Chapter 6

Using Kd-trees for Various Regressions

In last chapter, we discussed how to use kd-tree to make kernel regression more efficient. In fact, kd-tree can be used for other regressions, too. In this chapter, we will introduce how to apply it to speed up locally weighted linear regression and locally weighted logistic regression.

6.1 Locally Weighted Linear Regression

Linear regression can be used as a function approximator. Given a set of memory data points, known as *training* data points, a *global* linear regression finds a line with parameters such that the sum of the residual squares from the training data points to the line is minimized. In the example of Figure 6-1(a), each data point has one input and one output. A global linear regression finds a line,

$$\hat{y}(x) = \beta_0 + \beta_1 x$$

with β_0 and β_1, so that the sum of the residual squares is minimized, i.e.,

$$(\beta_o, \beta_1) = \arg\min \sum_{i=1}^{N} (y_i - \hat{y}(x_i))^2 = \arg\min \sum_{i=1}^{N} (y_i - \beta_o - \beta_1 x_i)^2$$

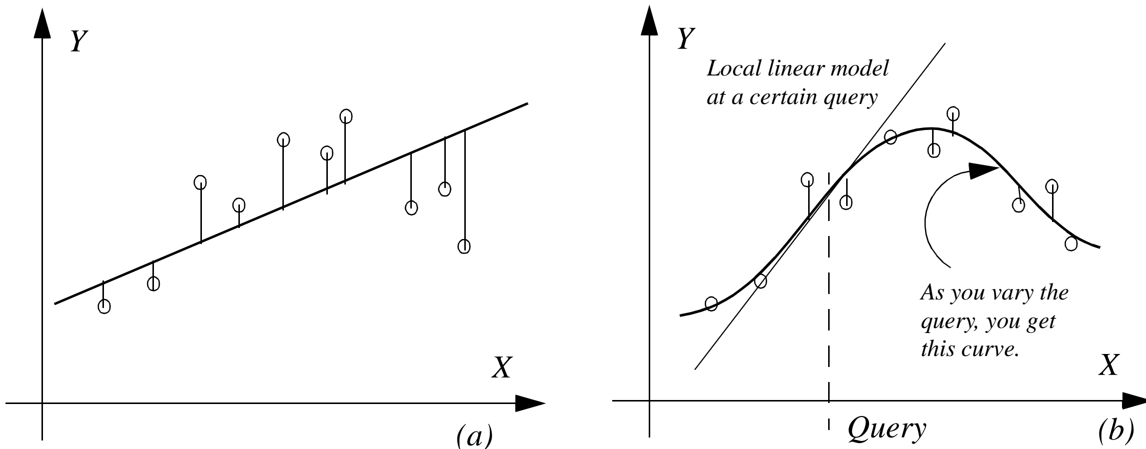

Figure 6-1: (a) A global linear regression (b) A locally weighted linear regression.

By global, we mean β_0 and β_1 are fixed for any possible *x*. Obviously, this linear function approximator would not work for any non-linear functions. That is the reason we have more interest in *locally weighted* linear regression.

Locally weighted regression assumes for any local region around a query point, x_q, the relationship between the input and output is linear. To construct the local function approximator, the local linear parameters can be approximated by minimizing the *weighted* sum of residual squares. For the example as shown in Figure 6-1(b), the weighted sum of residual squares is

$$\sum_{i=1}^{N} w_i^2 (y_i - \beta_o - \beta_1 x_i)^2$$

The weight, w_i, is usually a function of the Euclidean distance from the *i*'th training data points to the query, $// x_q - x_i //$. A popular form of the function is Gaussian.

After some algebra which requires no gradient descent, the linear parameters can be obtained directly by,

$$\hat{\beta} = (X^{T}WX)^{-1}(X^{T}WY) \tag{6-1}$$

where X is a N-row M-column matrix, N is the number of the training data points in memory, M is the dimensionality of the input space plus 1. If the input vector of the k'th data point in memory is x_k, the k'th row of X is $(1, x_k^T)$. Y is a vector consisting of the training data points' outputs. W is a diagonal matrix, whose k'th element is the square of the weight of the k'th training data point, w_k^2.

6.2 Efficient locally weighted linear regression

As we have known in last section, the crucial thing to improve the efficiency of locally weighted linear regression is to speed up the calculation of X^TWX and X^TWY. Since W is a diagonal matrix, X^TWX and X^TWY can be transformed as,

$$X^TWX = \sum_{i=1}^{N} w_i^2 x_i x_i^T \quad \text{and} \quad X^TWY = \sum_{i=1}^{N} w_i^2 x_i y_i$$

in which vector x_i corresponds to the i'th row of X, and y_i is the i'th element of Y vector.

Recall that the kd-tree is a binary tree, the root of the tree covers the whole input space, hence contains all the training data points in memory. The root can be split into two nodes: the left node and the right node, each of them covers a partition of the input space. Furthermore, the left node can be split into another pair of nodes, so does the right node. Hence, in the second layer there are four nodes at most. Therefore, to calculate X^TWX of all the memory data points, we can follow a recursion process,

$$(X^TWX)_{Root} = \sum_{i=1}^{N} w_i^2 x_i x_i^T = \sum_{i=1}^{N_{Left}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{Right}} w_i^2 x_i x_i^T$$

$$= (X^TWX)_{Left} + (X^TWX)_{Right}$$

$$= \sum_{i=1}^{N_{LeftLeft}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{LeftRight}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{RightLeft}} w_i^2 x_i x_i^T + \sum_{i=1}^{N_{RightRight}} w_i^2 x_i x_i^T$$

$$= (X^TWX)_{LeftLeft} + (X^TWX)_{LeftRight} + (X^TWX)_{RightLeft} + (X^TWX)_{RightRight}$$

in which N is the total number of training data points in memory, the sum of N_{Left} and N_{Right}, as well as the sum of $N_{LeftLeft}$, $N_{LeftRight}$, $N_{RightLeft}$, and $N_{RightRight}$, are equal to N.

Hence, to calculate X^TWX of the root, or any other node of the kd-tree, we can recursively sum its two children's X^TWX's. A leaf's X^TWX can be calculated according to the definition:

$$(X^TWX)_{Leaf} = \sum_{i=1}^{N_{Leaf}} w_i^2 x_i x_i^T$$

However, this recursion process does not bring us any gain in computational efficiency, because it still visits every training data point in memory. But sometimes we may be able to cutoff the computation at a node, if all the memory data points within this node have near-identical weights. In other words,

$$(X^TWX)_{Node} = \sum_{i=1}^{N_{Node}} w_i^2 x_i x_i^T \approx \bar{w}_{Node}^2 \left(\sum_{i=1}^{N_{Node}} x_i x_i^T \right) = \bar{w}_{Node}^2 (X^TX)_{Node} \tag{6-2}$$

If w_i, $i = 1, \ldots, N_{Node}$, are near identical.

This scenario happens for three reasons:

- All data points within the node are so far from the query vector, x_q, that their weights are near zeroes.
- All the data points are close together, providing no room for weight variation.
- The weight function varies negligibly over the partition of the input space covered by the current node.

Given a certain query, x_q, and a certain node, to judge if any of these situations happens, we can rely on the comparison of the lower bound and the upper bound of the weights of the memory data points within this node. Roughly speaking, if the difference between the upper bound and the lower bound is smaller than a threshold, then Equation 6-2 holds and the cutoff is permitted. Further discussion on the threshold will come latter in this section. To calculate the lower bound and the upper bound of the weights, recall that each node of the kd-tree corresponds to a hyper-rectangular partition of the input space, thus, given a query, x_q, it is straightforward to calculate the longest and the shortest distances from the query to the concerned hyper-rectangle. Because the weight function is a monotonic function of the distance, it is not difficult to calculate the lower bound and the upper bound of the weights based on the range of the distance.

Therefore, to calculate X^TWX for all the data points in memory, we can follow the recursive algorithm listed in Figure 6-2.

Similarly, we can efficiently calculate X^TWY. But be aware that we need to cache X^TX and X^TY into each node of kd-tree. When we build a kd-tree, we calculate X^TX and X^TY for each node, from the leaves in the bottom, upward to the root. Once this is done, the kd-tree is ready to handle any queries. When a query occurs, we follow the recursion algorithm in Figure 6-2, from

```
  calc_linear_XtWX(Node, Query)
{
  1. Compute Wmin(Node, Query) and Wmax(Node, Query);
  2. If ( Wmax - Wmin ) < Threshold
     Then
         Node->XtWX = 0.25 * (Wmax + Wmin)^2 * Node->XtX;
     Else
        (Node->Left)->XtWX = calc_XtWX(Node->left, Query);
        (Node->Right)->XtWX = calc_XtWX(Node->right, Query);
        Node->XtWX = (Node->Left)->XtWX + (Node->Right)->XtWX;
  3. Return result;
}
```

Figure 6-2: Using divide-and-conquer algorithm to calculate XtWX of a node.

the root downward to the leaves, to calculate $(X^TWX)_{Root}$, as well as $(X^TWY)_{Root}$, then we can do the locally weighted linear regression.

Concerning the threshold in Figure 6-2, a simple way is to assign a fixed one, ε, and see if $W_{max} - W_{min} < \varepsilon$. However, this is dangerous. Suppose a query is far away from all the memory data points, then even the root node of the kd-tree may satisfy $w_{max} - w_{min} < \varepsilon$, so that all the memory data points have the same weight, $0.5 \times (w_{max} + w_{min})$. This means that the prediction of the output of the query will be equal to the mean value of all the memory data points' outputs, i.e.,

$$\hat{y}_q = \left(\sum_{i=1}^{N} y_i \right) / N .$$

This may be wildly different from the non-approximate linear regression without kd-tree, which takes the prediction as an extrapolation of the linear function fitting those memory data points, referring to Figure 6-3,

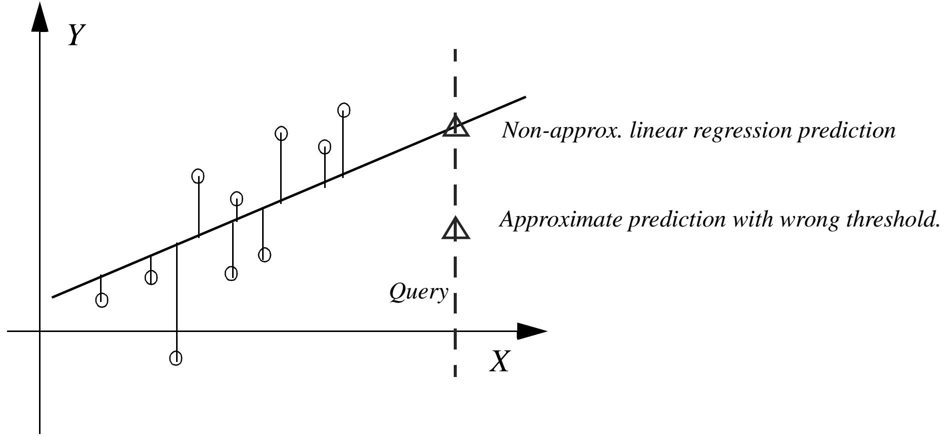

Figure 6-3: The danger of a wrong threshold of the cutoff condition.

This problem can be solved by setting ε to be a fraction of the total sum of weights involved in the regression: $\varepsilon = \tau \times \sum_{k=1}^{N} w_k$ for some small fraction τ. So we would then like to cutoff if and only if, $w_{max} - w_{min} < \tau \times \sum_{k=1}^{N} w_k$. But we do not know the value of $\sum_{k=1}^{N} w_k$ before we begin the prediction, and computing it would not be desirable (cost *O(N)*). Instead, we estimate a lower bound on $\sum_{k=1}^{N} w_k$. If, during the computation so far, we have accumulated sum-of-weights, w_{sofar}, and if currently we are visiting the *Node*'th node in the kd-tree and there are N_{Node} within this node, then,

$$w_{SoFar} + N_{Node} w_{min} \leq \sum_{k=1}^{N} w_k .$$

Therefore, the improved cutoff condition is to judge if,

$$w_{max} - w_{min} < \tau(w_{SoFar} + N_{Node} w_{min}) . \qquad (6\text{-}3)$$

6.3 Technical details

There are several details which we summarize briefly here,

- To ensure numerical stability of this algorithm, all attributes must be pre-scaled to a hypercube centered around the origin.
- The cost of building the tree is $O(M^2 N + N \log N)$, where M is the input space's dimensionality plus 1, and N is the number of data points in memory. It can be built lazily, (growing on-demand as queries occur) and data points can be added in $O(M^2 \times \text{Tree depth})$ time, though occasional rebalancing may be needed. The tree occupies $O(M^2 N)$ space. Huge memory savings are possible if nodes with fewer than M data points are not split, but instead retain the data points in a linked list.
- Instead of always searching the left child first it is advantageous to search the node closest to x_q first. This strengthens the w_{SoFar} bound.
- Ball trees [Omohundro, 91] plays a similar role to kd-trees used for range searching, but it is possible that a hierarchy of balls, each containing the sufficient statistics of data points they contain, could be used beneficially in place of the bounding boxes we used.
- The algorithms can be modified to permit the k nearest neighbors of x_q to receive a weight of 1 each no matter how far they are from the query. This can make the regression more robust.

6.4 Empirical Evaluation

We evaluated five algorithms for comparison.

First of all, we examined prediction on a dataset ABALONE from UCI repository, with 10 inputs and 4177 data points; the task was to predict the number of rings in a shellfish. In these experiments we removed a hundred data points at random as a testset, and examined each algorithm performing a hundred predictions; all variables were scaled to [0..1], and a kernel width of 0.03 was used. As Table 6-2 shows, the **Regular** method took almost a second per predic-

Table 6-1: Five linear regression algorithms

Regular	Direct computation of X^TWX as $\sum_{k=1}^{N} w_k^2 x_k x_k^T$.
Regzero	Direct computation of X^TX with an obvious and useful tweak. Whenever $w_k = 0$, do not bother with $O(M^2)$ operation of adding $w_k x_k x_k^T$.
Tree	The near-exact tree based algorithm. (we set $\tau = 10^{-7}$).
Approx.	The approximate tree-based algorithm with $\tau = 0.05$.
Fast	A wildly approximate tree-based algorithm with $\tau = 0.5$. This gives an extremely rough approximation to the weight function.

tion. **Regzero** saved 20% of that. **Tree** reduced **Regular**'s time by 50%, producing identical predictions (shown by the identical mean absolute errors of **Regular**, **Regzero**, and **Tree**). The **Approx.** algorithm gives an eighty-fold saving compared with **Tree**, and the **Fast** algorithm is about three times faster still. What price do **Approx.** and **Fast** pay in terms of predictive accuracy? Compare the standard error of the dataset (2.65 if the mean value of the training data points' outputs was always given as the predicted value) against **Tree**'s error of 1.65, **Approx.**'s error of 1.67, and **Fast**'s error of 1.71. We notice a small but not insignificant penalty relative to the percentage variance explained.

Table 6-2: Costs and errors predicting the ABALONE dataset

	Regular	**Regzero**	**Tree**	**Approx.**	**Fast**
Millisecs per prediction	980	800	460	5.7	1.7
Mean absolute error	1.65	1.65	1.65	1.67	1.71

The above results are from one run on a testset of size 100. Are they representative? Table 6-3 should reassure that reader, containing averages and confidence intervals from 20 runs with different randomly chosen testsets. The bottom row shows that the error of **Approx.** and **Fast** relative to the **Regular** algorithm is confidently estimated as being small.

Table 6-3: Millisecs to do the predictions, errors of the predictions, and errors relative to Regular.

Algorithms:	Regular	Regzero	Tree	Approx.	Fast
Millisecs	982.0±2.5	814.±3.3	468.±0.8	6.00±0.2	1.70±0.04
Abs. Error Mean	1.534±0.062	1.534±0.062	1.534±0.062	1.536±0.061	1.556±0.063
Excess error compared w/ **Regular**	0±0	0±0	0±0	0.023±0.034	0.032±0.032

Table 6-4: Performance on 5 UCI datasets and one robot dataset. All use locally weighted linear regression with kernel width 0.03

		Regular	Regzero	Tree	Approx.	Fast
Heart, 3-d, 170 datapnts. StdErr. 0.43	Cost	42.16	32.95	21.23	18.93	14.12
	Error	0.27	0.28	0.28	0.28	0.28
Pool, 3-d, 153 datapnts. StdErr. 2.21	Cost	34.65	33.45	22.33	4.41	0.80
	Error	0.63	0.63	0.63	0.63	0.62
Energy, 5-d, 2344 datapnts. StdErr. 286.07	Cost	535.87	484.30	323.37	5.11	1.10
	Error	11.93	11.93	11.93	15.15	21.60
Abalone, 10-d, 4077 datapnts. StdErr. 2.66	Cost	964.00	806.00	469.00	5.80	1.70
	Error	1.65	1.65	1.65	1.67	1.71
MPG, 9-d, 292 datapnts. StdErr. 6.82	Cost	70.10	55.18	34.35	11.61	2.00
	Error	1.92	1.92	1.92	1.92	1.93
Breast, 9-d, 599 datapnts. StdErr. 0.3	Cost	143.40	126.18	59.88	13.82	6.21
	Error	0.03	0.03	0.03	0.03	0.02

We also examined the algorithms applied to a collection of five UCI-repository datasets and one robot dataset (described in [Atkeson et al., 97]). Table 6-4 shows results in which all datasets had the same local model: locally weighted linear regression with a kernel width of 0.03 on the unit-scaled input attributes. Table 6-5 shows the results on a variety of different

Table 6-5: Same experiments, but with a variety of models. The models were selected by cross-validation depending on the specific domains.

		Regular	Regzero	Tree	Approx.	Fast
Heart, Kernel regress. kw = 0.015	Cost	37.86	25.84	14.32	13.42	0.50
	Error	0.22	0.22	0.22	0.22	0.24
Pool, Loc. wgted quad. regress., kw = 0.06	Cost	36.05	35.95	25.43	8.12	1.20
	Error	0.63	0.63	0.63	0.63	0.62
Energy, LW Quad. regress. without cross terms	Cost	546.48	356.12	202.29	25.53	1.60
	Error	6.12	6.12	6.12	6.03	7.50
Abalone, LW Linear regress. ignore 1 input.	Cost	958.90	717.34	203.91	2.35	1.40
	Error	1.33	1.33	1.33	1.33	1.34
MPG, Using all inputs but only has three in the dist. metrics	Cost	66.79	54.18	8.41	1.70	1.20
	Error	1.95	1.95	1.95	1.94	1.92
Breast, Only use five out of ten inputs.	Cost	44.06	43.96	2.20	2.20	0.50
	Error	0.01	0.01	0.01	0.01	0.02

local polynomial models. The pattern of computational savings without serious accuracy penalties is consistent with our earlier experiment.

The above examples all have fixed kernel widths. There are datasets for which an adaptive kernel-width (dependent on the current x_q) are desirable. At this point, two issues arise: the statistical issue of how to evaluate different kernel widths (for example, by the confidence interval width on the resulting prediction, or by an estimate of local variance, or by an estimate of local data density) and the computational cost of searching for the best kernel width for our chosen criterion. Here we are interested in the computational issue and so we resort to a very simple criterion: the local weight, $\sum w_i$.

Table 6-6: Prediction-time optimization of kernel width.

Using fixed Kernel width		Using variable Kernel width		Using variable Kernel width Goal weight is 8.0		
Kernel width	Mean error	Goal weight	Mean error	Algorithm	Mean error	Millisecs per prediction
0.25000	0.41	64	0.19	**Regular**	0.104	2000
0.12500	0.24	32	0.13	**Regzero**	0.104	1400
0.06250	0.24	16	0.11	**Tree**	0.104	395
0.03125	0.22	8	0.10	**Approx.**	0.103	181
0.01562	0.29	4	0.10	**Fast**	0.107	165
0.00781	0.37	2	0.11			
0.00391	0.41	1	0.15			
0.00195	0.51	0.5	0.51			

We artificially generated a dataset with 2-dimensional inputs, for which a variable kernel width is desirable. When evaluated on a testset of 100 data points we saw that no fixed kernel width did better than a mean error of 0.20 (Table 6-6, first two columns). We chose the simplest imaginable adaptive kernel-width prediction algorithm: on each top level prediction make eight inner-loop predictions make eight inner-loop predictions, with the kernel widths $\{2^{-2}, 2^{-3}, ..., 2^{-9}\}$; then choose to predict with the kernel width that produces a local weight $\sum w_i$ closest to some fixed goal weight. For dense data a small kernel width will thus be chosen, and for sparse data the kernel will be wide. The results are striking: The middle two columns of Table 6-6 reveal that for a wide range of goal-weights a testset error of 0.10 is achieved. At the same time, as the rightmost three columns show, the approximate methods continue to win computationally.

6.5 Kd-tree for logistic regression

Recall in Chapter 5, locally weighted logistic regression is to approximate the parameter vector β in the following formula,

$$P(y_q = 1 | S_p, x_q) = \pi_q = \frac{1}{1 + \exp(-[1, x_q^T]\beta)} \tag{6-4}$$

To do so, we should follow the Newton-Raphson recursion:

$$\hat{\beta}_{(r+1)} = \hat{\beta}_{(r)} + (X^T W X)^{-1} X^T W e \tag{6-5}$$

Suppose there are N training data points in memory, each training data point consists of a d-dimensional input vector and a boolean output. X is a $N \times (1 + d)$ matrix. The i'th row of X matrix is $(1, x_i^T)$. And W is a $N \times N$ diagonal matrix, whose i'th element is $W_i = w_i^2 \pi'_i$, where π'_i is a scalar, which is the derivative value of π_i with respect to the current estimate of β at the query x_q:

$$\pi_i' = \left. \frac{\exp(-[1, x_i^T]\beta)}{\{1 + \exp(-[1, x_i^T]\beta)\}^2} \right|_{\beta = \hat{\beta}_{(r)}} \tag{6-6}$$

For example, when a training data point's input is $x_i = [2]$, while the current estimate of β is $[0.5, 1]^T$, then π_i' is equal to *0.07*. As mentioned above, the i'th element of W diagonal matrix is also decided by the weight, w_i, which is a function of the distance from the i'th training data point to the query x_q. The last item, e is the ratio of $y_i - \pi_i$ to π'_i, i.e. $e = (y_i - \pi_i)/\pi_i'$. Newton-Raphson starts from a random estimate of β, usually we assign $\hat{\beta}_{(0)}$ to be zero vector. Although it is not strictly proved, usually with no more than 10 loops, the recursive process comes to a satisfactory estimate of β.

Now, our task is that what information we should cache into the nodes of kd-tree, so that we can approximate X^TWX and X^TWe quickly without any significant loss of the accuracy. The most important characteristic of the cached information is that it must be independent from any specific query, because we want to exploit the same cached information to handle various queries.

Our solution is to cache $\sum_{i \in \text{Node}} x_i$, $\sum_{i \in \text{Node}} x_i x_i^T$ and $\sum_{i \in \text{Node}} y_i x_i$, which are expressed as $\mathbf{1}^T X$, X^TX, and X^TY, too.

To calculate $(X^TWX)_{Node}$ of a particular kd-tree node, we can either do it precisely following its definition:

$$(X^TWX)_{Node} = \sum_{i \in \text{Node}} w_i^2 \pi_i' x_i x_i^T \tag{6-7}$$

where π_i' is the derivative value of the logistic function defined in Equation 6-6, w_i is the weight of the i'th data point with respect to the query.

When all the weights, w_i, $i \in Node$, are near identical, and so are the derivative values, π_i', $i \in Node$, we can approximate $(X^TWX)_{Node}$ as,

$$(X^TWX)_{Node} \approx \overline{w^2}\bar{\pi}' \sum_{i \in Node} x_i x_i^T = \overline{w^2}\bar{\pi}' X^T X \tag{6-8}$$

There are three scenarios that the weights, w_i, within a kd-tree node, are near identical, referring to Section 6-2. Hence, the cutoff condition and the threshold discussed in Section 6-2 should be employed for logistic regression, too. In other words, to make Equation 6-8 hold, the concerned kd-tree node should satisfy:

$$w_{max} - w_{min} < \tau(w_{SoFar} + N_{Node} w_{min}) \tag{6-9}$$

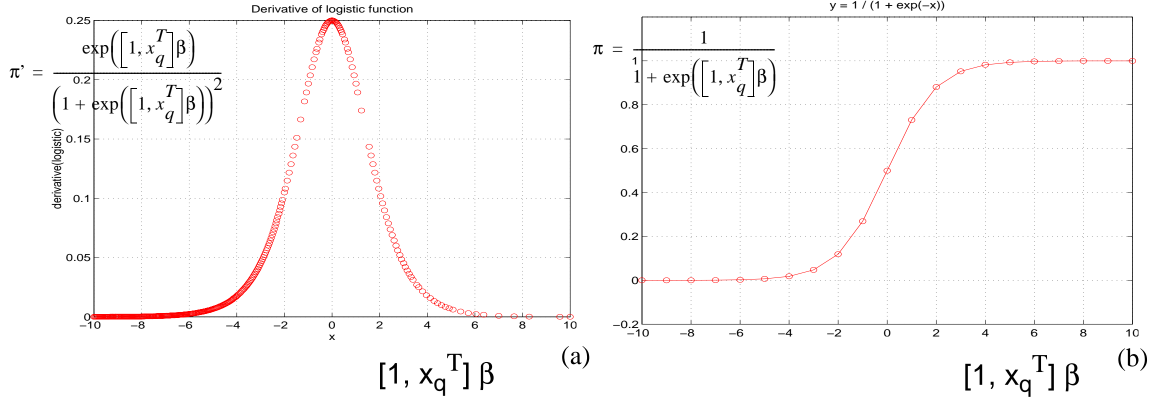

Figure 6-4: (a) The derivative function of logistic, which has symmetric two tails close to zero and a peak in the center. (b) The logistic function which is monotonic between 0 and 1.

To tell if the derivative values, π_i', $i \in Node$, do not differ too much, it looks that we can use a simple fixed threshold, ε_1:

$$\pi'_{max} - \pi'_{min} < \varepsilon_1$$

However, it is not easy to find the upper bound and lower bound of π_i'. Referring to Figure 6-4 (a) and (b), if Equation 6-10 holds,

$$\pi_{max} - \pi_{min} < \varepsilon_1 \tag{6-10}$$

the gap between π'_{max} and π'_{min} must be small, too. Since logistic function is monotonic, usually we can rely on the calculation of the logistic function values at the corners of the hyper-rectangle region in the input space represented by the kd-tree node, to find π_{max} and π_{min}.

Therefore, given a specific query x_q in conjunction with a certain estimate of β, to calculate $(X^TWX)_{Root}$ efficiently, we can recursively sum the two X^TWX's of the child nodes from the root on the top of the kd-tree downward to the leaves, in a way similar to that of locally weighted linear regression described in Figure 6-2. Sometimes the recursion can be cut off if both the two conditions in Equation 6-9 and 6-10 are satisfied, then the X^TWX of that node can be approxi-

mated as $\overline{w^2}\bar{\pi}'X^TX$. Thus, we need to cache X^TX into each node of the kd-tree before any query occurs.

More interestingly, notice that in Figure 6-4(a), the derivative of logistic function with respect to the scalar, $[1, x_q^T]\beta$, has a pair of long tails close to zero. That means, when the scalar $[1, x_q^T]\beta$ deviates from the origin, the derivative value, π', approaches zero quickly; and when the π'_{max} value of a kd-tree's node is near zero, it is unnecessary to calculate the X^TWX matrix of that node, because it must be a zero matrix according to $(X^TWX)_{Node} \le \overline{w^2}\bar{\pi}'_{max}(X^TX)_{Node}$.

Now, let's consider X^TWe of the training data points within a kd-tree's node, according to its definition,

$$
\begin{aligned}
(X^TWe)_{Node} &= \left(X^TW\left(\frac{Y-\pi}{\pi'}\right)\right)_{Node} = \sum_{i \in Node} x_i w_i^2 \pi_i' \left(\frac{y_i - \pi_i}{\pi_i'}\right) \\
&= \sum_{i \in Node} w_i^2 y_i x_i - \sum_{i \in Node} w_i^2 \pi_i x_i
\end{aligned}
\tag{6-11}
$$

In case the following two conditions are satisfied: (1) all the individual weights, w_i, $i \in Node$, are near identical, (2) all the predictions, π_i, $i \in Node$, are near identical, X^TWe can be approximated as,

$$
\begin{aligned}
(X^TWe)_{Node} &\approx \bar{w}^2_{Node} \sum_{i \in Node} y_i x_i - \bar{w}^2_{Node}\bar{\pi}_{Node} \sum_{i \in Node} x_i \\
&= \bar{w}^2_{Node} XY - \bar{w}^2_{Node}\bar{\pi}_{Node}\mathbf{1}^T X
\end{aligned}
\tag{6-12}
$$

Concerning the first cutoff condition related to the weights, we can use Equation 6-9 again to tell if the situation happens. Concerning the second cutoff condition about the prediction π_i, we can pre-define a fixed threshold, ε_2, to see if the following relationship is satisfied,

$$
\pi_{max} - \pi_{min} < \varepsilon_2 \tag{6-13}
$$

This cutoff condition is the same as Equation 6-10; furthermore, usually threshold ε_2 can be assigned to be equal to threshold ε_1. Referring to Figure 6-4(b), the function curve of π_i becomes flat when $[1, x_q^T]\beta$ deviates from the origin. Hence, there should be many chances for Equation 6-13 to hold. To find π_{max} and π_{min}, we can calculate the π values at the corners of the hyper-rectangular partition of the input space which the kd-tree node corresponds to.

In summary, to quickly approximate X^TWX and X^TWe, first of all, we should calculate $\mathbf{1}^TX$, X^TX, and X^TY for each kd-tree node respectively, and cache them into each node in conjunction with the number of data points within the node, `num`, `split_d` and `split_v`. When a query occurs, we follow a recursive algorithm similar to that of Figure 6-2, except that the cutoff conditions are different. The pseudo-code of the recursive algorithm for logistic regression is listed as Figure 6-5.

6.6 Empirical evaluation

In this section, we want to evaluate the performance of cached kd-tree's locally weighted logistic regression in two aspects: (1) how fast is it in comparison with the non-approximate locally weighted logistic regression? (2) how much does it lose in the accuracy?

We used again the four datasets from the UCI data repository which have been used in Section 4.4. Similar to the experiments we have done in Section 4.4, we shuffled the datasets five times each. Every time, we selected one third of the data points as the testing dataset, used the remaining two-thirds of the dataset as the training dataset. For every data point in the testing dataset, we assigned the input as a query, used locally weighted logistic regression based on the training dataset to predict its output, and compared the prediction with the real output of the data point to see if locally weighted logistic regression did correct job. We defined the error rate as the ratio of the number of wrong predictions to the number of total testing data points. Hence, the

```
calc_logistic_XtWX(Node, Query, est_Beta, W_SoFar)
{
  1. Compute Wmin(Node, Query) and Wmax(Node, Query);
  2. Computer dev_Pi_min(Node, est_Beta), dev_Pi_max(Node, est_Beta);
  3. If ( Wmax - Wmin ) < τ * (W_SoFar + Node->num * Wmin)
          and ( Pi_max - Pi_min ) < ε
     Then    Node->XtWX = 0.125 * (Wmax + Wmin)^2
                     * (dev_Pi_max + dev_Pi_min) * Node->XtX;
     Else
        (Node->Left)->XtWX =
            calc_logistic_XtWX(Node->left, Query, est_Beta, W_SoFar);
        (Node->Right)->XtWX = calc_logistic_XtWX(Node->right, ...);
        Node->XtWX = (Node->Left)->XtWX + (Node->Right)->XtWX;
        Update W_SoFar to include 0.25 * (Wmax + Wmin)^2;
  4. Return Node->XtWX;
}

calc_logistic_XtWe(Node, Query, est_Beta, W_SoFar)
{
  1. Compute Wmin(Node, Query) and Wmax(Node, Query);
  2. Computer Pi_min(Node, est_Beta), Pi_max(Node, est_Beta);
  3. If ( Wmax - Wmin ) < τ * (W_SoFar + Node->num * Wmin)
          and ( Pi_max - Pi_min ) < ε
     Then    Node->XtWe = 0.25 * (Wmax + Wmin)^2 * ( Node->XtY
                     - 0.5 * (Pi_max + Pi_min) * Node->1tX );
     Else
        (Node->Left)->XtWe =
             calc_logistic_XtWe(Node->left, Query, est_Beta, W_SoFar);
        (Node->Right)->XtWe = calc_logistic_XtWX(Node->right, ...);
        Node->XtWX = (Node->Left)->XtWe + (Node->Right)->XtWe;
        Update W_SoFar to include 0.25 * (Wmax + Wmin)^2;
  4. Return Node->XtWe;
}
```

Figure 6-5: Using the cached information of kd-tree to quickly approximate the XtWX and XtWe for locally weighted logistic regression.

lower the error rate, the more accurate the locally weighted logistic regression algorithms are. Since for every raw UCI dataset, we shuffled it for five times, thus we got five error rates. In Table 6-7, we listed the mean values of the error rates in conjunction with their standard deviations. In this way, we want to reassure the readers the representativeness of our results.

The first two rows of Table 6-7 are the performance of the regular locally weighted logistic regression without the help of cached kd-tree. As we expected, the error rates (in the second row) are exactly the same as those in Table 4-1. The first row recorded the milliseconds it took the regular locally weighted logistic regression to do one prediction for each datasets. As we have noticed, the computational cost varies a lot from 119.20 to 880.20. That is because the datasets have various dimensionalities of the input space which range from 6 to 34, also because the sizes of the training datasets differ a lot from 230 to 512.

Table 6-7: Performance on 4 UCI datasets

		Ionos. 234 datapnts 34 dim	**Pima** 512 datapnts 8 dim	**Breast** 191 datapnts 9 dim	**Bupa** 230 datapnts 6 dim
Non-approx.	Cost	880.20±5.63	263.20±1.48	119.20±7.85	548.10±4.47
	Error (%)	13.0±0.4	22.5±2.8	3.1±0.7	31.0±2.7
Kd-tree	Cost	906.20±11.19	5.40±0.13	8.28±1.09	5.03±0.04
	Error (%)	7.9±2.8	23.4±3.0	3.1±1.2	31.7±2.2
Cost gain		0.971	48.75	36.36	103.22
Accuracy loss		-38.93%	4.00%	0.0%	2.26%

The third and the fourth rows show the performance of the cached kd-tree's locally weighted logistic regression[1]. We expected that the improved logistic regression was much faster than the regular one while it did not lose too much in the accuracy. To make the comparison easier to follow, in the fifth row we calculated the multiplications of the costs of the regular logistic

regression to those of the kd-tree's. As we see in the table, "Bupa" dataset, which is of low dimensionality with fairly small number of data points, benefited the most from the cached kd-tree: the efficiency improved more than 100 times. "Breast" dataset has a medium dimensionality and the number of data points is small. But still, the cached kd-tree improved the efficiency of locally weighted logistic regression 36 times. "Pima" consists of more data points, so it is not surprising that its multiplication is higher than that of "Breast"'s. "Ionos." is a special dataset because its dimensionality is high. In this case, cached kd-tree does not help to save the computational cost, instead it slightly enlarges the cost.

However, an interesting thing is that cached kd-tree improved the accuracy of locally weighted logistic regression applied to the "Ionos." dataset: the error rate dropped from 13.0% to 7.9%, in other words, the accuracy improved 38.93%, as shown in the last row in the table. Other datasets like "Pima" and "Bupa" did lose some accuracy, but not significantly.

6.7 Summary

In Chapter 5, we explored the use of kd-trees with some cached information, and we found improvements in the efficiency of kernel regression. In this chapter, we discussed how to cache different information into the kd-tree's node so as to improve the efficiency of locally weighted linear regression and locally weighted logistic regression. We found that for different memory-based learning, the cached information is different. Consequently, the cutoff thresholds should also be modified. Cached kd-trees can help both locally weighted linear regression and locally weighted logistic regression improve their computational efficiency, and at the same time not sacrifice their accuracy too much. This contribution is more significant when the size of the training dataset becomes larger. The limitation of cached kd-tree is that when the input space's

1. There are several control knobs for cached kd-tree's locally weighted logistic regression: Kernel width (*kw*), the fraction parameter for the weight's cutoff (τ), the fixed thresholds for the derivative and the prediction ($\varepsilon 1$ and $\varepsilon 2$). We found that the prediction accuracy is not very sensitive to $\varepsilon 1$ and $\varepsilon 2$, so we set both of them as 0.01.τ is also assigned to be 0.01. But Kernel width (*kw*) varies from dataset to dataset, tuned up by cross-validation.

dimensionality is higher than 10, a kd-tree cannot help to improve the efficiency too much. Further research needs to be done combat the curse of dimensionality.