The Lazy Learning Toolbox

For use with Matlab©

Mauro Birattari and Gianluca Bontempi

IRIDIA
Université Libre de Bruxelles
Brussels, Belgium
{*mbiro, gbonte*}*@ulb.ac.be*

May 25, 1999

Abstract

Lazy learning is a memory-based technique that, once a query is received, extracts a prediction interpolating locally the neighboring examples of the query which are considered relevant according to a distance measure. This toolbox implements a data-driven method to select on a query-by-query basis the optimal number of neighbors to be considered for each prediction. As an efficient way to identify and validate local models, the recursive least squares algorithm is adopted. Furthermore, beside the *winner-takes-all* strategy for model selection, the toolbox implements also a *local combination*, performed on a query-by-query basis, of the most promising models.

This manual describes the functions included in the toolbox as well as the algorithms implemented and the underlying theory.

1 Introduction

The Lazy Learning Toolbox *For use with Matlab*© consists of four functions, written in C language, that implement the lazy learning methods for regression developed at IRIDIA, Université Libre de Bruxelles.

The software is part of a larger IRIDIA project, whose goal is the implementation of a set of local modeling approaches for data analysis and regression (Bontempi *et al.*, 1999).

The aim of this manual is to provide the user with a description of the functions composing the toolbox. Nevertheless, we try also to present in a formal way the algorithms we developed, and the underlying theory.

The Toolbox was developed for Matlab 5.2 on a PC running Red Hat Linux 5.0 for the joy of all its users. The authors enthusiastically recommend the principal software used: GNU Emacs, LaTeX 2_ε, gcc, and Matlab for Linux.

The *latest* version of the IRIDIA "Lazy Learning Toolbox for Use with Matlab" is available at `http://iridia.ulb.ac.be/~lazy/`. This manual is distributed together with the Toolbox but is also available as a technical report of the IRIDIA laboratory, Université Libre de Bruxelles (TR/IRIDIA/99-7).

In what follows, we refer to the current release 1.0 of the Toolbox which is, by the way, also the first "official" one.

This manual is organized as follows: Section 1 introduces lazy learning in general and the main features of the Toolbox. In Sections 2 to 5 we give a theoretical background on lazy learning and we derive the algorithms implemented in the Toolbox. Section 6 describes the functions composing the toolbox and Section 7 shows how these functions can be used to perform a prediction given a database of examples.

Legal issues and conditions

By using the toolbox the user agrees to all of the following:

- If any work where this toolbox has been used is going to publish, please remember that the software was obtained free of charge and please include a reference to:

  > Birattari M. , Bontempi G. & Bersini H. 1999. "Lazy learning meets the recursive least-squares algorithm", in *Advances in Neural Information Processing Systems 11*, M.S. Kearns, S.A. Solla, and D.A. Cohn, Eds., MIT Press, Cambridge, MA.

  and the url of the toolbox home page: `http://iridia.ulb.ac.be/~lazy/`. If the work is in the control field, please include also a reference to:

  > Bontempi G. , Birattari M. & Bersini H. 1999. Lazy learning for local modeling and control design. *International Journal of Control.* vol. 72, no. 7/8, pp. 643–658.

- The toolbox is copyrighted by Mauro Birattari, Gianluca Bontempi, and IRIDIA, Université Libre de Bruxelles. It is not permitted to use any part of this software in commercial and/or military applications.

- The toolbox is provided "as-is" without warranty of any kind, either express or implied, including but not limited to the implied warranties or conditions of merchantability or fitness for a particular purpose. In no event shall Mauro Birattari, Gianluca Bontempi, and/or the IRIDIA-ULB laboratory

2 Lazy Learning

Lazy learning (Aha, 1997) postpones all the computation until an explicit request for a prediction is received. The request is fulfilled interpolating locally the examples considered relevant according to a distance measure. Each prediction requires therefore a local modeling procedure that can be seen as composed of a *structural* and of a *parametric* identification. The parametric identification consists in the optimization of the parameters of the local approximator. On the other hand, structural identification involves, among other things, the selection of a family of local approximators, the selection of a metric to evaluate which examples are more relevant, and the selection of the *bandwidth* which indicates the size of the region in which the data are correctly modeled by members of the chosen family of approximators. For a comprehensive tutorial on local learning and for further references see Atkeson *et al.* (1997).

This toolbox implements a method in which the family of local approximators and the bandwidth are selected locally and tailored for each query point by means of a local leave-one-out cross-validation.

In what follows, the problem of bandwidth selection is reduced to the selection of the number k of neighboring examples which are given a non-zero weight in the local modeling procedure.

As far as the family of local approximator is concerned, the toolbox considers polynomials of different degrees and allows a local model selection, as well as a local combination of approximators of different degrees. In both cases, the models are compared on the basis of a local leave-one-out cross-validation.

Each time a prediction is required for a specific query point, a set of local models is identified, each with a different polynomial degree and each including a different number of neighbors. The generalization ability of each model is then assessed through a local cross-validation procedure. Finally, a prediction is obtained either combining or selecting the different models on the basis of some statistic of their cross-validation errors.

The major feature of this toolbox consists in the adoption of the *recursive least squares* algorithm for the identification of the local models. This is an appealing and efficient solution to the intrinsically incremental problem of identifying and validating a sequence of local linear models centered in the query point, each including a growing number of neighbors. It is worth noticing here that a leave-one-out cross-validation of each model considered does not involve any significant computational overload, since it is obtained though the PRESS statistic (Myers, 1990) which simply uses partial results returned by the recursive least squares algorithm.

3 Local Weighted Regression

Given two variables $\mathbf{x} \in \Re^m$ and $y \in \Re$, let us consider the mapping f: $\Re^m \to \Re$, known only through a set of n examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ obtained as follows:

$$y_i = f(\mathbf{x}_i) + \varepsilon_i, \tag{1}$$

where $\forall i$, ε_i is a random variable such that $E[\varepsilon_i] = 0$ and $E[\varepsilon_i \varepsilon_j] = 0$, $\forall j \neq i$, and such that $E[\varepsilon_i^r] = \mu_r(\mathbf{x}_i)$, $\forall r \geq 2$, where $\mu_r(\cdot)$ is the unknown r^{th} moment of the distribution of ε_i and is defined as a function of $\mathbf{x}_i$. In particular for $r = 2$, the last of the above mentioned properties implies that no assumption of global homoscedasticity is made.

The problem of local regression can be stated as the problem of estimating the value that the regression function $f(\mathbf{x}) = E[y|\mathbf{x}]$ assumes for a specific query point $\mathbf{x}$, using information pertaining only to a neighborhood of $\mathbf{x}$.

In what follows, only polynomials of degree one will be considered as local approximators. The extension to generic polynomials, and in particular to polynomials of degree zero and two, will be proposed at the end of the section.

Given a query point $\mathbf{x}_q$, and under the hypothesis of a local homoscedasticity of ε_i, the parameter $\boldsymbol{\beta}_1$ of a local first-degree polynomial approximating $f(\cdot)$ in a neighborhood of $\mathbf{x}_q$, can be obtained solving the local polynomial regression:

$$\sum_{i=1}^{n} \left\{ \left(y_i - \mathbf{x}'_{1,i} \boldsymbol{\beta}_1\right)^2 K\left(\frac{D(\mathbf{x}_i, \mathbf{x}_q)}{h}\right) \right\}, \tag{2}$$

where, given a metric on the space $\Re^m$, $D(\mathbf{x}_i, \mathbf{x}_q)$ is the distance from the query point to the i^{th} example, $K(\cdot)$ is a weight function, h is the bandwidth, and where the vectors $\mathbf{x}_{1,i}$ have been obtained by pre-appending a constant value 1 to each vector $\mathbf{x}_i$ in order to consider a constant term in the regression.

In matrix notation, the solution of the above stated weighted least squares problem is given by:

$$\hat{\boldsymbol{\beta}}_1 = (\mathbf{X}'_1 \mathbf{W}' \mathbf{W} \mathbf{X}_1)^{-1} \mathbf{X}'_1 \mathbf{W}' \mathbf{W} \mathbf{y} = (\mathbf{Z}'\mathbf{Z})^{-1} \mathbf{Z}' \mathbf{v} = \mathbf{P} \mathbf{Z}' \mathbf{v}, \tag{3}$$

where $\mathbf{X}_1$ is a matrix whose i^{th} row is $\mathbf{x}'_{1,i}$, $\mathbf{y}$ is a vector whose i^{th} element is y_i, $\mathbf{W}$ is a diagonal matrix whose i^{th} diagonal element is $w_{ii} = \sqrt{K\left(D(\mathbf{x}_i, \mathbf{x}_q)/h\right)}$, $\mathbf{Z} = \mathbf{W}\mathbf{X}_1$, $\mathbf{v} = \mathbf{W}\mathbf{y}$, and the matrix $\mathbf{X}'_1 \mathbf{W}' \mathbf{W} \mathbf{X}_1 = \mathbf{Z}'\mathbf{Z}$ is assumed to be non-singular so that its inverse $\mathbf{P} = (\mathbf{Z}'\mathbf{Z})^{-1}$ is defined.

Once obtained the local first-degree polynomial approximation, a prediction of $y_q = f(\mathbf{x}_q)$, is finally given by:

$$\hat{y}_{1,q} = \mathbf{x}'_{1,q}\hat{\boldsymbol{\beta}}_1. \tag{4}$$

Moreover, exploiting the linearity in the parameters of the local approximator, a leave-one-out cross-validation estimation of the error variance $E[(y_q - \hat{y}_{1,q})^2]$ can be obtained without any significant overload. In fact, using the PRESS statistic (Myers, 1990), it is possible to calculate the error $e^{cv}_{1,j} = y_j - \mathbf{x}'_{1,j}\hat{\boldsymbol{\beta}}_1^{-j}$, without explicitly identifying the parameters $\hat{\boldsymbol{\beta}}_1^{-j}$ from the examples available with the j^{th} removed. The formulation of the PRESS statistic for the case at hand is the following:

$$e^{cv}_{1,j} = y_j - \mathbf{x}'_{1,j}\hat{\boldsymbol{\beta}}_1^{-j} = \frac{y_j - \mathbf{x}'_{1,j}\mathbf{P}\mathbf{Z}'\mathbf{v}}{1 - \mathbf{z}'_j\mathbf{P}\mathbf{z}_j} = \frac{y_j - \mathbf{x}'_{1,j}\hat{\boldsymbol{\beta}}_1}{1 - h_{jj}}, \tag{5}$$

where $\mathbf{z}'_j$ is the j^{th} row of $\mathbf{Z}$ and therefore $\mathbf{z}_j = w_{jj}\mathbf{x}_{1,j}$, and where h_{jj} is the j^{th} diagonal element of the *Hat matrix* $\mathbf{H} = \mathbf{Z}\mathbf{P}\mathbf{Z}' = \mathbf{Z}(\mathbf{Z}'\mathbf{Z})^{-1}\mathbf{Z}'$.

In order to extend this method to polynomials of generic degree d, it is sufficient to substitute to the matrix $\mathbf{X}_1$ a matrix $\mathbf{X}_d$ whose i^{th} row contains the constant 1, the components of the i^{th} example $\mathbf{x}_i$ together with their powers up to the d^{th}, and the appropriate cross-terms. In particular, if $d = 0$ the matrix $\mathbf{X}_d$ reduces to a vector $\mathbf{X}_0$ whose elements are all equal to the constant 1. For $d = 2$, the i^{th} row of $\mathbf{X}_2$ is a vector of $(m+1)(m+2)/2$ elements which is composed by the homogeneous polynomial of degree zero, i.e. the constant 1, by the homogeneous polynomial of degree one, i.e. the vector $\mathbf{x}_i$ itself, and by the homogeneous polynomial of degree two in the m dimensional space $\Re^m$.

4 Recursive Local Regression

Also in this section we will present the case in which the local model is a polynomial of degree one. Again, the extension to generic polynomial approximators of any degree is straightforward.

We will assume also that a metric on the space $\Re^m$ is given. All the attention will be thus centered on the problem of bandwidth selection.

If as a weight function $K(\cdot)$ the indicator function

$$K\left(\frac{D(\mathbf{x}_i, \mathbf{x}_q)}{h}\right) = \begin{cases} 1 & \text{if } D(\mathbf{x}_i, \mathbf{x}_q) \le h, \\ 0 & \text{otherwise;} \end{cases} \tag{6}$$

is adopted, the optimization of the parameter h can be conveniently reduced to the optimization of the number k of neighbors to which a unitary weight is assigned in the local regression evaluation. In other words, we reduce the problem of bandwidth selection to a search in the space of $h(k) = D(\mathbf{x}(k), \mathbf{x}_q)$, where $\mathbf{x}(k)$ is the k^{th} nearest neighbor of the query point.

The main advantage deriving from the adoption of the weight function defined in Eq. 6, is that, simply by updating the parameter $\hat{\boldsymbol{\beta}}_1(k)$ of the model identified using the k nearest neighbors, it is straightforward and inexpensive to obtain $\hat{\boldsymbol{\beta}}_1(k+1)$. In fact, performing a step of the standard recursive least squares algorithm (Bierman, 1977), we have:

$$\begin{cases} \mathbf{P}(k+1) = \mathbf{P}(k) - \dfrac{\mathbf{P}(k)\mathbf{x}_1(k+1)\mathbf{x}_1'(k+1)\mathbf{P}(k)}{1+\mathbf{x}_1'(k+1)\mathbf{P}(k)\mathbf{x}_1(k+1)} \\ \boldsymbol{\gamma}(k+1) = \mathbf{P}(k+1)\mathbf{x}_1(k+1) \\ e(k+1) = y(k+1) - \mathbf{x}_1'(k+1)\hat{\boldsymbol{\beta}}_1(k) \\ \hat{\boldsymbol{\beta}}_1(k+1) = \hat{\boldsymbol{\beta}}_1(k) + \boldsymbol{\gamma}(k+1)e(k+1) \end{cases} \tag{7}$$

where $\mathbf{P}(k) = (\mathbf{Z}'\mathbf{Z})^{-1}$ when $h = h(k)$, and $\mathbf{x}_1(k)$ is the vector obtained pre-appending the constant 1 to the k^{th} nearest neighbor of the query point, and where $y(k)$ is the corresponding output.

Moreover, once the matrix $\mathbf{P}(k+1)$ is available, the leave-one-out cross-validation errors can be directly calculated without the need of any further model identification:

$$e^{cv}_{1,j}(k+1) = \frac{y_j - \mathbf{x}'_{1,j}\hat{\boldsymbol{\beta}}_1(k+1)}{1-\mathbf{x}'_{1,j}\mathbf{P}(k+1)\mathbf{x}_{1,j}}, \qquad \forall j : \ D(\mathbf{x}_j, \mathbf{x}_q) \le h(k+1). \tag{8}$$

It will be useful in the following to define for each value of k the $[k \times 1]$ vector $\mathbf{e}^{cv}_1(k)$ that contains all the leave-one-out errors associated to the model $\hat{\boldsymbol{\beta}}_1(k)$.

Once an initialization $\hat{\boldsymbol{\beta}}_1(0) = \tilde{\boldsymbol{\beta}}_1$ and $\mathbf{P}(0) = \tilde{\mathbf{P}}$ is given, Eq. 7 and Eq. 8 recursively evaluate for different values of k a local approximation of the regression function $f(\cdot)$, a prediction of the value of the regression function in the query point, and the vector of leave-one-out errors from which it is possible to extract an estimate of the variance of the prediction error. Notice that $\tilde{\boldsymbol{\beta}}_1$ is an *a priori* estimate of the parameter and $\tilde{\mathbf{P}}$ is the covariance matrix that reflects the reliability of $\tilde{\boldsymbol{\beta}}_1$ (Bierman, 1977). For non-reliable initialization, the following is usually adopted: $\tilde{\mathbf{P}} = \lambda\mathbf{I}$, with λ large and where $\mathbf{I}$ is the identity matrix.

The recursive algorithm described by Eq. 7 and Eq. 8 returns for a given query point $\mathbf{x}_q$, a set of predictions $\hat{y}_{1,q}(k) = \mathbf{x}'_{1,q}\hat{\boldsymbol{\beta}}_1(k)$, together with a set of associated leave-one-out error vectors $\mathbf{e}^{cv}_1(k)$. On the basis of these cross-validation errors, different statistics can be evaluated in order to compare the models obtained. Among them, the most natural and easy to implement is the *mean square error*:

$$mse^{cv}_1(k) = \frac{1}{k}\sum_{j=1}^{k} e^{cv}_{1,j}(k). \tag{9}$$

This same method described by Eq. 7 and Eq. 8 can be adopted also to identify a generic local model $\hat{\boldsymbol{\beta}}_d$, which is a polynomial of a generic degree d, if the matrix $\mathbf{X}_d$ is created as described at the end of Section 3. In this toolbox, the models of second degree are indeed identified and validated using this recursive method.

Local constant models

As far as constant models are concerned, an even more efficient method is available (Birattari & Bontempi, 1999) to compute $\hat{y}_{0,q}(k) = \hat{\boldsymbol{\beta}}_0(k)$. In this toolbox the following equations are adopted:

$$\hat{y}_{0,q}(k) = \frac{k-1}{k}\hat{y}_{0,q}(k-1) + \frac{1}{k}y(k), \tag{10}$$

and

$$mse_0^{cv}(k) = \frac{k(k-2)^2}{(k-1)^3} mse_0^{cv}(k-1) + \frac{1}{k-1}\big(y(k) - \hat{y}_{0,q}(k-1)\big)^2, \tag{11}$$

where $\hat{y}_{0,q}(0)$, and the recursion on $mse_0^{cv}(k)$ is started for $k = 2$: as a detail, it can be noticed that $mse_0^{cv}(1)$ does not need to be initialized since for $k = 2$ the first term in Eq. 11 equals zero. Equations 10 and 11 allow the exact computation of the leave-one-out mean square error, without the need of explicitly computing each single cross-validation error $e_{0,j}^{cv}(k)$.

5 Local Model Selection and Combination

From the information made available by Eq. 7 and 8 and/or by Eq. 10 and 11, a final prediction $\hat{y}_q$ of the value of the regression function can be obtained in different ways. Two main paradigms deserve to be considered: the first is based on the selection of the *best* approximator according to a given criterion, while the second returns a prediction as a combination of more local models.

We will consider now the case in which the degree d of the polynomial approximators has been selected *a priori* by the analyst. The extension to the automatic selection among or combination of local polynomials of different degree is proposed in a following subsection.

If the selection paradigm, frequently called *winner-takes-all*, is adopted, the most natural way to extract a final prediction $\hat{y}_{d,q}$, consists in comparing, on the basis of the classical *mean square error* criterion, the prediction obtained for each value of k, given the degree d of the local approximator.

$$\hat{y}_{d,q} = \mathbf{x}'_{d,q}\hat{\boldsymbol{\beta}}_d(\hat{k}), \quad \text{with } \hat{k} = \arg\min_{k \in \mathcal{K}(d)} mse_d^{cv}(k); \tag{12}$$

where $\mathcal{K}(d)$ is a range, defined by the analyst, from which the optimal number of neighbors is selected.

As an alternative to the *winner-takes-all* paradigm, we explored also the effectiveness of local combinations of estimates (Wolpert, 1992). Adopting also in this case the *mean square error* criterion, the final prediction of the value y_q is obtained as a weighted average of the best b models, where b is a parameter of the algorithm. Suppose the predictions $\hat{y}_{d,q}(k)$ and the error vectors $\mathbf{e}_d^{cv}(k)$ have been ordered creating a sequence of integers $\{k_i\}$ so that $mse_d^{cv}(k_i) \leq mse_d^{cv}(k_j)$, $\forall i < j$. The prediction of y_q is given by

$$\hat{y}_{d,q} = \frac{\sum_{i=1}^{b} \zeta_i \, \hat{y}_{d,q}(k_i)}{\sum_{i=1}^{b} \zeta_i}, \tag{13}$$

where the weights are the inverse of the mean square errors: $\zeta_i = 1/mse_d^{cv}(k_i)$. This is an example of the *generalized ensemble method* (Perrone & Cooper, 1993).

Selection of the polynomial degree of the local approximator

The same strategies described in Eq. 12 and 13 can be adopted also to select/combine models of different degrees, by comparing the predictions obtained for each ordered pair $<d,k>$ on the basis of the *mean square error* criterion:.

In this case the *winner-takes-all* strategy is implemented as follows:

$$\hat{y}_q = \mathbf{x}'_{\hat{d},q} \hat{\boldsymbol{\beta}}_{\hat{d}}(\hat{k}), \quad \text{with } <\hat{d},\hat{k}> = \arg\min_{<d,k>} mse_d^{cv}(k); \tag{14}$$

where $d \in \mathcal{D}$, $k \in \mathcal{K}(d)$ and where $\mathcal{D}$ and $\mathcal{K}(d)$ are appropriate ranges defined by the analyst.

As far as the local combination is concerned, let us consider a sequence $\{< d_i, k_i >\}$ ordered so that $mse_{d_i}^{cv}(k_i) \leq mse_{d_j}^{cv}(k_j)$, $\forall i < j$. The prediction of y_q is given in this case by:

$$\hat{y}_q = \frac{\sum_{i=1}^{b} \zeta_i \, \hat{y}_{d_i,q}(k_i)}{\sum_{i=1}^{b} \zeta_i}, \tag{15}$$

where b is the number of local models the analyst wants to combine, and where the weights are the inverse of the mean square errors: $\zeta_i = 1/mse_{d_i}^{cv}(k_i)$.

6 The Toolbox

In this section we present a detailed description of all the function composing the toolbox, and of their external behavior. In the following, we will assume that the reader is somehow acquainted with Matlab, that she is willing to test this toolbox (!), and that on her computer Matlab 5.2 or higher and an appropriate C compiler are correctly installed.

A notation will be introduced which does not match exactly the one used in the previous section but is closer to the Matlab notation. Any potential source of confusion will be handled by spelling out the new meaning of the symbols which are re-defined and by using a slightly different font.

6.1 Local Constant Models

For each query, the function `conLL.c` identifies and validates using a growing number of nearest-neighbors, a number of different local polynomial approximators of degree 0 (see Eq. 10, and 11). Among these models, the best one is selected by Eq. 12.

The function can be compiled from the Matlab command line as follows:

```
>> mex -O conLL.c
```

This creates in the current directory the file `conLL.ext` where the extension `ext` assumes different forms according to the platform. From now on, if the current directory is in the Matlab path, the function `conLL` can be called no matter what the current directory is.

The general way to call this function is:

```
>> [h,s,t,k,H,S,T,I] = conLL(X,Y,Q,id_par);
```

where the variable involved have the following meaning:

Input:

`X[n,m]`	Examples: Input matrix
`Y[n,1]`	Examples: Output vector
`Q[q,m]`	Queries: Input matrix
`id_par[2,1]`	Identification parameters: minimum and maximum number of neighbors to be considered: `id_par=[idm;idM]`

The i^{th} row of the matrix `X[n,m]` is the i^{th} input example $\mathbf{x}_i$, and the i^{th} element of the vector `Y[n,1]` is the corresponding output y_i. Similarly, each row of the matrix `Q[q,m]` describes a query point.

The 4^{th} input `id_par[2,1]` defines the range $\mathcal{K}(0) = \{idm, \dots, idM\}$ from which the best number of neighbors is selected (see Eq. 12).

The function accepts also a 5^{th} input: a vector `W[m,1]` of weights that can be used to modify the relative contribution of the d dimensions in the distance function.

The nearest-neighbors of each query point are obtained through an exhaustive search in the $\Re_1^m$ metric space (Manhattan distance):

$$D(\mathbf{x}_i, \mathbf{x}_q) = \frac{\sum_{j=1}^{m} W_{(j)} \left|\mathbf{x}_{i(j)} - \mathbf{x}_{q(j)}\right|}{\sum_{j=1}^{m} W_{(j)}}, \tag{16}$$

where $W_{(j)}$, $\mathbf{x}_{i(j)}$, and $\mathbf{x}_{q(j)}$ are the j^{th} components of the vectors `W[m,1]`, $\mathbf{x}_i$, and $\mathbf{x}_q$ respectively; and where $W_{(j)} = 1$, $\forall j$, if the vector `W[m,1]` is not given.

Output:

`h[q,1]`	Prediction with the selected number of neighbors for each query
`s[q,1]`	Leave-one-out error of the prediction obtained with the selected number of neighbors for each query
`t[0,0]`	DUMMY VARIABLE: for compatibility purposes
`k[q,1]`	Selected number of neighbors for each query
`H[idM,q]`	All the predictions obtained for each query using a number of neighbors in the range between 1 and `idM` (see `id_par`)
`S[idM,q]`	Leave-one-out error of all the predictions obtained for each query in `H[idM,q]`
`T[0,0,0]`	DUMMY VARIABLE: for compatibility purposes
`I[idM,q]`	Index of the `idM`-nearest-neighbors of each query point

The j^{th} element of the vector `h[q,1]` is the prediction relative to the j^{th} query obtained with the selected number of nearest-neighbors.

The j^{th} element of the vector `s[q,1]` is the leave-one-out mean square error of prediction relative to the j^{th} query, obtained with the selected number of nearest-neighbors.

The j^{th} element of the vector `k[q,1]` is the number of neighbors which has been selected in cross validation in order to answer to the j^{th} query.

The element in position (k, j) of the matrix `H[idM,q]` is the prediction of the output to the j^{th} query, yielded by the approximator identified with the first k nearest-neighbors of the j^{th} query itself.

The element in position (k, j) of the matrix `S[idM,q]` is the leave-one-out mean square error of the approximator identified with the first k nearest-neighbors of the j^{th} query point.

The element in position (k, j) of the matrix `I[idM,q]` is the index of the k^{th} nearest-neighbor of the j^{th} query point, i.e. the original position in the matrix `X[n,m]` of the k^{th} neighbor of the j^{th} query point.

6.2 Local Linear Models

For each query, the function `linLL.c` identifies and validates using a growing number of nearest-neighbors, a number of different local polynomial approximators of degree 1 (see Eq. 7, eq:RecursivePress, and 9). Among these models, the best one is selected by Eq. 12.

The function can be compiled from the Matlab command line as follows:

```
>> mex -O linLL.c
```

This creates in the current directory the file `linLL.ext` where the extension `ext` assumes different forms according to the platform. From now on, if the current directory is in the Matlab path, the function `linLL` can be called no matter what the current directory is.

The general way to call this function is:

```
>> [h,s,t,k,H,S,T,I] = linLL(X,Y,Q,id_par);
```

where the variable involved have the following meaning:

Input:

`X[n,m]`	Examples: Input matrix
`Y[n,1]`	Examples: Output vector
`Q[q,m]`	Queries: Input matrix
`id_par[2,1]`	Identification parameters: minimum and maximum number of neighbors to be considered: `id_par=[idm;idM]`

The i^{th} row of the matrix `X[n,m]` is the i^{th} input example $\mathbf{x}_i$, and the i^{th} element of the vector `Y[n,1]` is the corresponding output y_i. Similarly, each row of the matrix `Q[q,m]` describes a query point.

The 4^{th} input `id_par[2,1]` defines the range $\mathcal{K}(1) = \{idm, \dots, idM\}$ from which the best number of neighbors is selected (see Eq. 12).

The function accepts also a 5^{th} input: a scalar `LAMBDA[1,1]` which is a regularization parameter. The default value is `LAMBDA = 1E6`. This parameter is used to define the diagonal matrix $\tilde{\mathbf{P}} = \lambda \mathbf{I}$, used to initialize the recursive algorithm described in Eq. 7.

The function accepts also a 6^{th} input: a vector `W[m,1]` of weights that can be used to modify the relative contribution of the d dimensions in the distance function.

The nearest-neighbors of each query point are obtained through an exhaustive search in the $\Re_1^m$ metric space (Manhattan distance):

$$D(\mathbf{x}_i, \mathbf{x}_q) = \frac{\sum_{j=1}^{m} W_{(j)} \left|\mathbf{x}_{i(j)} - \mathbf{x}_{q(j)}\right|}{\sum_{j=1}^{m} W_{(j)}}, \tag{17}$$

where $W_{(j)}$, $\mathbf{x}_{i(j)}$, and $\mathbf{x}_{q(j)}$ are the j^{th} components of the vectors `W[m,1]`, $\mathbf{x}_i$, and $\mathbf{x}_q$ respectively; and where $W_{(j)} = 1$, $\forall j$, if the vector `W[m,1]` is not given.

Output:

`h[q,1]`	Prediction with the selected number of neighbors for each query
`s[q,1]`	Leave-one-out error of the prediction obtained with the selected number of neighbors for each query
`t[m+1,q]`	Selected model for each query
`k[q,1]`	Selected number of neighbors for each query
`H[idM,q]`	All the predictions obtained for each query using a number of neighbors in the range between 1 and `idM` (see `id_par`)
`S[idM,q]`	Leave-one-out error of all the predictions obtained for each query in `H[idM,q]`
`T[m+1,idM,q]`	All the models considered for each query
`I[idM,q]`	Index of the `idM`-nearest-neighbors of each query point

The j^{th} element of the vector `h[q,1]` is the prediction relative to the j^{th} query, obtained with the selected number of nearest-neighbors.

The j^{th} column of the matrix `t[m+1,q]` is a vector that contains the parameters of the model, obtained with the selected number of nearest-neighbors, used to answer to the j^{th} query. Each column is then:

$$|a_0, a_1, a_2, \ldots, a_m|' \tag{18}$$

where a_0 is the constant term of the model, and the generic a_i is the parameter associated with the i^{th} input variable $\mathbf{x}_{(i)}$. **Remark:** A translation of the axes is considered which centers all the local models in the respective query point.

The j^{th} element of the vector `s[q,1]` is the leave-one-out mean square error of prediction relative to the j^{th} query, obtained with the selected number of nearest-neighbors.

The j^{th} element of the vector `k[q,1]` is the number of neighbors which has been selected in cross validation in order to answer to the j^{th} query.

The element in position (k, j) of the matrix `H[idM,q]` is the prediction of the output to the j^{th} query, yielded by the approximator identified with the first k nearest-neighbors of the j^{th} query itself.

The element in position (k, j) of the matrix `S[idM,q]` is the leave-one-out mean square error of the approximator identified with the first k nearest-neighbors of the j^{th} query point.

The element in position (i, k, j) of the matrix `T[m+1,idM,q]` is the i^{th} parameter (see Eq. 18) of the local model identified with the first k nearest-neighbors of the j^{th} query point.

The element in position (k, j) of the matrix `I[idM,q]` is the index of the k^{th} nearest-neighbor of the j^{th} query point, i.e. the original position in the matrix `X[n,m]` of the k^{th} neighbor of the j^{th} query point.

6.3 Local Quadratic Models

For each query, the function `quaLL.c` identifies and validates using a growing number of nearest-neighbors, a number of different local polynomial approximators of degree 2 (see Eq. 7, eq:RecursivePress, and 9). Among these models, the best one is selected by Eq. 12.

The function can be compiled from the Matlab command line as follows:

```
>> mex -O quaLL.c
```

This creates in the current directory the file `quaLL.ext` where the extension `ext` assumes different forms according to the platform. From now on, if the current directory is in the Matlab path, the function `quaLL` can be called no matter what the current directory is.

The general way to call this function is:

```
>> [h,s,t,k,H,S,T,I] = quaLL(X,Y,Q,id_par);
```

where the variable involved have the following meaning:

Input:

`X[n,m]`	Examples: Input matrix
`Y[n,1]`	Examples: Output vector
`Q[q,m]`	Queries: Input matrix
`id_par[2,1]`	Identification parameters: minimum and maximum number of neighbors to be considered: `id_par=[idm;idM]`

The i^{th} row of the matrix `X[n,m]` is the i^{th} input example $\mathbf{x}_i$, and the i^{th} element of the vector `Y[n,1]` is the corresponding output y_i. Similarly, each row of the matrix `Q[q,m]` describes a query point.

The 4^{th} input `id_par[2,1]` defines the range $\mathcal{K}(2) = \{idm, \ldots, idM\}$ from which the best number of neighbors is selected (see Eq. 12).

The function accepts also a 5^{th} input: a scalar `LAMBDA[1,1]` which is a regularization parameter. The default value is `LAMBDA = 1E6`. This parameter is used to define the diagonal matrix $\tilde{\mathbf{P}} = \lambda \mathbf{I}$, used to initialize the recursive algorithm described in Eq. 7.

The function accepts also a 6^{th} input: a vector `W[m,1]` of weights that can be used to modify the relative contribution of the d dimensions in the distance function.

The nearest-neighbors of each query point are obtained through an exhaustive search in the $\Re_1^m$ metric space (Manhattan distance):

$$D(\mathbf{x}_i, \mathbf{x}_q) = \frac{\sum_{j=1}^{m} W_{(j)} \left|\mathbf{x}_{i(j)} - \mathbf{x}_{q(j)}\right|}{\sum_{j=1}^{m} W_{(j)}}, \tag{19}$$

where $W_{(j)}$, $\mathbf{x}_{i(j)}$, and $\mathbf{x}_{q(j)}$ are the j^{th} components of the vectors `W[m,1]`, $\mathbf{x}_i$, and $\mathbf{x}_q$ respectively; and where $W_{(j)} = 1$, $\forall j$, if the vector `W[m,1]` is not given.

Output:

`h[q,1]`	Prediction with the selected number of neighbors for each query
`s[q,1]`	Leave-one-out error of the prediction obtained with the selected number of neighbors for each query
`t[p,q]`	Selected model for each query: `p=(m+1)*(m+2)/2`
`k[q,1]`	Selected number of neighbors for each query
`H[idM,q]`	All the predictions obtained for each query using a number of neighbors in the range between 1 and `idM` (see `id_par`)
`S[idM,q]`	Leave-one-out error of all the predictions obtained for each query in `H[idM,q]`
`T[p,idM,q]`	All the models considered for each query
`I[idM,q]`	Index of the `idM`-nearest-neighbors of each query point

The j^{th} element of the vector `h[q,1]` is the prediction relative to the j^{th} query, obtained with the selected number of nearest-neighbors.

The j^{th} column of the matrix `t[p,q]` is a vector that contains the parameters of the model, obtained with the selected number of nearest-neighbors, used to answer to the j^{th} query. Each column is then:

$$|a_0, a_1, a_2, \ldots, a_{11}, a_{12}, a_{13}, \ldots, a_{22}, a_{23}, a_{24}, \ldots, a_{33}, a_{34}, a_{35}, \ldots|' \qquad (20)$$

where a_0 is the constant term of the model, a_i is the parameter associated with the i^{th} input variable $\mathbf{x}_{(i)}$, a_{ii} is the parameter of the quadratic term $\mathbf{x}_{(i)}^2$, and a_{il} is the parameter of the cross term $\mathbf{x}_{(i)}\mathbf{x}_{(l)}$. **Remark:** A translation of the axes is considered which centers all the local models in the respective query point.

The j^{th} element of the vector `s[q,1]` is the leave-one-out mean square error of prediction relative to the j^{th} query, obtained with the selected number of nearest-neighbors.

The j^{th} element of the vector `k[q,1]` is the number of neighbors which has been selected in cross validation in order to answer to the j^{th} query.

The element in position (k, j) of the matrix `H[idM,q]` is the prediction of the output to the j^{th} query, yielded by the approximator identified with the first k nearest-neighbors of the j^{th} query itself.

The element in position (k, j) of the matrix `S[idM,q]` is the leave-one-out mean square error of the approximator identified with the first k nearest-neighbors of the j^{th} query point.

The element in position (i, k, j) of the matrix `T[m+1,idM,q]` is the i^{th} parameter (see Eq. 20) of the local model identified with the first k nearest-neighbors of the j^{th} query point.

The element in position (k, j) of the matrix `I[idM,q]` is the index of the k^{th} nearest-neighbor of the j^{th} query point, i.e. the original position in the matrix `X[n,m]` of the k^{th} neighbor of the j^{th} query point.

6.4 Local Combination of Models

For each query, the function `clqLL.c` identifies and validates using a growing number of nearest-neighbors, a number of different local polynomial approximators of degree 0, 1, and 2 (see Eq. 10, 11, 7, 8, and 9). According to the value of some input variables, the function selects the best one as in Eq. 14, or combines a number of best models as in Eq. 15

The function can be compiled from the Matlab command line as follows:

```
>> mex -O clqLL.c
```

This creates in the current directory the file `clqLL.ext` where the extension `ext` assumes different forms according to the platform. From now on, if the current directory is in the Matlab path, the function `clqLL` can be called no matter what the current directory is.

The general way to call this function is:

```
>> [h,t] = clqLL(X,Y,Q,id_par);
```

where the variable involved have the following meaning:

Input:

`X[n,m]`	Examples: Input matrix
`Y[n,1]`	Examples: Output vector
`Q[q,m]`	Queries: Input matrix
`id_par[_,_]`	Identification parameters: details on dimensions follow

Optional Input:

`cmb_par[_,_]`	Combination parameters. Default: `cmb_par = 1`. Details on dimensions follow
`LAMBDA[1,1]`	Initialization of the diagonal elements of the local variance/covariance matrix. Default: `LAMBDA = 1E6`
`W[1,m]`	Weights used to evaluate the distances. Default: `W = ones(1,m)`

The i^{th} row of the matrix `X[n,m]` is the i^{th} input example $\mathbf{x}_i$, and the i^{th} element of the vector `Y[n,1]` is the corresponding output y_i. Similarly, each row of the matrix `Q[q,m]` describes a query point.

The 4^{th} input `id_par[_,_]` defines the range in which the number of neighbors is searched. The identification parameter can assume the following forms:

1. `id_par[3,3]` $$\texttt{id\_par} = \begin{vmatrix} \texttt{idm0} & \texttt{idM0} & \texttt{valM0} \\ \texttt{idm1} & \texttt{idM1} & \texttt{valM1} \\ \texttt{idm2} & \texttt{idM2} & \texttt{valM2} \end{vmatrix}$$

   where [idm$d$,idMd] is the range $\mathcal{K}(d)$ in which the best number of neighbors is searched when identifying the local model of degree d. The scalar valMd defines the maximum number of neighbors on which the local model of degree d is validated:

   $$mse_d^{cv}(k) = \frac{1}{m_d(k)} \sum_{j=1}^{m_d(k)} e_{d,j}^{cv}(k).$$

   where $m(k) = \min(k, \texttt{valM}d)$.

2. `id_par[3,2]` $$\texttt{id\_par} = \begin{vmatrix} \texttt{idm0} & \texttt{idM0} \\ \texttt{idm1} & \texttt{idM1} \\ \texttt{idm2} & \texttt{idM2} \end{vmatrix}$$

   where [idm$d$,idMd] have the same meaning as in point 1, and valMd assumes the default value of idMd for all the values of d: for every local model considered, all the neighbors used in identification are used also in validation.

3. `id_par[3,1]` $$\texttt{id\_par} = \begin{vmatrix} \texttt{c0} \\ \texttt{c1} \\ \texttt{c2} \end{vmatrix}$$

   Here idm$d$ and idMd are obtained according to the following formulas:

   $$\texttt{idm}d = \texttt{floor(3 * p}d\texttt{ * c}d\texttt{)}$$
   $$\texttt{idM}d = \texttt{ceil(5 * p}d\texttt{ * c}d\texttt{)}$$

   where pd is the number of parameter of the model of degree d. Recommended choice: c$d = 1$. The validation parameters get the default value as in point 2.

The function accepts also a 5^{th} input, `cmb_par[_,_]` that determines the behavior of the function for what concerns the combination/selection of the local models. If `cmb_par` is not given, the best model is selected among those identified as specified by `id_par`. In this case, the model combination reduces to a simple model selection. The default value for `cmb_par` is 1 as it will be clear from what follows. If given, `cmb_par` can assume the following to forms:

1. `cmb_par[3,1]` $\quad$ `cmb_par` $= \begin{vmatrix} \texttt{cmb0} \\ \texttt{cmb1} \\ \texttt{cmb2} \end{vmatrix}$

   where `cmb`d is the number of models of degree d that will be included in the local combination. Each local model will be therefore a combination of the best `cmb0` models of degree 0, the best `cmb1` models of degree 1, and the best `cmb2` models of degree 1, all identified as specified by `id_par`.

2. `cmb_par[1,1]` $\quad$ `cmb_par` $= \begin{vmatrix} \texttt{cmb} \end{vmatrix}$

   where `cmb` is the number of models that will be combined, disregarding any constraint on the degree of the models that will be considered. Each local model will be therefore a combination of the best `cmb` models, identified as specified by `id_par`.

The function accepts also a 6^{th} input: a scalar `LAMBDA[1,1]` which is a regularization parameter. The default value is `LAMBDA = 1E6`. This parameter is used to define the diagonal matrix $\tilde{\mathbf{P}} = \lambda \mathbf{I}$, used to initialize the recursive algorithm described in Eq. 7.

The function accepts also a 7^{th} input: a vector `W[m,1]` of weights that can be used to modify the relative contribution of the d dimensions in the distance function.

The nearest-neighbors of each query point are obtained through an exhaustive search in the $\Re_1^m$ metric space (Manhattan distance):

$$D(\mathbf{x}_i, \mathbf{x}_q) = \frac{\sum_{j=1}^{m} W_{(j)} \left| \mathbf{x}_{i(j)} - \mathbf{x}_{q(j)} \right|}{\sum_{j=1}^{m} W_{(j)}}, \tag{21}$$

where $W_{(j)}$, $\mathbf{x}_{i(j)}$, and $\mathbf{x}_{q(j)}$ are the j^{th} components of the vectors `W[m,1]`, $\mathbf{x}_i$, and $\mathbf{x}_q$ respectively; and where $W_{(j)} = 1$, $\forall j$, if the vector `W[m,1]` is not given.

Output:

`h[q,1]`	Prediction with the selected number of neighbors for each query
`t[_,q]`	Selected model for each query: details on dimensions follow

The j^{th} element of the vector `h[q,1]` is the prediction relative to the j^{th} query.

The j^{th} column of the matrix `t[p,q]` is a vector that contains the parameters of the model used to answer to the j^{th} query. If according to `id_par` and `cmb_par` only constant models are considered `t[p,q]` reduces to a vector `t[1,q]` in which each "column" is the single parameter of a constant model. If at least one model of degree 1 and no model of degree 2 are considered, each column of `t[p,q]` is a vector that contains `p=m+1` parameters: $|a_0, a_1, a_2, \ldots|$ where a_0 is the constant term of the model and a_i is the parameter associated with the i^{th} input variable $\mathbf{x}_{(i)}$. If at least one model of degree 2 is considered, each column of `t[p,q]`, where `p=(m+1)*(m+2)/2`, is a vector that contains the parameters of a local model with the following convention:

$$|a_0, a_1, a_2, \ldots, a_{11}, a_{12}, a_{13}, \ldots, a_{22}, a_{23}, a_{24}, \ldots, a_{33}, a_{34}, a_{35}, \ldots|'$$

where a_0 is the constant term of the model, a_i is the parameter associated with the i^{th} input variable $\mathbf{x}_{(i)}$, a_{ii} is the parameter of the quadratic term $\mathbf{x}^2_{(i)}$, and a_{il} is the parameter of the cross term $\mathbf{x}_{(i)}\mathbf{x}_{(l)}$. **Remark:** A translation of the axes is considered which centers all the local models in the respective query point.

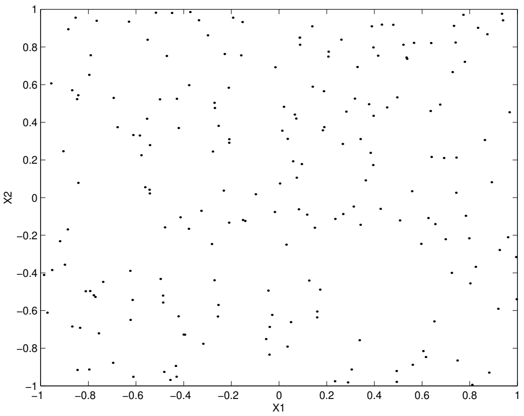

Figure 1: Distribution of the examples in the input space.

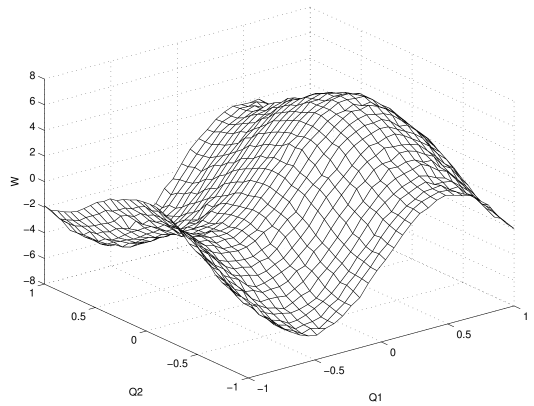

Figure 2: The target is a set of $32^2 = 1024$ query points distributed on a grid.

7 An Example

In this section we will propose an example of use of the Lazy Learning Toolbox. We suppose that the four functions have been compiled and that they are in the Matlab path.

Let us consider the problem of learning the input-output relation:

$$y = 4\sin\left(\pi\,\mathbf{x}_{(1)}\right) + 2\cos\left(\pi\,\mathbf{x}_{(2)}\right) + N(0, 0.1),$$

where the variables $\mathbf{x}_{(1)}$ and $\mathbf{x}_{(2)}$ range over the domain $[-1, 1]$ and $N(0, 0.1)$ is a Gaussian random noise with mean equal to 0, and standard deviation equal to 0.1.

Data generation

Let us assume that only a training set of $n = 200$ examples is available:

```
>> n=200;
>> m=2;
>> X=1-2*rand(n,m);
>> plot(X(:,1),X(:,2),'.');
>> Y=4*sin(pi*X(:,1))+2*cos(pi*X(:,2))+.1*randn(n,1);
```

The distribution of the examples if shown in Fig. 1.

Let us assume also that we want to predict the output in a set of $32^2 = 1024$ query points equally spaced and distributed on a grid:

```
>> qq=linspace(-1,1,32);
>> [Q1,Q2]=meshgrid(qq);
>> Q=[Q1(:),Q2(:)];
>> w=4*sin(pi*Q(:,1))+2*cos(pi*Q(:,2))+.1*randn(1024,1);
>> W=reshape(w,32,32);
>> mesh(Q1,Q2,W);
```

Here `w[1024,1]` is the "real" value of the output associated with the query points (see Fig. 2). Of course it will not be use to perform the prediction and will be used only in a very final phase to quantitative compare the results.

In what follows we show how to extract a prediction, and as far as the functions `conLL`, `linLL` , and `quaLL` are concerned, how to extract the number of nearest-neighbors used to perform each prediction. Since functions allocate memory and perform additional computations according to the number of output parameter specified at the Matlab prompt, we suggest not to ask for the full set of outputs if they are not needed. Of course, since Matlab recognizes the output matrices based on their position, if the i^{th} is needed also the j^{th}, with $j \leq i$ must be specified.

The following examples illustrate the main features of the Lazy Toolbox but are not exhaustive. Please refer to the previous section for a complete description of the functions. The results proposed are not to be considered as the best that can be obtained with the Toolbox on this problem, they are simply examples . . .

7.1 Local Constant Models

Using local constant models, the prediction of the values assumed on the grid can be obtained as follows:

```
>> id_par=[4;15];
>> tic;h=conLL(X,Y,Q,id_par);toc
elapsed_time =
    0.0497
>> rmse(h,w)
ans =
    0.7022
>> H=reshape(h,32,32);
>> mesh(Q1,Q2,H);
```

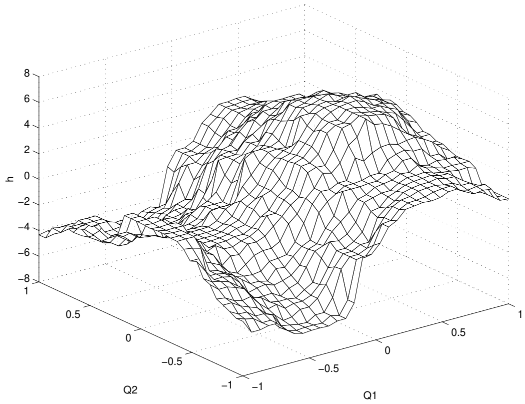

Figure 3: Prediction obtained with local constant models. $RMSE = 0.7022$.

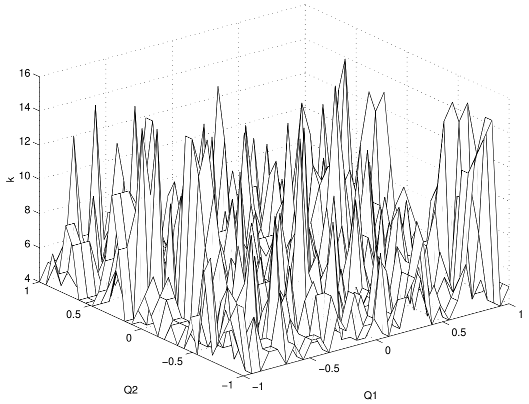

Figure 4: Number of nearest-neighbors used to perform each prediction.

For each of the 1024 query points, with this experiment the function automatically selects in the range $[4 \dots 15]$ the best number of neighbors to be used to fit a local constant model (Fig. 3). The entire experiment, i.e. the prediction for all the query points, took $0.0497s$ on a Pentium 400MHz. The achieved *root mean square error* is 0.7022.

7.2 Local Linear Models

Using local linear models, the prediction of the values assumed on the grid can be obtained as follows:

```
>> id_par=[4;15];
>> tic;h=linLL(X,Y,Q,id_par);toc
elapsed_time =
    0.1444
>> rmse(h,w)
ans =
    0.3255
>> H=reshape(h,32,32);
>> mesh(Q1,Q2,H);
```

For each of the 1024 query points, with this experiment the function automatically selects in the range $[4 \dots 15]$ the best number of neighbors to be used to fit a local linear model (Fig. 5). The entire experiment, i.e. the prediction for all the query

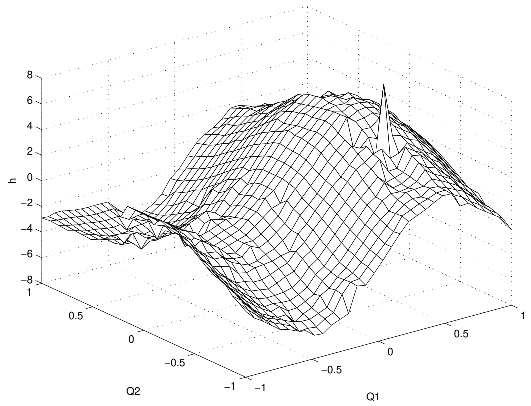

Figure 5: Prediction obtained with local linear models. $RMSE = 0.3255$.

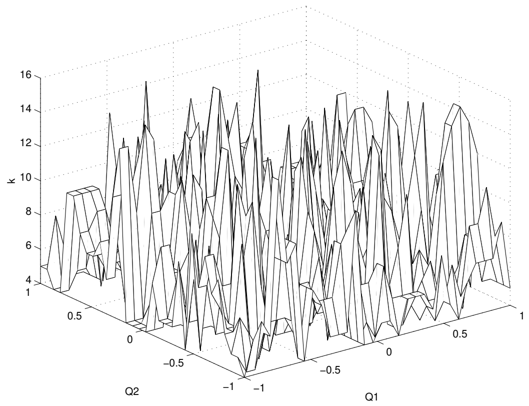

Figure 6: Number of nearest-neighbors used to perform each prediction.

points, took $0.1444s$ on a Pentium 400MHz. The achieved *root mean square error* is 0.3255.

7.3 Local Quadratic Models

Using local linear models, the prediction of the values assumed on the grid can be obtained as follows:

```
>> id_par=[4;15];
>> tic;h=quaLL(X,Y,Q,id_par);toc
elapsed_time =
     0.3298
>> rmse(h,w)
ans =
     0.2176
>> H=reshape(h,32,32);
>> mesh(Q1,Q2,H);
```

For each of the 1024 query points, with this experiment the function automatically selects in the range $[4 \ldots 15]$ the best number of neighbors to be used to fit a local quadratic model (Fig. 7). The entire experiment, i.e. the prediction for all the query points, took $0.3298s$ on a Pentium 400MHz. The achieved *root mean square error* is 0.2176.

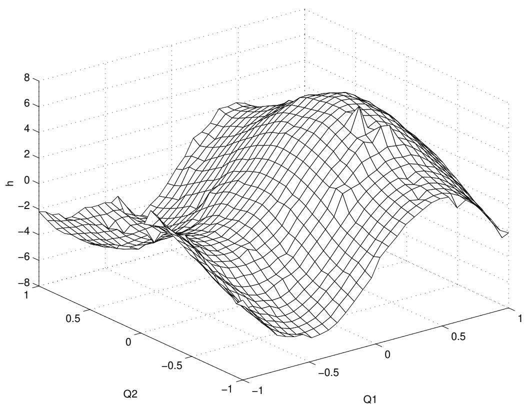

Figure 7: Prediction obtained with local quadratic models. $RMSE = 0.2176$.

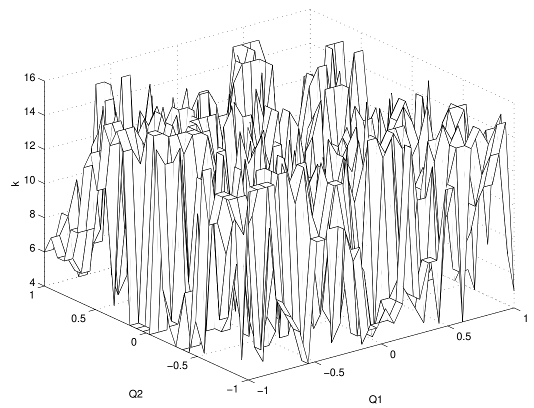

Figure 8: Number of nearest-neighbors used to perform each prediction.

7.4 Local Combination of Models

The function `clqLL` is more complex and more powerful than the previous three functions. Indeed, for particular values of the parameters, it is possible to obtain exactly the same predictions yielded by `conLL`, `linLL`, and `quaLL`. This three function, on the other hand, return more information, e.g. all the local models considered, the mean square error, etc. and thus are not completely superseded by `clqLL`. The same results obtained in the previous experiments can be obtained as follows:

```
>> id_par=[[4,15];[4,15];[4,15]];
>> cmb_par=[1;0;0];
>> h0=clqLL(X,Y,Q,id_par,cmb_par);
>> cmb_par=[0;1;0];
>> h1=clqLL(X,Y,Q,id_par,cmb_par);
>> cmb_par=[0;0;1];
>> h2=clqLL(X,Y,Q,id_par,cmb_par);
```

Where `h0` is the vector of prediction obtained by selecting locally the best constant approximator, `h1` by selecting the best linear approximator, and `h2` by selecting the best quadratic one.

The selection of the best polynomial degree, together with the selection of the best number of neighbors for each query point can be obtained as follows:

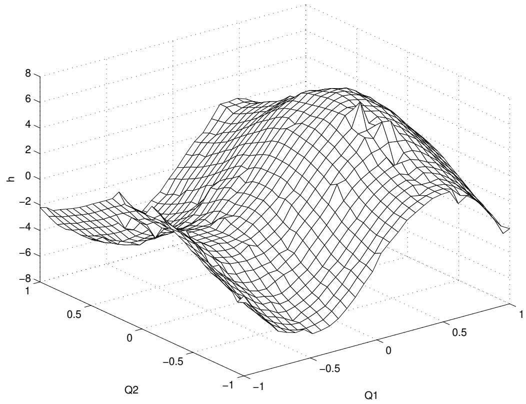

Figure 9: Prediction obtained with local selection of the degree of the polynomial and of the number of neighbors. $RMSE = 0.2148$.

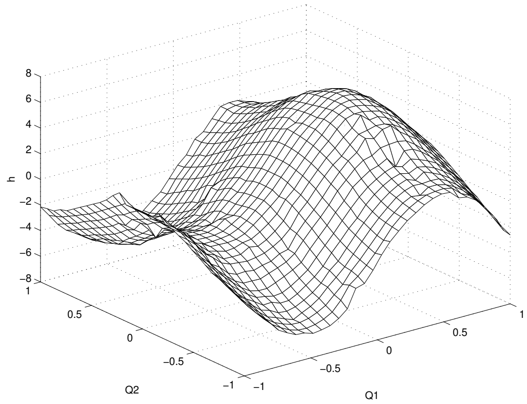

Figure 10: Prediction obtained with local combination of 5 models of different polynomial degree and identified with a different number of neighbors. $RMSE = 0.1883$.

```
>> id_par=[[4,15];[4,15];[4,15]];
>> tic;h=clqLL(X,Y,Q,id_par);toc
elapsed_time =
    0.4839
>> rmse(h,w)
ans =
    0.2148
>> H=reshape(h,32,32);
>> mesh(Q1,Q2,H);
```

For each of the 1024 query points, with this experiment the function automatically selects in the range $[4 \ldots 15]$ the best number of neighbors to be used to fit the local models of different polynomial degree, and then automatically selects the degree which is expected to perform better (Fig. 9). The entire experiment, i.e. the prediction for all the query points, took $0.4839s$ on a Pentium 400MHz. The achieved *root mean square error* is 0.2148.

An even better result can be obtained by *combining* locally a number of local model. In the following example we combine for each query 5 local models leaving unchanged the ranges in which the best number of nearest-neighbors are searched for.

```
>> id_par=[[4,15];[4,15];[4,15]];
>> cmb_par=5;;
>> tic;h=clqLL(X,Y,Q,id_par,cmb_par);toc
elapsed_time =
    0.4874
>> rmse(h,w)
ans =
    0.1883
>> H=reshape(h,32,32);
>> mesh(Q1,Q2,H);
```

For each of the 1024 query points, with this experiment the function automatically selects and combines the best 5 models (according to a local leave-one-out validation) of degree from zero to two, and identified with a number of neighbors in the range $[4 \dots 15]$ (Fig. 10). The entire experiment, i.e. the prediction for all the query points, took $0.4874s$ on a Pentium 400MHz. The achieved *root mean square error* is 0.1883.

In order to obtain a higher flexibility, the analyst can specify for each polynomial degree a different range in which to search for the best number of neighbors:

```
>> id_par=[[3,10];[9,15];[12,25]];
```

Furthermore, a default option is available that allows the analyst to consider a range of nearest-neighbors whose bounds are function of the number of parameters of the local approximator. For example, with the following:

```
>> id_par=[1;1;.8];
```

the best number of neighbors is searched in a range between 3 and 5 times the number of parameter for models of degree zero and one, and between 0.8×3 and 0.8×5 for the model of degree two. By putting one of the three components of the vector `id_par` to zero, one obtains that the models of the corresponding degree are not considered. For example, with:

```
>> id_par=[1;1;0];
```

for the problem at hand whose input is two dimensional, the function identifies only local model of degree zero and one and looks for the best number of neighbors in the range $[3 \dots 5]$ for the former, and in the range $[9 \dots 15]$ for the latter. The same configuration can be explicitly specified with:

```
>> id_par=[[3,5];[9,15];[0,0]];
```

As far as the combination parameter is concerned, two options are available. With the following:

```
>> cmb_par=10;
```

the analyst obtains that for each query point the best 10 local models, among those considered according to `id_par`, are combined without any constraint on their degree. The second option consists in explicitly specifying the number of models required for each degree. For example:

```
>> cmb_par=[2;5;3];
```

requires that for each query the best 2 constant models are combined with the best 5 linear models and with the best 3 quadratic models.

References

Aha D.W. 1997. Editorial. *Artificial Intelligence Review*, **11**(1–5), 1–6. Special Issue on Lazy Learning.

Atkeson C.G. , Moore A.W. & Schaal S. 1997. Locally weighted learning. *Artificial Intelligence Review*, **11**(1–5), 11–73.

Bierman G.J. 1977. *Factorization Methods for Discrete Sequential Estimation.* New York, NY: Academic Press.

Birattari M. & Bontempi G. 1999. *Lazy learning Vs. Speedy Gonzales: A fast algorithm for recursive identification and recursive validation of local constant models.* Technical Report: Iridia, Université Libre de Bruxelles. TR/IRIDIA/99-6. *Submitted for publication.*

Birattari M. , Bontempi G. & Bersini H. 1999. "Lazy learning meets the recursive least-squares algorithm", in *Advances in Neural Information Processing Systems 11*, M.S. Kearns, S.A. Solla, and D.A. Cohn, Eds., to be published, MIT Press, Cambridge, MA.

Bontempi G. , Birattari M. & Bersini H. 1999. Lazy learning for local modeling and control design. *International Journal of Control.* vol. 72, no. 7/8, pp. 643–658.

Bontempi G. , Birattari M. 1999. *Toolbox For Neuro-Fuzzy Identification and Data Analysis. For use with Matlab* Technical Report: Iridia, Université Libre de Bruxelles. TR/IRIDIA/99-9.

Friedman J.H. 1994. *Flexible metric nearest neighbor classification.* Technical Report: Department of Statistics, Stanford University.

Myers R.H. 1990. *Classical and Modern Regression with Applications.* Boston, MA: PWS-KENT.

Perrone M.P. & Cooper L.N. 1993. When networks disagree: Ensemble methods for hybrid neural networks. *Pages 126–142 of:* Mammone R. J. (ed), *Artificial Neural Networks for Speech and Vision.* Chapman and Hall.

Schaal S. & Atkeson C.G. 1998. Constructive incremental learning from only local information. *Neural Computation*, **10**(8), 2047–2084.

Wolpert D. 1992. Stacked Generalization. *Neural Networks*, **5**, 241–259.