two inputs with added noise: x_i = uniformly generated random vector with all components between 0 and 100 and y_i = a function of x_i (which ranges between 0 and 100 in height), with gaussian noise of standard deviation 10.

10,000 data points were generated. Experiments were run for different values of kernel width K_w. In all experiments, the cutoff threshold τ was 0.005. Figure 5-4 (a1) shows the test-set error on 1000 test points for both regular kernel regression (“Regular KR”) and cached kd-tree’s kernel regression (“Tree KR”) graphed for different values of K_w. The values are very close, indicating that Tree KR is providing, for a wide range of kernel widths, a very close approximation to Regular KR. Figure 5-4 (a2) shows the computational cost (in terms of the summations that dominate the cost of KR) of the two methods. Regular KR sums all points, and so is a constant 10,000 in cost. Tree KR is substantially cheaper for all values of K_w, but particularly so for very small and very large values.

Figures 5-4 (b1) and (b2) show corresponding figures for a similar trigonometric function of five inputs. This still shows similar prediction performance as Regular KR. The cost of kd-tree’s Kernel regression is still always less than Regular KR, but in the worst case the computational saving is only a factor of three (when K_w = 40, Tree KR cost = 3,200). This is not an

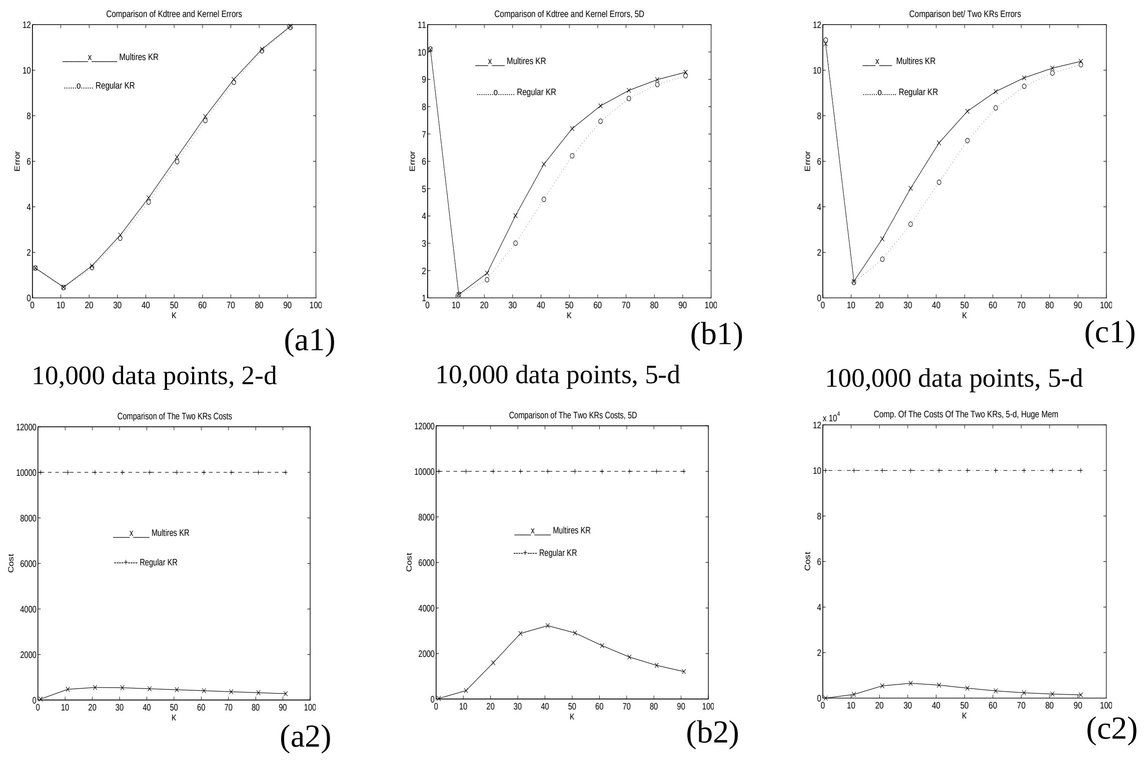


Figure 5-4: Comparison between the errors (*1) and the costs (*2) between regular kernel regression versus cached kd-tree's one. In the cases of (a*), the dataset is of 2-d inputs, of size 10,000. In (b*), 5-d inputs, dataset size 10,000. In (*c), 5-d inputs, 100,000 data points.

especially impressive result. However, for any fixed dimensionality and kernel width, costs rise sub-linearly (in principle logarithmically) with the number of data points. To check this, we ran the same set of experiments for a dataset of ten times the size: 100,000 points. The results, in Figure 5-4 (c1) and (c2), show that with this large increase in data, the effectiveness of cached kd-tree's KR becomes more apparent. For example, consider the K_w = 40 case. With 100,000 data points instead of 10,000, the cost is only increased from 3,200 to 5,700 while the cost of Regular KR (of course) increased from 10,000 to 100,000.

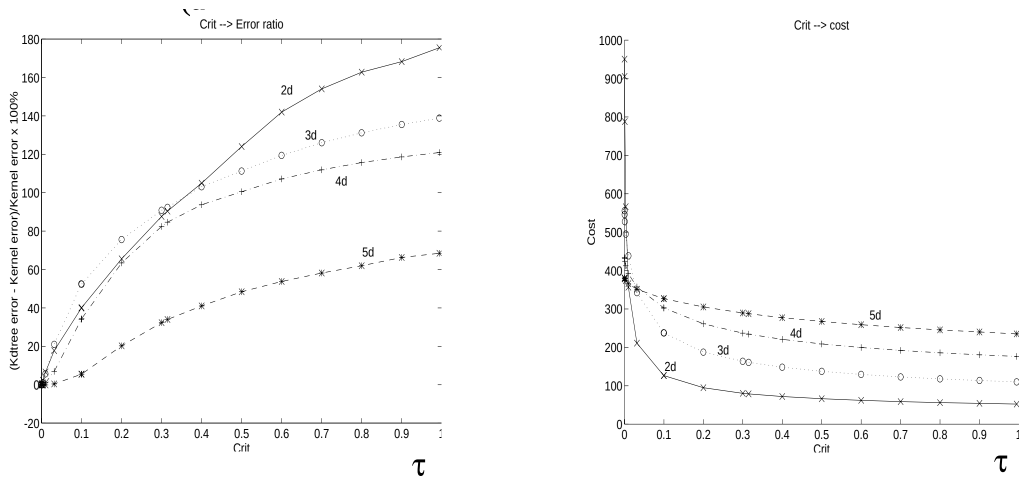


Figure 5-5: (Upper) the relative accuracy and (lower) the computational cost of kd-tree's KR against τ --- the cutoff threshold.

Investigating the τ threshold parameter

Next, we will examine the effect of the τ parameter on the behavior of the algorithm. As τ is increased we expect the computational cost to be reduced, but at the expense of the accuracy of the predictions in comparison to the regular KR. The results in Figure 5-5 agree with this expectation: the left hand graph shows that for 2-d, 3-d, 4-d and 5-d datasets (each with 10,000 points) the proportional error between cached kd-tree's and regular regression increases with τ. The right hand graph shows a corresponding decrease in computational cost.

Real datasets

In another experiment, we ran cached kd-tree's KR on data from several real-world and robot-learning datasets. Further details of the datasets can be found in [Maron et al, 94]. They include an industrial packaging process for which the slowness of prediction had been a reasonable cause for concern. Encouragingly, cached kd-tree's KR speeds up prediction by a factor of 100 with no discernible difference in prediction quality between cached kd-tree's and regular KR. This and other results are tabulated below. The costs and error values given are averages taken over an independent test set. Notably, the datasets with the least savings were **pool**, which had few data points, and **robot**, which was high dimensional.

Table 5-1: Real dataset test of cached kd-tree's kernel regression

Domain	*Dataset Size*	*Dim of Input*	*Regular KR Cost*	*Tree's KR Cost*	*Regular KR Err.*	*Tree's KR Error*
Energy	2144	5-d	2144	232.9	1.687	1.690
Package	32000	3-d	32000	289.0	6.07	6.09
Pool	213	3-d	213	50.7	2.125	2.123
Protein	4664	3-d	4664	383.8	1.036	1.106
Robot	871	14-d	871	225	6.354	6.976

High dimensional, non-uniform data

Our final experiment concerned the question of how well the method performs if the number of input variables is relatively large, but if the attributes are not independent. For example, a common scenario in robot learning is for the input vectors to be embedded on a lower-dimensional manifold. We performed two experiments, each with 9 inputs and 10,000 data points. In the first experiment, the components of the input vectors were distributed uniformly randomly. In the second experiment the input vectors were distributed on a non-linear 2-d manifold of the 9-d input space. The results were:

Table 5-2: Cached kd-tree's kernel regression for sub-manifold cases

	9-d uniform	*9-d inputs on 2-d manifold*
Regular KR cost	10,000	10,000
Cached kd-tree's KR cost	3,100	430
Regular KR mean testset error	13.07	1.06
Cached kd-tree's KR mean testset error	13.08	1.15

The results indicate that, as would be expected, the cost advantage of cached kd-tree's KR is not large (a factor of 3) for 9-d uniform inputs, but is far better if the inputs are distributed within a lower-dimensional space.

5.5 Summary

Kernel regression with the help of the cached kd-tree is preferable in case the application needs the following properties:

- Flexibility to work throughout the local/global spectrum.
- The ability to make predictions with different parameters without needing a retraining phase.

In addition, cached kd-tree's Kernel regression has a number of additional flexibilities. Once the kd-tree structure is built, it is possible to make different queries with not only different kernel widths K_w, but also different Euclidean distance metrics, with subsets of attributes ignored, or with some other distance metrics such as Manhattan. It is also possible to apply the same technique with different weight functions and for classification instead of regression.

Dimensionality is a weakness of cached kd-tree's Kernel regression. Diminishing returns set in above approximately 10 dimensions if the data points are distributed uniformly. This is an inherent problem for which no solution seems likely because uniform data points in high dimensions will have almost all data points almost exactly the same distance apart, and a useful notion of locality breaks down.

This chapter discussed an efficient implementation of kernel regression. In next chapter, we will apply exactly the same algorithm to locally weighted linear regression and locally weighted logistic regression, in which a prediction fits a local polynomial or a local logistic function to minimize the locally weighted sum squared error. The only difference is that each node of the kd-tree stores the regression design matrices of all points below it in the tree. This permits fast prediction and also fast computation of confidence intervals and analysis of variance information.