The following is from Chapter 3 of the book *Expert Trading Systems: Modeling Financial Markets with Kernel Regression*, Wiley, 2000.

Chapter 3: Kernel Regression

3.1 The Basic Concept

Kernel regression is one class of data modeling methods that fall within the broader category of *smoothing* methods. The general purpose of smoothing is to find a line or surface which exhibits the general behavior of a dependent variable (lets call it ***Y***) as a function of one or more independent variables. No attempt is made to fit ***Y*** exactly at every point. If there is only one independent variable, then the resulting *smoothing* is a line. If there is more than one independent variable, the *smoothing* is a surface. Smoothing methods that are based upon a mathematical equation to represent the line or surface are called parametric methods. On the other hand, data driven methods that only smooth the data without trying to find a single mathematical equation are called nonparametric methods. Kernel regression is a nonparametric smoothing method for data modeling.

The distinguishing feature of kernel regression methods is the use of a *kernel* to determine a weight given to each data point when computing the smoothed value at any point on the surface. There are many ways to choose a kernel. Wolfgang Hardle reviews the relevant literature in his book on this subject [Ha90]. Another overview of the subject by A. Ullah and H. D. Vinod is included in Chapter 4, Handbook of Statistics 11 [Ul93].

When using data to create models, it is useful to separate the data into several categories:

1) *Learning* data
2) *Testing* data
3) *Evaluation* data (i.e., Reserved data for final evaluation)

If the amount of available data is small, then there are strategies for using all the data records to create a model. For such cases, the number of *learning* points is equal to the total number of data points and the number of *testing* and *evaluation* points are zero. When modeling financial markets this is rarely the case. There is almost always enough data for both learning and test data sets and usually enough to leave some for final evaluation. The usual strategy is to divide the data with *nlrn*, *ntst* and *nevl* points assigned to the three data sets. For various subspaces of the candidate predictor space, the *nlrn* learning points are used to make predictions for the *ntst* testing points and then some measure of performance is computed. One iterates through spaces following a searching strategy. Only if the final measured performance meets the modeling objectives, does one then use the remaining *nevl* points for final *out-of-sample* testing.

To illustrate the procedure for a single one dimensional space, consider Table 3.1.1. In this table values are included for the dependent variable ***Y*** and a single independent variable ***X***.

Point	Data Set	*X*	*Y*
1	Learning	1	12.0
2	Learning	3	18.0
3	Learning	5	20.0
4	Learning	7	17.0
5	Test	2	14.0
6	Test	4	18.5
7	Test	6	19.0

Table 3.1.1 7 Data Points: 4 in the Learning Set and 3 in the Test Set

In the table we see that ***nlrn*** is 4 and ***ntst*** is 3. The 4 learning points are to be used to smooth the data in such a way as to estimate the ***Y*** values for the 3 test points. Since ***Y*** values are already included for the 3 test points we will be able to compare the estimated values with the actual values.

There are many methods for performing this task but the following discussion is limited to a single variation of kernel regression smoothing. The first decision that must be made is the choice of a kernel. Hardle discusses many alternatives but for this example a simple exponential kernel is used:

$$w(x_i, x_j, k) = e^{-k D_{ij}^2} \qquad \textbf{(3.1.1)}$$

In this equation the kernel $w(x_i, x_j, k)$ is the weight applied to the i^{th} learning point when estimating the value of ***Y*** for the j^{th} test point. The parameter ***k*** is called the *smoothing parameter* and D_{ij}^2 is the squared distance between the learning and test points. If ***k*** is assigned a value of 0 then all points are equally weighted. As ***k*** increases, the nearer points are assigned greater weights relative to points further away from the j^{th} test point. As ***k*** approaches infinity, the relative weight of the nearest point becomes infinitely greater relative to all other points.

The simplest kernel regression paradigm is what I will call the *Order 0 algorithm*:

$$y_j = \frac{\sum_{i=1}^{nlrn} w(x_i, x_j, k) Y_i}{\sum_{i=1}^{nlrn} w(x_i, x_j, k)} \qquad \textbf{(3.1.2)}$$

In the statistical literature this equation is often referred to as the *Nadaraya-Watson estimator* [Ha90, Ul93]. In this equation y_j is the value of Y computed for the j^{th} test point. The values Y_i are the actual values of Y for the learning points. This simple algorithm computes y_j as a weighted average of the Y values of the learning points. As an example, consider only the test point at x=2 (i.e., point 5) in Table 3.1.1. In Table 3.1.2 the kernels required in the calculation are included for k=1 and k=0.1:

Point	*X*	*Y*	D_{i5}^2	*w* for k=1	*w* for k=0.1
1	1	12	1	0.3679	0.9048
2	3	18	1	0.3679	0.9048
3	5	20	9	0.0001	0.4066
4	7	17	25	1.4e-11	0.0821

Table 3.1.2 Computed values of $w(x_i, x_5, k)$ for k=1 and k=0.1

From this table and from Equation 3.1.2 we can compute two values of y_5. For k=1 we obtain a value of 15.008 which is very close to the average value of points 1 and 2. The reason why the calculation is dominated by these two points is that the weights for these points are much greater than the weights for points 3 and 4. For k=0.1, point 3 and 4 have a significant influence upon the calculation and the resulting value is 15.956.

Results for all 3 test points and for both values of k are included in Table 3.1.3:

POINT	*X*	*Y*	*y (k=1)*	$(Y-y)^2$	*y (k=0.1)*	$(Y-y)^2$
5	2	14.0	15.008	1.0017	15.956	3.8256
6	4	18.5	18.998	0.2485	17.605	0.8012
7	6	19.0	18.500	0.2500	18.179	0.6734
Sum	Not Used	51.5	Not Used	1.5002	Not Used	5.3002

Table 3.1.3 Values of *y* and $(Y-y)^2$ for the test points

The values of *y* can be used to compare the two alternatives: k=1 and k=0.1. If we choose VR (i.e., Variance Reduction) as the modeling criterion, then Equation 1.4.1 is used. In this table we use the notation *y* instead of ***Ycalc*** as used in Equation 1.4.1. The sums of $\mathbf{(Y(i) - Ycalc(i))^2}$ are 1.5002 and 5.3002 for k=1 and k=0.1 respectively. The value of ***Yavg*** is 51.5 / 3 = 17.167 and the sum of $\mathbf{(Y(i) - Yavg)^2}$ is therefore 15.167. The values of VR are 100*(1 – 1.5002/15.167) =

90.1 for k=1 and 65.1 for k=0.1. With so few learning and test points there is really no significance to these results. They do, however, illustrate the basic concept.

It should be emphasized that the values of $\boldsymbol{y}$ (or ***Ycalc***) can be computed at any point and not just at the test points. The smoothed curves generated using $\boldsymbol{k}$=1 and $\boldsymbol{k}$=0.1 are shown in Figure 3.1.1. For this example, we see that the curve generated by kernels using $\boldsymbol{k}$=1 is much closer to the test points then the curve generated using $\boldsymbol{k}$=0.1. In fact, the smoothed $\boldsymbol{k}$=1 curve gives the appearance that it actually passes through the learning points. This is not the case. For example, the value at $\boldsymbol{x}$=1 is 12.11 whereas the actual value of the learning point at $\boldsymbol{x}$=1 is 12.00. The resolution of the figure is just not enough to clearly show this small difference. The differences for the $\boldsymbol{k}$ = 0.1 curve are, however, quite obvious.

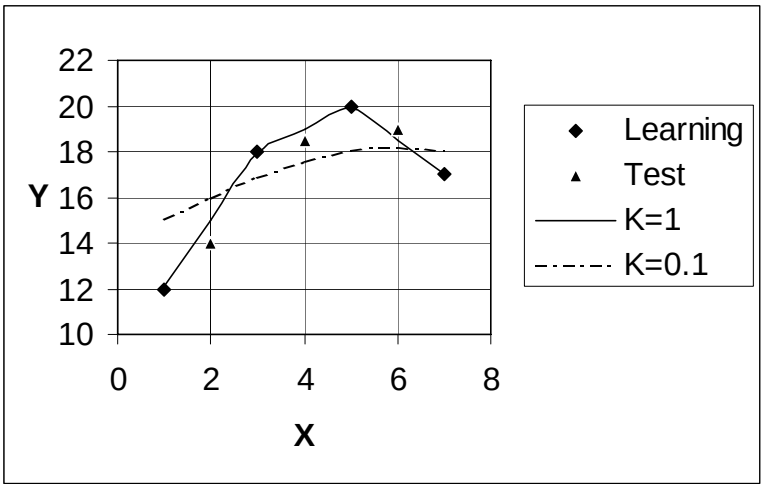

Figure 3.1.1 Smoothed Curves using k=1 and k=0.1

There are many problems associated with the use of Equations 3.1.1 and 3.1.2 and these problems will be considered in the following sections.

3.2 Higher Order Algorithms

In Section 3.1 a simple example based upon Equation 3.1.2 was discussed. This equation will be referred to as the *Order 0 Algorithm*. It uses a polynomial of Order 0 as the fitting function. (A polynominal of Order 0 is just a constant.) Expanding this notation, the *Order 1 Algorithm* thus uses a polynomial of Order 1. Rather than a weighted average, this algorithm is based upon a hyperplane. For a single dimension, the hyperplane is a straight line. In $\boldsymbol{p}$ dimensions it is simply:

$$\boldsymbol{y} = \boldsymbol{a}_1 + \boldsymbol{a}_2 \boldsymbol{x}_1 + \boldsymbol{a}_3 \boldsymbol{x}_2 + \cdots + \boldsymbol{a}_{\boldsymbol{p}+1} \boldsymbol{x}_{\boldsymbol{p}} \qquad \textbf{(3.2.1)}$$

Thus for the *Order 1 Algorithm, p+1* coefficients are required to define the hyperplane. Clearly we can define even higher order algorithms. For example, the *Order 2 Algorithm* is based upon a polynomial of Order 2. For a single dimension it is a parabola:

$$y = a_1 + a_2 x_1 + a_3 x_1^2 \qquad \textbf{(3.2.2)}$$

For two dimensions it includes the interaction term as well as the second order terms:

$$y = a_1 + a_2 x_1 + a_3 x_2 + a_4 x_1^2 + a_5 x_1 x_2 + a_6 x_2^2 \qquad \textbf{(3.2.3)}$$

Going to three dimensions, there is the constant term, 3 first order terms, 3 second order terms and 3 interactions terms. There are therefore 10 coefficient (i.e., $a_1\ a_2, \ldots, a_{10}$). We see that as the dimensionality of the space increases, the number of coefficients required to specify the model increases rapidly. The *Order 2 Algorithm* equation in p dimensions is:

$$y = a_1 + \sum_{j=1}^{p} a_{j+1} x_i + \sum_{j=1}^{p} \sum_{k=j}^{p} b_{jk} x_j x_k \qquad \textbf{(3.2.4)}$$

In general, the number of coefficients required in p dimensional space for the *Order 2 Algorithm* is $\mathbf{1 + p + p*(p+1)/2}$. We could continue to even higher order algorithms, but for applications in which there is a substantial noise component in the signal (like in financial market modeling), there is no sense in going to higher order algorithms.

Once an algorithm has been selected, the coefficients are determined for every test point using the learning data points. Typically the method of linear least squares with weighting is used to determine the N coefficients. We can recast Equations 3.2.1 and 3.2.4 as follows:

$$y = \sum_{j=1}^{N} A_j g_j(X) \qquad \textbf{(3.2.5)}$$

Comparing Equations 3.2.1 and 3.2.5, we see that for the *Order 1 Algorithm*, A_j is just a_j. Comparing Equations 3.2.4 and 3.2.5, the connection is less obvious for *Order 2*. For example, for 2 dimensions (i.e., p=2), b_{11} is renumbered as A_4, b_{12} is renumbered as A_5, and b_{22} is renumbered as A_6. In Equation 3.2.5, $g_j(X)$ is a function of the vector X. For example, for Equation 3.2.1 with p=3 we see that g_1=1, $g_2=x_1$, $g_3=x_2$ and $g_4=x_3$. For Equation 3.2.4 with p=3 the first four g_j's are the same as for Equation 3.2.1. The following g_j's are $g_5=x_1^2$, $g_6=x_1x_2$, $g_7=x_1x_3$, $g_8=x_2^2$, $g_9=x_2x_3$ and $g_{10}=x_3^2$. Using Equation 3.2.5 the least square formulation is the same for both *Order 1* and *Order 2 Algorithms*:

$$CA = V \qquad \textbf{(3.2.6)}$$

A derivation of this equation is included in Appendix A. In this equation $\boldsymbol{C}$ is an $\boldsymbol{N}$ by $\boldsymbol{N}$ matrix and $\boldsymbol{A}$ and $\boldsymbol{V}$ are vectors of length $\boldsymbol{N}$. The $\boldsymbol{C}$ matrix is symmetrical and the term $\boldsymbol{C_{jk}}$ is defined as follows:

$$\boldsymbol{C}_{jk} = \sum_{i=1}^{Nlrn} \boldsymbol{w}_i \boldsymbol{g}_j(\boldsymbol{X}_i)\boldsymbol{g}_k(\boldsymbol{X}_i) \qquad \textbf{(3.2.7)}$$

The term $\boldsymbol{w_i}$ is the weight computed for the $\mathbf{i^{th}}$ learning point using, for example, Equation 3.1.1. The term $\boldsymbol{X}_\mathbf{i}$ is the $\boldsymbol{X}$ vector for the $\mathbf{i^{th}}$ learning point. (The reader should note the difference between uppercase $\boldsymbol{X}$ and lowercase $\boldsymbol{x}$. The lowercase $\boldsymbol{x}$ refers to an element of the $\boldsymbol{X}$ vector.) $\boldsymbol{V}$ vector terms are defined as follows:

$$\boldsymbol{V}_j = \sum_{i=1}^{Nlrn} \boldsymbol{w}_i \boldsymbol{g}_j(\boldsymbol{X}_i)\boldsymbol{Y}_i \qquad \textbf{(3.2.8)}$$

In this equation $\boldsymbol{Y_i}$ is the value of $\boldsymbol{Y}$ for the $\mathbf{i^{th}}$ learning point. Once again, the difference between uppercase and lowercase should be noted. Uppercase $\boldsymbol{Y}$ is used for actual values of $\boldsymbol{Y}$ and lowercase $\boldsymbol{y}$ is used for calculated values of $\boldsymbol{Y}$.

To illustrate the process, we can repeat the example from Section 3.1 but using the *Order 1 Algorithm*. Using the data in Table 3.1.1 and the weights as shown in Table 3.1.2 for $\boldsymbol{k}$=1, the $\boldsymbol{C}$ matrix and $\boldsymbol{V}$ vector must first be computed and then the $\boldsymbol{A}$ vector is determined by solving the $\boldsymbol{N}$ linear equations represented by Equation 3.2.6. This example is for a one dimensional space so $\boldsymbol{N}$ is 2. Using Equation 3.2.5, the equation for *y* is:

$$\boldsymbol{y} = \boldsymbol{A}_1 + \boldsymbol{A}_2 \boldsymbol{x} \qquad \textbf{(3.2.9)}$$

From this equation we see that $\boldsymbol{g_1} = 1$ and $\boldsymbol{g_2} = \boldsymbol{x}$. There is no need to use a subscript for $\boldsymbol{x}$ as we are considering the one dimensional case. In the following equations $\boldsymbol{x_i}$ refers to the value of $\boldsymbol{x}$ for the $\mathbf{i^{th}}$ learning point. The equations for the terms of $\boldsymbol{C}$ and $\boldsymbol{V}$ are therefore as follows:

$$\boldsymbol{C}_{11} = \sum_{i=1}^{4} \boldsymbol{w}_i = 0.7359 \qquad \textbf{(3.2.10)}$$

$$\boldsymbol{C}_{12} = \sum_{i=1}^{4} \boldsymbol{w}_i \boldsymbol{x}_i = 1.4721 \qquad \textbf{(3.2.11)}$$

$$\boldsymbol{C}_{22} = \sum_{i=1}^{4} \boldsymbol{w}_i \boldsymbol{x}_i^{\,2} = 3.6819 \qquad \textbf{(3.2.12)}$$

$$\boldsymbol{V}_1 = \sum_{i=1}^{4} \boldsymbol{w}_i \boldsymbol{Y}_i = 11.0389 \qquad \textbf{(3.2.13)}$$

$$V_2 = \sum_{i=1}^{4} w_i x_i Y_i = 24.2924 \qquad \textbf{(3.2.14)}$$

The two equations to be solved are:

$$C_{11} A_1 + C_{12} A_2 = V_1$$
$$C_{21} A_1 + C_{22} A_2 = V_2 \qquad \textbf{(3.2.15)}$$

Since the ***C*** matrix is symmetric, $C_{21} = C_{12}$. Solving 3.2.15, the resulting values of A_1 and A_2 are 9.0034 and 2.9980. Using Equation 3.2.9 with X=2 we get a value of y equal to 14.999 (i.e., 9.0034 + 2*2.9980). The results for all 3 test points are summarized in table 3.2.1.

Point	*X*	*Y*	A_1	A_2	*Ycalc*
5	2	14.0	9.0034	2.9980	14.999
6	4	18.5	15.0005	0.9995	18.999
7	6	19.0	27.4841	-1.4975	18.499

Table 3.2.1 Order 1 Results for 3 Test Points

The point to note in examining Table 3.2.1 is that a separate line (i.e., separate values of A_1 and A_2) are computed for each test point. The differences from point to point are due to the different weights assigned to the learning points based upon the distances from the test points (see Equation 3.1.1). However, if the value of ***k*** in Equation 3.1.1 is 0, then all points are equally weighted and the values of the A_k's are constant for all test points. This point is illustrated in the following example. Consider the following table:

Point	Data Set	*x1*	*x2*	*Y*
1	Learning	1	-6	5
2	Learning	3	8	-11
3	Learning	5	0	9
4	Learning	7	2	-3
5	Learning	9	-5	53
6	Learning	11	7	-57
7	Learning	13	3	-19
8	Learning	15	-1	31
9	Test	2	0	5
10	Test	6	4	-11
11	Test	10	-2	31

Table 3.2.2 11 Data Points: 8 in the Learning Set and 3 in the Test Set

Assuming all learning points are given the same weight (e.g., w_I=1), results for all three algorithms are included in Table 3.2.3:

Point	*x1*	*x2*	*Y*	*y0*	*y1*	*y2*
9	0	5	5	1.000	1.715	5.249
10	6	4	-11	1.000	-15.450	-11.167
11	-2	3	31	1.000	17.450	31.407

Table 3.2.3 Results for Test Points using 3 Algorithms with equal weighting

In this table the ***y0*** values are the values computed using the *Order 0 Algorithm*, and ***y1*** and ***y2*** are the results using the *Orders 1 and 2 Algorithms*.

The results in Table 3.2.3 illustrate several points. Notice that all 3 values of ***y0*** are the same (i.e., exactly 1). The value 1 is just the average value of ***Y*** for the eight learning points included in Table 3.2.2. In other words, if all learning points are equally weighted, *Order 0* yields a single value for all test points. At first glance this appears to be a useless result. However, as we will see in Section 3.3, unit weighting can prove to be quite advantageous, even if the *Order 0 Algorithm* is used.

The results in Table 3.2.3 for the *Orders 1 and 2 Algorithms* (i.e., ***y1*** and ***y2***) do vary from test point to test point even though all learning points were equally weighted. The two equations obtained were:

$$y1 = 0.2848 + 0.7152\, x_1 + 5.0065\, x_2 \qquad \textbf{(3.2.16)}$$

$$y2 = 3.0488 + 1.1200\, x_1 + 0.8039\, x_2 - 0.0100\, x_1^2 - 0.9889\, x_1 x_2 - 0.0037\, x_2^2 \qquad \textbf{(3.2.17)}$$

Note that both of these equations apply to all three test points. Comparing the values of ***y0***, ***y1*** and ***y2*** to ***Y*** in Table 3.2.2, the values of ***y2*** are clearly much close than either the values of ***y0*** and ***y1***. One should not conclude that *Order 2* is always preferable to *Order 0* or *Order 1*. For this particular example the Order 2 polynomial turned out to yield the best results.

3.3 The Bandwidth Concept

If we consider development of a one-dimensional model (i.e., *y* as a function of ***x***), and if we use a kernel-smoothing algorithm then the *bandwidth* concept refers to the definition of the kernel (e.g., Equation 3.3.1). The bandwidth ***h*** is defined as a region around a test point in which only learning points that fall within this region are given nonzero weights. For a test point located at x_j only learning points that fall within the region x_j-***h/2*** to x_j+***h/2*** are used for the estimation of ***y.*** Clearly, if no learning points fall within this region, then *y* cannot be estimated. As a result of this problem, another approach called the *K-nearest neighbor estimate (K-N/N)* is often used [Ha90]. For smoothing problems in which the learning values of ***x*** may be chosen, a constant bandwidth is perfectly acceptable. Even if the ***x*** values cannot be chosen, if there is high data density, then a constant bandwidth is still a reasonable choice. If *K-N/N* is used we can consider it as a variable bandwidth approach. In other words, for every test point, the bandwidth is chosen so that the ***K*** closest learning points are given nonzero weights.

Can we extend the bandwidth concept to higher dimensions? Equation 3.1.1 provides the clue. In this equation the kernel is based upon a squared distance. We can easily define a distance in ***p*** dimensional space and base bandwidth upon this distance. It should be clear that if the ***x*** variables in ***p*** dimensional space have radically different scales, then a straight Cartesian distance is useless. The long dimension would totally dominate the calculation. Typically one uses some sort of normalized distance when considering kernel regression in ***p*** dimensional space (where ***p*** is greater than one). The most common methods of normalizing distances are to divide the actual values for each dimension by either the range or standard deviation of the dimension. In ***p*** dimensional space the value of D_{ij} (i.e., the normalized distance between the j^{th} test point and the i^{th} learning point is computed as follows:

$$D_{ij}^2 = \sum_{d=1}^{p} \left((x_{di} - x_{dj}) / N_d \right)^2 \qquad \textbf{(3.3.1)}$$

The value of D_{ij} is just the square root of D_{ij}^2**.** However, it is usually more advantageous to work with D_{ij}^2 because it eliminates the need for taking square roots. If the numbers of learning and test point are large, then avoiding the square root calculation can save significant compute time. In this equation x_{di} is the $\mathbf{d^{th}}$ dimension of the $\mathbf{i^{th}}$ learning point and x_{dj} is similarly defined for the $\mathbf{j^{th}}$ test point. The N_d term is the normalization constant for the $\mathbf{d^{th}}$ dimension (e.g., the range for dimension ***d***).

The primary reason for introducing the bandwidth concept is to eliminate the effect of very distant points upon the estimate at a given test point. Clearly this can be accomplished by using a large smoothing constant. For example, using Equation 3.1.1 we see that the choice of a large value of ***k*** will also eliminate the influence of distant points. However, this might be counter productive because a large value of ***k*** also reduces the smoothing and in the extreme, the estimates are based primarily on the single nearest neighbor of each test point.

Another reason for using bandwidth (or *K-N/N*) is to save a considerable amount of compute time. Defining the number of learning and test points as N_l and N_t, if all learning points are used to determine all test points, the calculational complexity is $O(N_l * N_t)$. In other words, the time to complete the calculation increases as the product of N_l and N_t. For modeling of financial markets, these numbers can be very large and therefore some approach to reducing calculational complexity is necessary. We will see in Chapter 4 that high performance kernel regression systems are based upon the bandwidth concept.

3.4 Error Estimates

In Sections 3.1 and 3.2 *Order 0, 1 and 2 Algorithms* were described. The errors associated with the predicted values of ***Y*** can easily be computed. Typically, predicted errors are expressed as

σ_y, which is the estimated standard deviation of the prediction. The predicted value of σ_y for *Order 0* is simply the standard deviation of the weighted average:

$$\sigma_y^2 = \frac{\sum_{i=1}^{n} w_i (Y_i - y_j)^2}{(n-1)\sum_{i=1}^{n} w_i} \qquad \textbf{(3.4.1)}$$

In this equation y_j is the predicted value of the $\mathbf{j^{th}}$ test point and is computed as the weighted average of the n values of Y used to make the prediction (i.e., Equation 3.1.2). For every test point there is a value of σ_y. The equation is valid with or without a bandwidth limitation on the choice of data points. Using a bandwidth (or some similar device to reduce the number of data points) reduces the value of n. Notice that as n decreases the term ($n-1$) in the denominator decreases and tends to increase σ_y. However, as n decreases, distant points are rejected and one would expect that the nearer values of Y_i would be closer to the actual value of Y at the test point. Thus one would expect that there is an optimum value of n (or bandwidth) at which the value of σ_y is minimized.

Derivation of Equation 3.4.1 follows naturally from the more general derivation of σ_y for all three algorithms. The three Algorithms are based upon 3 different fitting functions. The fitting function for *Order 0* is a constant:

$$y = a_1 \qquad \textbf{(3.4.2)}$$

Order 1 uses a hyperplane in p dimensional space (Equation 3.2.1):

$$y = a_1 + a_2 x_1 + a_3 x_2 + \ldots + a_{p+1} x_p \qquad \textbf{(3.2.1)}$$

Order 2 uses a complete second order multinomial (Equation 3.2.4):

$$y = a_1 + \sum_{j=1}^{p} a_{j+1} x_i + \sum_{j=1}^{p} \sum_{k=j}^{p} b_{jk} x_j x_k \qquad \textbf{(3.2.4)}$$

All three equations can be recast into the form of Equation 3.2.5:

$$y = \sum_{j=1}^{N} A_j g_j(X) \qquad \textbf{(3.2.5)}$$

For *Order 0*, $N=1$ regardless of the dimensionality of the space. For *Order 1*, $N=1+p$ for a p dimensional space. For *Order 2*, $N=1+p+p*(p+1)/2$. The functions $g_j(X)$ can be determined by comparing Equations 3.4.2, 3.2.1 and 3.2.4 with Equation 3.2.5. Examples for *Order 1* and *2* are included in Section 3.2. For *Order 0* the function g_1 is 1.

In Section 3.2 the values of the A_j's were determined by solving the matrix equation 3.2.6. The elements of the C matrix were determined using Equation 3.2.7. Using this same formulation, Equation 3.1.2 (the order 0 prediction for test point j) can be derived. Since N=1, Equation 3.2.6 is simply a single equation:

$$C_{11} A_1 = V_1 \qquad \textbf{(3.4.3)}$$

where

$$C_{11} = \sum_{i=1}^{n} w_i g_1 g_1 = \sum_{i=1}^{n} w_i \qquad \textbf{(3.4.4)}$$

$$V_1 = \sum_{i=1}^{n} w_i g_1 Y_i = \sum_{i=1}^{n} w_i Y_i \qquad \textbf{(3.4.5)}$$

Substituting 3.4.4 and 3.4.5 into 3.4.3 we solve for A_i. The resulting equation for A_i is the same as Equation 3.1.2 and is the predicted value of Y (according to Equation 3.4.2). Thus the general formulation used in Section 3.2 for *Orders 1* and *2* is also applicable for *Order 0.*

The method of least squares includes a general formulation for σ_y [Wo67, Ga92]:

$$\sigma_y^2 = \frac{S}{n - N} \sum_{j=1}^{N} \sum_{k=1}^{N} g_j(X) g_k(X) C_{jk}^{-1} \qquad \textbf{(3.4.6)}$$

The value of σ_y is thus determined by taking the square root of Equation 3.4.6. In this equation, C^{-1} refers to the inverse matrix of C. The term C^{-1}_{jk} is therefore the term on the j^{th} row of the k^{th} column of the inverse matrix. The symbol S refers to the sum of the residuals and is computed as follows:

$$S = \sum_{i=1}^{n} w_i (Y_i - y_i)^2 \qquad \textbf{(3.4.7)}$$

The values of Y_i are the actual values of Y for the learning points and the values of y_i are the computed values of Y for the learning points (using Equation 3.4.2, 3.2.1 or 3.2.4). We can show that Equation 3.4.6 is equivalent to Equation 3.4.1 for *Order 0* by substituting 3.4.7 into 3.4.6 and noting that N=1 and $g_1(x)$=1. Since N=1, C^{-1}_{11} is a scalar and is just $1/C_{11}$.

For *Orders 1* and *2*, the C matrix is computed in order to determine the values of y_j. The C matrix can be inverted to solve Equation 3.2.6, however this is not the fastest way for solving simultaneous linear equations. There is an additional cost required if this matrix is inverted so in high performance systems, one would generally invert the matrix only for final results in which the values of σ_y are required. For *Order* 1 with p=1 (i.e., one dimensional), N=2. For this case Equation 3.4.6 reduces to the following:

$$\sigma_y^2 = \frac{S}{n-2}(C_{11}^{-1} + 2\,x_j C_{12}^{-1} + x_j^2 C_{22}^{-1}) \qquad \textbf{(3.4.8)}$$

The constant 2 associated with the C^{-1}_{12} term is due to the fact that the C^{-1} matrix is symmetric and therefore $2C^{-1}_{12}$ is used instead of (C^{-1}_{12}+C^{-1}_{21}). From this equation we see that the value of σ_y is different at every x_j (i.e., the x value of the j^{th} test point). For *Order* 2 with p=1 (i.e., one dimensional), N=3. For this case Equation 3.4.6 reduces to the following:

$$\sigma_y^2 = \frac{S}{n-3}(C_{11}^{-1} + 2\,x_j C_{12}^{-1} + 2\,x_j^2 C_{13}^{-1} + x_j^2 C_{22}^{-1} + 2\,x_j^3 C_{23}^{-1} + x_j^4 C_{33}^{-1}) \qquad \textbf{(3.4.9)}$$

For *Order* 1 with p=2 (i.e., two dimensional), N=3. For this case Equation 3.4.6 reduces to the following:

$$\sigma_y^2 = \frac{S}{n-3}\left(\begin{array}{l} C_{11}^{-1} + 2\,x_{1j} C_{12}^{-1} + 2\,x_{2j} C_{13}^{-1} + x_{1j}^2 C_{22}^{-1} + \\ 2x_{1j}\,x_{2j} C_{23}^{-1} + x_{2j}^2 C_{33}^{-1} \end{array}\right) \qquad \textbf{(3.4.10)}$$

Clearly, as p increases, Equation 3.4.6 becomes increasingly cumbersome and there is no justification for expressing the equation explicitly. In any computer code in which this equation is used, a modified version of the loop form as expressed in 3.4.6 is preferable to explicit forms such as 3.4.8 to 3.4.10. The equation can be modified to eliminate the need for computing terms below the diagonal of the matrix.

To illustrate this calculation, consider the data included in Table 3.2.2. This data includes eight learning points and three test points. The data is two dimensional (i.e., p=2). The values of y computed using the three algorithms are included in Table 3.2.3. At this point we will compute the values of σ_y for the test points for each of the three algorithms. To simplify the calculation we will again assume that all learning points are equally weighted (i.e., w_i=1). The three fitting functions for this problem are:

***Order 0*:** $y = A_1$ **(3.4.11)**

***Order 1*:** $y = A_1 + A_2\,x_1 + A_3\,x_2$ **(3.4.12)**

***Order 2*:** $y = A_1 + A_2\,x_1 + A_3\,x_2 + A_4 x_1^2 + A_5\,x_1\,x_2 + A_6 x_2^2$ **(3.4.13)**

For *Order 0*, the C matrix contains only 1 element and is computed using Equation 3.4.4:

$$C_{11} = \sum_{i=1}^{8} w_i = 8 \qquad \textbf{(3.4.14)}$$

The term C^{-1}_{11} is thus 0.125. The value of $\boldsymbol{S}$ is computed using equation 3.4.7 and a result of 7608 (i.e., $(5-1)^2 + (-11-1)^2 + (9-1)^2 \ldots$) is obtained. The value of σ_y for all 3 test points is determined using Equation 3.4.6. Noting that $\boldsymbol{N}=1$ and $\boldsymbol{g_1}=1$, we obtain the following equation:

$$\sigma_y^2 = \frac{\boldsymbol{S}}{\boldsymbol{n}-1}\boldsymbol{C}_{11}^{-1} = \frac{7608}{7} * 0.125 = 135.86 \qquad \textbf{(3.4.15)}$$

The computed value of σ_y is thus 11.65 and is the same for all three test points (since the value of $\boldsymbol{y_j}$ is the same (i.e., 1) for all three points).

For *Order 1*, the $\boldsymbol{C}$ matrix is 3 by 3 and therefore contains 9 elements. The elements are computed according to Equation 3.2.7. Using $\boldsymbol{w_i}=1$, $\boldsymbol{g_1}=1$, $\boldsymbol{g_2}=\boldsymbol{x_1}$ and $\boldsymbol{g_3}=\boldsymbol{x_2}$ the following matrix is obtained:

$$\boldsymbol{C} = \begin{Bmatrix} 8 & 64 & 8 \\ 64 & 680 & 88 \\ 8 & 88 & 188 \end{Bmatrix} \qquad \textbf{(3.4.16)}$$

Inverting this matrix we obtain the C^{-1} matrix:

$$\boldsymbol{C}^{-1} = \begin{Bmatrix} 0.5061 & -0.04773 & 0.0008091 \\ -0.04773 & 0.006068 & -0.0008091 \\ 0.0008091 & -0.0008091 & 0.005663 \end{Bmatrix} \qquad \textbf{(3.4.17)}$$

Since all learning points are equally weighted, one hyperplane is determined and is applicable to all three test points. The coefficients of the plane are included in the equation for $\boldsymbol{y1}$ in Equation 3.2.16 (i.e., 0.2848, 0.7152 and 5.0065). Using this equation $\boldsymbol{S}$ is determined according to Equation 3.4.7:

$$\boldsymbol{S} = \sum_{\boldsymbol{i}=1}^{\boldsymbol{n}} \boldsymbol{w_i}(\boldsymbol{Y_i} - (\boldsymbol{A}_1 + \boldsymbol{A}_2\boldsymbol{x}_1 + \boldsymbol{A}_3\boldsymbol{x}_2))^2 \qquad \textbf{(3.4.18)}$$

The value obtained for $\boldsymbol{S}$ is 3182.3. The values of σ_y can now be computed from Equation 3.4.10. For example for Point 9 in Table 3.2.2 the value of $\boldsymbol{x_1}$ is 2 and the value of $\boldsymbol{x_2}$ is 0. The resulting equation is:

$$\sigma_y^2 = \frac{\boldsymbol{S}}{\boldsymbol{n}-3}(\boldsymbol{C}_{11}^{-1} + 4\boldsymbol{C}_{12}^{-1} + 4\boldsymbol{C}_{22}^{-1}) = \frac{3182.3 * 0.3395}{5} = 216.0 \qquad \textbf{(3.4.19)}$$

The value of σ_y is thus 14.7. The values of σ_y for Points 10 and 11 are determined in a similar manner and both values are 11.56. For both of these points the equations are more complicated than 3.4.19 because the terms containing x_2 must be included.

For *Order 2*, the *C* matrix is 6 by 6. In a similar manner we obtain the following matrix:

$$\boldsymbol{C} = \begin{Bmatrix} 8 & 64 & 8 & 680 & 88 & 188 \\ 64 & 680 & 88 & 8128 & 888 & 1152 \\ 8 & 88 & 188 & 888 & 1152 & 548 \\ 680 & 8128 & 888 & 103496 & 9784 & 10508 \\ 88 & 888 & 1152 & 9784 & 10508 & 4360 \\ 188 & 1152 & 548 & 10580 & 4360 & 8516 \end{Bmatrix} \qquad \textbf{(3.4.20)}$$

The calculation proceeds in a manner similar to the *Order 1* calculation. In place of Equation 3.4.12 in 3.4.18, Equation 3.4.13 is used. The resulting value of $\boldsymbol{S}$ is 4.044. The resulting values of σ_y for the three test points are 1.49, 0.90 and 0.91. These values are considerably less than the values obtained using the *Orders 0* and *1 Algorithms.* This is not a surprising result if one examines Table 3.2.3 and notices how much closer the values of ***y2*** are to the ***Y*** values as compared to ***y1*** and ***y0***. The results are summarized in Table 3.4.1.

Point	*x1*	*x2*	*Y*	σ_y *(Order 0)*	σ_y *(Order 1)*	σ_y *(Order 2)*
9	2	0	5	11.65	14.70	1.49
10	6	4	-11	11.65	11.56	0.90
11	10	-2	31	11.65	11.56	0.91

Table 3.4.1 Values of σ_y for 3 algorithms using data from Table 3.2.2

3.5 Applying Kernel Regression to Time Series Data

In the previous sections kernel regression was applied to data in which there was no special significance to the order in which the records were analyzed. However, in most financial applications the order of the data is extremely important. For example daily data is ordered by date and intra-day data is ordered by date/time. There are other possible orderings of the data. For example, if one wishes to analyze a group of stocks using daily data, one might order the data on the basis of date/stock_code. Once we acknowledge that the order of the data is important, then the choice of learning and test data sets becomes crucial.

The time dimension introduces another level of complexity to the analysis: *how much importance do we attach to recent data records as opposed to earlier records?* Is there a simple way to take this effect into consideration? Common sense leads us to the basic conclusion that if we are to predict a value of ***Y*** at a given time, we should only use learning data from an earlier time[1]. Using this principle, one would choose learning data as the earlier records and then use the later records for testing. But this procedure tends to be overly restrictive. For example, if we

[1] When the amount of data is relatively small or if there is reason to believe that the model is relatively invariant with time, the analyst might choose to use future data to make predictions on the past.

have 10 years of daily data from 1988 through 1998 (approximately 2500 records of data), and if we want to test on about one third of the records, then the last three or four years would be used for testing. But this seems to be unrealistic particularly for the later test data records. We would, for example, be modeling records in 1998 using data from 1988 to about 1994 and not including the more recent records in the modeling process.

There is a simple solution to this problem. All that one must do is to make the learning data set dynamic. In other words, once a record has been tested, it is then available for updating the learning data set prior to the testing of the next record. The analyst can allow the learning data set to grow or alternatively, for each record added, the earliest remaining record in the learning set can be discarded. These two alternatives will be referred to as the *growing* option and the *moving window* option. For cases in which the learning set is not changed, we will refer to this alternative as the *static* option.

To illustrate these three alternatives, consider the data in Table 3.2.2 and the results included in Table 3.2.3. The results in Table 3.2.3 were computed using the *static* option. In other words, learning points 1 through 8 were used to compute predicted values of ***Y*** for test points 9, 10 and 11. The three values included in Table 3.2.3 for each test point are the predicted value using the three algorithms (i.e., *Orders 0, 1* and *2*). We can repeat the calculations using the *growing* and *moving window* options.

The results for the *growing* option are included in Table 3.5.1:

Point	***x1***	***x2***	***Y***	***y0***	***y1***	***y2***
9	0	5	5	1.000	1.715	5.249
10	6	4	-11	1.444	-14.974	-11.214
11	-2	3	31	0.200	17.695	31.435

Table 3.5.1 Results for Test Points using the *Growing* Option

We see that the results for Point 9 are exactly the same as for the *static* option (i.e., Table 3.2.3). However, Point 9 is then used in the learning set for the calculations for Point 10 and both Points 9 and 10 are used in the calculations for Point 11. The values of ***y0*** can most easily be verified. Since all points are weighted equally, the value of ***y0*** for Point 10 is just the average value of ***Y*** for Points 1 through 9 (i.e., (8 * 1.000 + 5) / 9 = 1.444). The value for Point 11 is the average value for Points 1 through 10 and is computed in a similar manner (i.e., (8*1.000 + 5 – 11) / 10 = 0.200). Using the *static* option with equally weighted learning points, a single plane was determined and was used to compute all three values of ***y1*** in Table 3.2.3. One single plane is no longer applicable when the *growing* option is used. The coefficients of the plane must be recomputed for each test point. Similarly, the coefficient used in the computations of the values of ***y2*** must also be recomputed for each test point.

The results for the moving option are included in Table 3.5.2:

Point	*x1*	*x2*	*Y*	*y0*	*y1*	*y2*
9	0	5	5	1.000	1.715	5.249
10	6	4	-11	1.000	-12.664	-11.379
11	-2	3	31	1.000	30.506	31.720

Table 3.5.2 Results for Test Points using the *Moving Window* Option

We first note that the results for Point 9 are exactly the same as observed in both Tables 3.2.3 and 3.5.1: a perfectly reasonable outcome. The results for ***y0*** for Points 10 and 11 at first glance seem strange: they are exactly the same as the results in Table 3.2.3 (i.e., using the *static* option). However, inspection of Table 3.2.2 provides the explanation. The computation for Point 10 requires discarding Point 1 and adding Point 9 to the learning data set. Since the ***Y*** values for both of these points are the same (i.e., 5), then the average value of ***Y*** remains the same. Since ***y0*** is just the average value of ***Y*** it remains 1. Similarly Point 11 is computed by discarding Points 1 and 2 and adding Points 9 and 10. Since the ***Y*** values of Points 2 and 11 are the same (i.e., -11) we once again get an average value of 1. However, the values of ***y1*** and ***y2*** for Points 10 and 11 are different from both the *static* and *growing* option results. The computation of ***y1*** and ***y2*** require not only the ***Y*** values of the learning data points but also the values of ***x1*** and ***x2***. Since a different set of values is used for each test point, the coefficients change from test point to test point and the results are different then the results obtained using the other options.

3.6 Searching for a Model

Typically when modeling financial data the analyst proposes a large number of potentially useful candidate predictors. Depending upon the problem and the available computer resources, the number of candidate predictors can range up into the hundreds. The strategy proposed for such analyses is to first try to build a model using each candidate predictor individually. Once all such 1D (i.e., one-dimensional) spaces have been considered, the analysis proceeds to 2D spaces, then 3D spaces, etc. If one considers all possible combinations, the number of spaces to be examined explodes exponentially as the number of candidate predictors increases.

To illustrate the magnitude of the problem, we need to be able to compute C_d^n which is the number of combinations of ***n*** things taken ***d*** at a time. For our purposes ***n*** is the number of candidate predictors and ***d*** is the dimensionality of the spaces. All books on probability theory include the following equation for C_d^n:

$$C_d^n = \frac{n!}{d!(n-d)!} \qquad \textbf{(3.6.1)}$$

For example, if ***n***=100 and ***d***=2, the number of possible 2D spaces is 100! / (2! * 98!) which reduces to 100*99/2 = 4950. If we plan to examine all possible spaces starting from 1D spaces up to a dimensionality of ***dmax***, then S_{total} (the total number of spaces) is simply:

$$S_{total} = \sum_{d=1}^{dmax} C_d^n \qquad \textbf{(3.6.2)}$$

n	*dmax=1*	*dmax=2*	*dmax=3*	*dmax=4*	*dmax=5*
5	5	15	25	30	31
10	10	55	175	385	637
20	20	210	1350	6195	21699
50	50	1275	20875	251175	2369935
100	100	5050	166750	4087975	79375495
150	150	11325	562625	20822900	612422930
200	200	20100	1333500	66018450	2601668490

Table 3.6.1 Values of S_{total} for combinations of *n* and *dmax*

Table 3.6.1 includes values of S_{total} for various combinations of ***n*** and ***dmax***. It is clear from this table that exhaustive searches for all combinations of candidate predictors become increasingly costly as ***n*** and ***dmax*** increase. What is required is a searching strategy that limits the number of combinations to be examined to a reasonable number. The definition of *reasonable* is of course problem dependent and machine dependent. One can quickly estimate the approximate time to analyze a single space and knowing how much time we wish to devote to the analysis, we can compute the number of spaces that can be examined in the desired time.

The simplest searching strategy is to use a *forward stepwise* approach. One would simply locate the best candidate predictor using a 1D analysis, and then pair this *best predictor* with all others to find the *best pair*. Using the *best pair* one would then proceed to examination of all triples that can be made from the *best pair* to obtain the *best triple*. This procedure can be carried on to higher and higher dimensions. The total amount of spaces examined using this procedure is very small. For example using this very simple *forward stepwise* approach with ***n***=200 and ***dmax***=5, we would only have to examine 200 + 199 + 198 + 197 + 196 = 990 spaces. From Table 3.6.1 we see that an exhaustive search of all combinations requires examining over 2.6 billion spaces! What is really required is a strategy that is a compromise between these two extremes.

A simple modification of the *forward stepwise* approach is to allow a user specified number of spaces to survive each level of dimensionality. These *survivors* would then be used with all other candidate predictors to create spaces at the next higher dimension. For example, lets assume a parameter called ***num_survivors*(*d*)** which is the number of survivors that will be used to create spaces at dimension ***d***+1. Assume ***n***=200 and ***num_survivors*(1)** is specified as 10. After the 200 1D spaces are examined and the 10 best performers are noted, the number of 2D spaces to be examined is 1945. This number includes all combinations of the 10 best (i.e., 10*9/2 = 45) plus all combinations of each of the 10 best with all the remaining 190 (i.e., 10*190 = 1900).

Calculation of the number of 3D spaces that must be examined is complicated by the fact that some predictors might appear in more than one of the best 2D spaces. For example, lets continue the previous example using ***n***=200 and set ***num_survivors*(2)** to 3. Lets assume that the best 3

pairs are (X7,X19), (X21,X47) and (X36,X52). The number of 3D spaces that would have to be examined would be 3*198 = 594. However, if the third space was (X47,X52) then the 3D space (X21,X47,X52) would be examined twice unless the search algorithm included some mechanism for preventing duplicate examinations. In other words, there are only 593 different 3D spaces for this case. We can, however, state an upper limit upon $S(d+1)$, the number of spaces that will be examined at dimensionality $d+1$ using this algorithm:

$$S(d+1) \leq num\_survivors(d) * (n-d) \qquad (3.6.3)$$

This equation gives the analyst a quick method for estimating the total number of spaces that will be examined. For example, consider the case of n=200 and $dmax$=5. Assume that *num_survivors*(*d*) is set to 10 for all values of d. There are 200 1D spaces and we have already computed $S(2)$=1945. We can use Equation 3.6.3 to estimate upper limits for $S(3)$, $S(4)$ and $S(5)$: $S(3)$<=10*198=1980, $S(4)$<=10*197=1970 and $S(5)$<=10*196=1960. The upper limit for S_{total} for this example is 8055. This number of spaces is about 8 times greater than the number of spaces examined using a simple *forward stepwise* search (i.e., 990) but is still many orders of magnitude less than the number of spaces that would be examined if all possible combinations were considered.

Any analysis must include a definition of the *modeling criterion MC.* A number of possible definitions of *MC* were discussed in Section 1.4. This list is by no means exhaustive. For example, when modeling financial markets one might prefer some sort of criterion based upon trading performance. However, regardless of the choice of *MC,* we end up with a single value for each space and spaces can be graded on the basis of this value. The best space is simply the space with the highest value of *MC.* A search algorithm should include some sort of criterion for aborting the search when it becomes pointless to proceed. The choice of a maximum value of $dmax$ is really quite arbitrary. What we would like to choose is a dimensionality which is high enough to capture the really good model (or models) if such models really exist but on the other hand not be so high that the data density is absurdly low (see Section 1.3).

One approach to this problem is to treat the parameter $num\_survivors(d)$ as an upper limit. An added criterion for evaluation of a space might be the required improvement from one dimension to the next higher dimension. If we define this required improvement in *MC* as δ, then a space failing to meet this criterion is immediately rejected regardless of the measured value of *MC*. For example, assume that one of the survivors of the 2D analyses is the space (X7,X19) and the measured value of *MC* for this space is 5.78. Assume that δ = 1 and the 3D spaces (X7,X11,X19) and (X7,X19,X46) are the best two 3D spaces created from (X7,X19). Furthermore, the values of *MC* for these two spaces are 7.31 and 6.49 respectively. The first of these two spaces would be included in the list of possible survivors but the second would be immediately rejected. This space (i.e., (X7,X11,X19)) would only become a survivor if the value 7.31 turns out to be within the top $num\_survivors(3)$ of 3D spaces examined. If there are no survivors for a particular level of dimensionality d, then the search is aborted even if $d<dmax$.

3.7 Timing Considerations

The use of kernel regression in data modeling for the types of problems associated with financial markets requires careful consideration of computational time. When one is faced with the task of developing prediction models in which there are thousands of data records and hundreds of candidate predictors, computational efficiency is of utmost importance. For computers exploiting a single processor, the total time for an analysis $\boldsymbol{T_{total}}$ can be estimated as follows:

$$\boldsymbol{T_{total} = S_{total} * T_{avg}} \qquad \textbf{(3.7.1)}$$

In this equation $\boldsymbol{S_{total}}$ is the total number of spaces examined and $\boldsymbol{T_{avg}}$ is the average time required per space. (In Section 4.9, parallel processing is considered and the implications regarding this equation are discussed.) In the previous section some aspects related to controlling $\boldsymbol{S_{total}}$ were considered. In this section the emphasis is on controlling $\boldsymbol{T_{avg}}$.

In Section 3.1 the basic concept of a kernel regression analysis was described. Values of the dependent variable ***Y*** were predicted for ***ntst*** test points using ***nlrn*** learning points. Using this very simple approach to kernel regression the time required for an evaluation of a single space would be $O(\boldsymbol{nlrn*ntst})$. In other words, the value of $\boldsymbol{T_{avg}}$ would tend to be proportional to the product ***nlrn*ntst***. If we assume that both ***nlrn*** and ***ntst*** are proportional to ***ntot*** (i.e., the total number of data records available for the analysis), we see that $\boldsymbol{T_{avg}}$ would thus be proportional to $ntot^2$. For typical cases in which the values of ***ntot*** is several thousand (or even several tens of thousands if intraday data is used), the values of T_{avg} become intolerably large.

In Section 3.3 we discussed the bandwidth concept in which only nearby learning points are used to predict values of ***Y*** for each test point. A simple way of accomplishing this is to compute the distance from every learning point to each test point, and use only those points within a specified distance. However, the computation of all these distances is still $O(\boldsymbol{nlrn*ntst})$. It is true that time will be saved because the remainder of the calculation will be faster. However, we are still left with a term that is increasing as $\boldsymbol{ntot^2}$ .and eventually this term will dominate the time required per space.

There is another major problem created by choosing a maximum distance between learning and test points. For most problems the data density is not even approximately constant throughout a particular space. In other words there are regions within the space where many learning points are concentrated and other regions which are sparsely populated. If we must select one distance for the entire space, then it will yield estimates of ***Y*** for some test points based upon many learning points and some estimates based upon few points. It is also possible that some test data points will fall within regions where none of the learning points are within the specified distance.

A simple solution to this problem is to specify a new parameter ***numnn***, which is the *number of nearest neighbors* that must be used for each test point. To accomplish this in the simplest and most straightforward manner, for each test point one would first compute all the distances to the learning points and then sort the distances. The closest ***numnn*** learning points would then be used to compute the estimated value of ***Y***. The time required to sort ***nlrn*** distances is $O(\boldsymbol{nlrn}*log(\boldsymbol{nlrn}))$. Since we would require ***ntst*** such sorts, we would be left with a term which

is $O(\mathbf{ntst}*\mathbf{nlrn}*log(\mathbf{nlrn}))$. This simple solution is thus plagued with a very high computational cost. What is required is some method of controlling the choice of learning points for each test point in such a manner that we achieve this in a rapid and efficient manner. Alternative approaches to this problem are considered in Chapter 4.