Matlab Functions for Radial Basis Function Networks

Mark J. L. Orr[1]

Institute for Adaptive and Neural Computation
Division of Informatics, Edinburgh University
Edinburgh EH8 9LW, Scotland, UK

June 30, 1999

Abstract

This manual is for a package of Matlab functions for nonparametric regression using radial basis function neural networks. The methods implemented employ techniques such as forward selection and ridge regression to control model complexity and regression trees to generate RBF centres and radii.

The first version of the package and manual was published on the web[2] in 1996. This is the second version[3] and incorporates some improvements and extensions. The theory behind the methods implemented in the package is covered in an introductory document[4] from 1996 plus an update[5] of developments between 1996 and 1999.

[1] mjo@anc.ed.ac.uk
[2] www.anc.ed.ac.uk/∼mjo/software/rbf.zip
[3] www.anc.ed.ac.uk/∼mjo/software/rbf2.zip
[4] www.anc.ed.ac.uk/∼mjo/papers/intro.ps
[5] www.anc.ed.ac.uk/∼mjo/papers/recad.ps

Contents

Tutorial Introduction

These manual pages are for a set of Matlab functions for nonparametric regression and classification (supervised learning) using radial basis function (RBF) networks. Several different methods have been implemented and they use either forward selection or ridge regression to control model complexity. Some of the methods use a regression tree to help select the centres and sizes of the hidden units in the network. This manual and the associated software are the second version of a package previously published on the web [5].

There are new versions of the methods based on forward selection (`rbf_fs_2`) and ridge regression (`rbf_rr_2`) which can now optimise the RBF widths. Two new methods, `rbf_rt_1` and `rbf_rt_2`, use regression trees (rather than the training set inputs) to create a set of candidate RBFs but differ in the way in which a subset is selected for the network. Each method has a set of parameters which can be configured or left to take default values. The methods all have demo programs and are supported by a collection of utility functions. Some of these may be of general use and are covered in this manual.

The rest of this tutorial introduction describes RBF networks and illustrates the basic operation of the methods by applying them to a toy problem. Elsewhere in the manual there are reference pages for the methods and their supporting utilities which explain their operation in greater detail and give examples of how to run them. Scripts for all examples can be found in the `man/examples` folder of the distribution. For brevity some minor details, such as the annotation of plot axes, have been skipped in the text of the manual, but the scripts contain every last command.

The latest version the software (and this manual), as well as other resources for RBF networks, can be found at

`http://www.anc.ed.ac.uk/~mjo/rbf.html`

Supervised Learning

Supervised learning, or what statisticians know as nonparametric regression, is the problem of estimating a function, $y(\mathbf{x})$, given only a training set of pairs of input-output points, $\{(\mathbf{x}_i, y_i)\}_{i=1}^{p}$, sampled, usually with noise, from the function. No knowledge of the true function is assumed, except that it is likely to be smooth (which tends to be true in practice) The software described in this manual is designed for multidimensional inputs ($\mathbf{x} \in \mathcal{R}^n$) but only for scalar outputs ($y \in \mathcal{R}$).

For the purposes of illustration we shall use a toy problem which is easy to plot, that is, one where the input variable, as well as the output, is a scalar ($n = 1$). Toy problems are useful not only for illustration but also for developing methods and testing software. One of the utilities, `get_data`, is specifically designed to produce data for a number of toy problems which it "knows about" by name, and we'll use 100 cases from it's Hermite data set for our examples in this introduction.

```
>> conf.name = 'hermite';
>> conf.p = 100;
```

```
>> [x, y] = get_data(conf);
```

By default `get_data` returns training data: randomly placed and unordered inputs and noisy outputs. However, it can also be configured to produce ordered and evenly spaced inputs and noiseless outputs suitable for use as a test set.

```
>> conf.p = 1000;
>> conf.ord = 1;
>> conf.std = 0;
>> [xt, yt] = get_data(conf);
```

Here `conf.p` is the number of cases, `conf.ord` controls input point ordering and `conf.std` sets the standard deviation of the output noise. Both training and test data are plotted in figure 1.

```
>> hold off
>> plot(xt, yt, 'k--')
>> hold on
>> plot(x, y, 'r*')
```

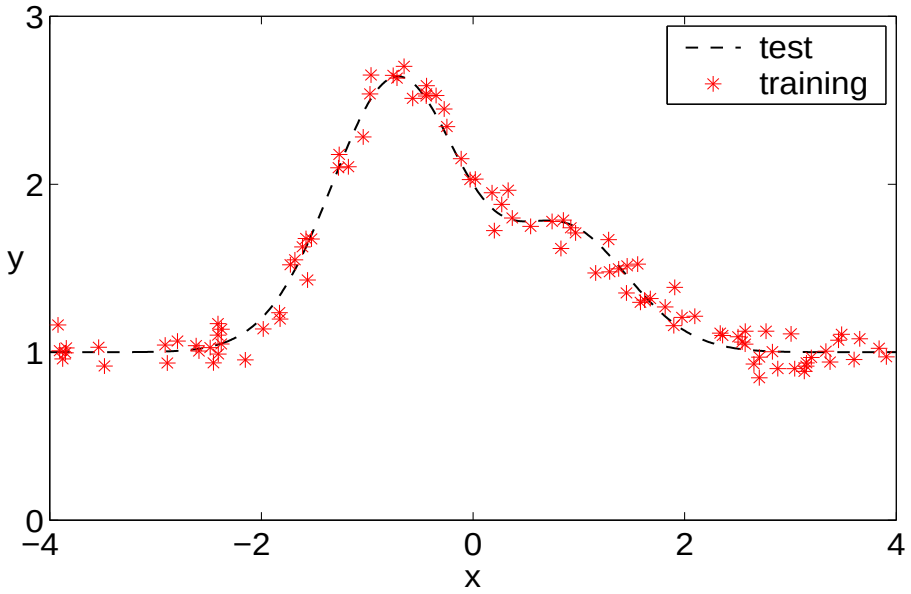

Figure 1: A sample of the Hermite data.

Linear Networks

Single-layer neural networks model functions with a linear equation of the form

$$f(\mathbf{x}) \;=\; \sum_{j=1}^{m} w_j \, h_j(\mathbf{x}) \,.$$

The linearity is with respect to the weights, $\{w_j\}_{j=1}^{m}$, not the input variable $\mathbf{x}$. The basis functions, $\{h_j\}_{j=1}^{m}$, are the transfer functions of the hidden units in the network (see figure 2).

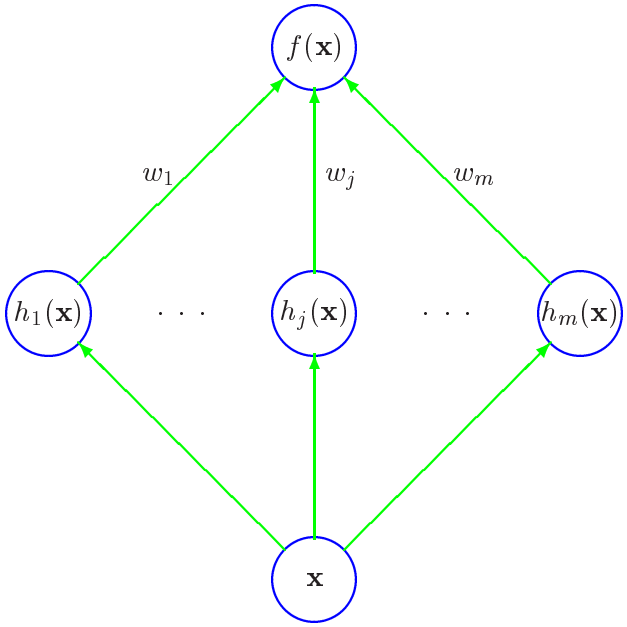

Figure 2: A single-layer neural network with input $\mathbf{x}$, hidden units h_j, weights w_j and output $f(\mathbf{x})$.

Radial Basis Functions

Radial basis functions [1] are one possible choice for the hidden unit transfer functions in a linear network. The response of an RBF to an input $\mathbf{x}$ decreases or increases smoothly and monotonically with distance between the input and the RBF's centre $\mathbf{c} \in \mathcal{R}^n$. The distance between two points is determined by the difference of their coordinates and by a set of parameters, $\mathbf{r} \in \mathcal{R}^n$, which scale each dimension. The square distance between $\mathbf{x}$ and $\mathbf{c}$ is

$$z^2 = \sum_{k=1}^{n} \frac{(x_k - c_k)^2}{r_k^2} .$$

This is not the most general linear distance metric, which would be

$$z^2 = (\mathbf{x} - \mathbf{c})^\top \mathbf{R}^{-1} (\mathbf{x} - \mathbf{c}) ,$$

and have at least $\frac{1}{2}n(n+1)$ free parameters in the matrix $\mathbf{R}$, more than the n in the vector $\mathbf{r}$. However, using $\mathbf{r}$ rather than $\mathbf{R}$, represents a fair compromise between the need to have scaling parameters and the difficulty of optimising too many of them. In effect we are restricting $\mathbf{R}$ ro be diagonal.

The package function for calculating RBF responses is `rbf_dm`. Its main configuration field is the type of RBF (see the manual page for details). For illustration, figure 3 depicts several different types of RBF in 1D all centred at $c = 0.5$ and with width $r = 0.2$. It was produced with the following code.

```
x = linspace(0, 1, 100);
c = 0.5;
r = 0.2;
h1 = rbf_dm(x, c, r, struct('type', 'm'));
h2 = rbf_dm(x, c, r, struct('type', 'c'));
h3 = rbf_dm(x, c, r, struct('type', 'g'));
hold off
plot(x, h1, 'r-')
hold on
plot(x, h2, 'g-')
plot(x, h3, 'b-')
```

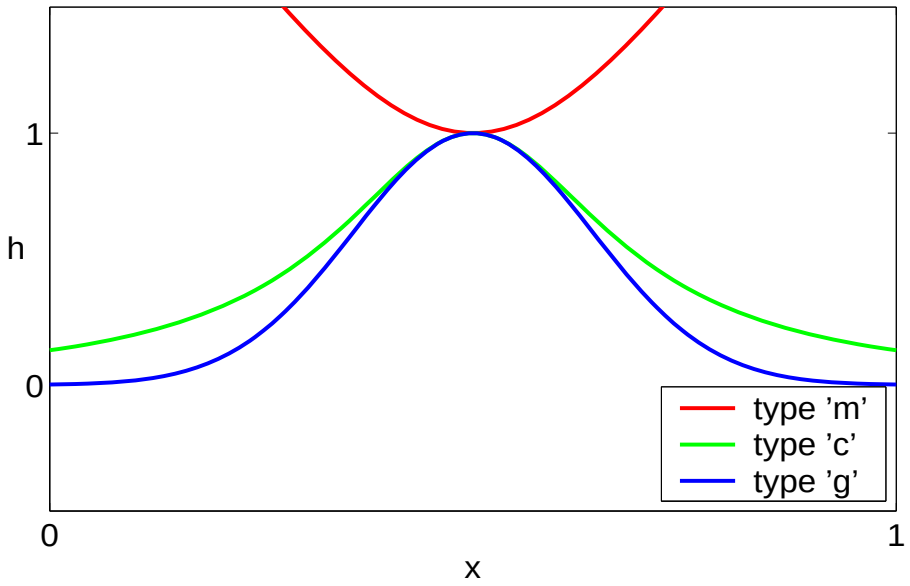

Figure 3: Three different types of 1-dimensional radial functions.

Least Squares Training

Given a network consisting of m RBFs with centres $\{\mathbf{c}_j\}_{j=1}^m$ and radii $\{\mathbf{r}_j\}_{j=1}^m$ and a training set with p patterns, $\{(\mathbf{x}_i, y_i)\}_{i=1}^p$, the optimal network weights can be found by minimising the sum of squared errors

$$E = \sum_{i=1}^{p} (f(\mathbf{x}_i) - y_i)^2 .$$

This leads to a set of p linear equations in m unknown weights and can be solved with the so-called *normal equation*,

$$\mathbf{w} = \left(\mathbf{H}^\top \mathbf{H}\right)^{-1} \mathbf{H}^\top \mathbf{y},$$

covered in most text books on regression analysis (e.g. [7]). Here $\mathbf{H}$ is the *design matrix*, its elements are $H_{ij} = h_j(\mathbf{x}_i)$ and $\mathbf{y} = [y_1\ y_2\ \ldots\ y_p]^\top$ is the p-dimensional vector of training set output values.

The function `rbf_dm` can be used to generate the design matrix for both the training and testing of an RBF network, as in the following illustration. We'll use the Hermite data set (figure 1) again. The training and test sets are obtained by:

```
[x, y, conf] = get_data('hermite');
conf.p = 1000;
conf.ord = 1;
conf.std = 0;
[xt, yt] = get_data(conf);
```

We assume (dubiously, as will soon become apparent) that each training set input contributes one RBF centre to the network and that all RBFs have the same width $r = 0.4$. The next block of code calculates the design matrix.

```
c = x;
r = 0.4;
H = rbf_dm(x, c, r);
```

Without a fourth input argument to configure it, `rbf_dm` sets the RBF type to a default, which is type `'g'` (Gaussian). Next we solve the normal equation. Anticipating numerical problems, we use the pseudo-inverse.

```
w = pinv(H' * H) * (H' * y);
```

Finally, we test the network by calculating the design matrix for the test set and multiplying it by the weight vector learned from the test set. This effectively calculates $f(x) = \sum_{j=1}^{m} w_j \, h_j(x)$ for all the inputs x in the test set.

```
Ht = rbf_dm(xt, c, r);
ft = Ht * w;
```

The result is plotted in figure 4 which compares `ft` (predicted) and `yt` (target).

```
hold off
plot(xt, yt, 'k--')
hold on
plot(xt, ft, 'r-')
```

Clearly something is not quite right as the fit is too rough. The reason for this is that the model we used, with as many unknowns (weights) as there are constraints (training set cases), is far too complex and is overfitting the data (fitting the noise as well as the signal). The problem of matching the complexity of the model to the data is sometimes known as the bias-variance dilemma and must be addressed by all nonparametric regression or classification methods [2]. The solution, in the case of RBF networks, is to be much more subtle about the choice of centres and radii or to take other steps to control the complexity of the model.

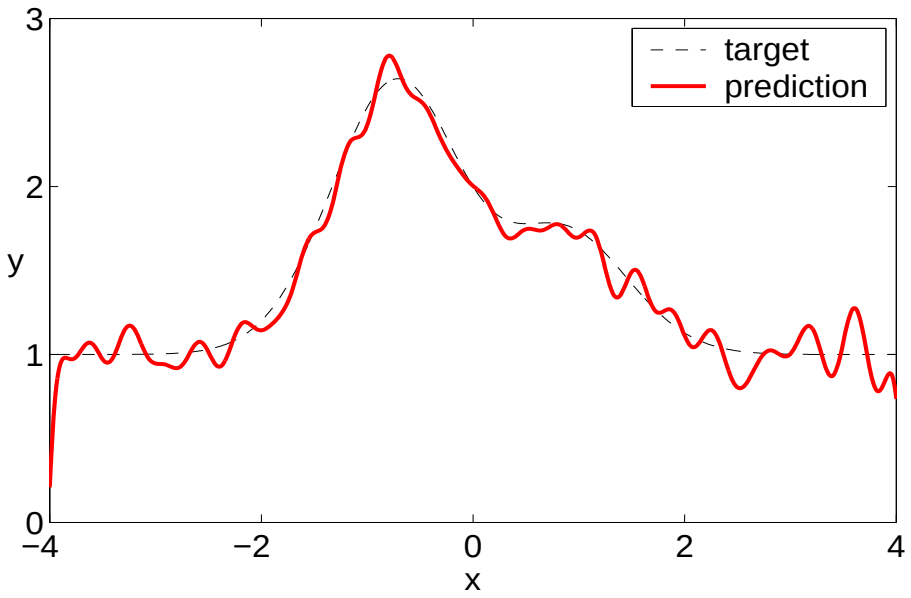

Figure 4: Predicting the Hermite function with a naive choice for the number of centres.

The different methods provided in the package, `rbf_fs_2`, `rbf_rr_2`, `rbf_rt_1` and `rbf_rt_2`, implement different strategies for dealing with this problem. Their manual pages, as well as explaining how to run the software, continue the Hermite example and show how better fits than that shown in figure 4 can be obtained.

Model Selection Criteria

Minimising the mean square error of the training set is unlikely to achieve good results on unseen test data because the model will pay too much attention to unique peculiarities of the training set (primarily the noise). Consequently, it's important to be able to estimate the likely mean square error on unseen data and to optimise this quantity instead. Over the years, statisticians have developed a number of such estimates, often called model selection criteria because they can be used to choose between competing models.

The methods in this software package can be configured to use a variety of different model selection criteria via the `msc` field of their configuration structure. Four of these criteria are based on modifying the training set sum-squared-error (SSE), to take account of the effective number of parameters in the model (γ). They are: unbiased estimate of variance (UEV), final prediction error (FPE), generalised cross-validation (GCV) and Bayesian information criterion (BIC).

$$\begin{aligned}
\mathrm{UEV} &= \frac{1}{(p-\gamma)}\,\mathrm{SSE}\,, \\
\mathrm{FPE} &= \frac{p+\gamma}{p\,(p-\gamma)}\,\mathrm{SSE}\,, \\
\mathrm{GCV} &= \frac{p}{(p-\gamma)^2}\,\mathrm{SSE} \\
\mathrm{BIC} &= \frac{p+(\ln(p)-1)\,\gamma}{p\,(p-\gamma)}\,\mathrm{SSE}\,,
\end{aligned}$$

where

$$\begin{aligned} \text{SSE} &= (\mathbf{y} - \mathbf{H}\,\mathbf{w})^\top (\mathbf{y} - \mathbf{H}\,\mathbf{w})\,, \\ \mathbf{w} &= \left(\mathbf{H}^\top \mathbf{H} + \lambda\,\mathbf{I}_m\right)^{-1} \mathbf{H}^\top \mathbf{y}\,, \\ \gamma &= m - \lambda \operatorname{tr} \mathbf{A}\,, \end{aligned}$$

and p is the number of training set cases. Here we are using ridge regression (also known as weight-decay) to penalise large weights and λ, which is known as the regularisation parameter, controls the strength of the penalty. Only GCV and BIC are effective, the others tend not to limit the model complexity enough and lead to overfitting. BIC often performs slightly better than the less conservative GCV. See [4] for more details.

Two other quite different model selection criterion are also offered as alternatives by some of the methods. Leave-one-out cross-validation (LOO) is given by

$$\text{LOO} = \frac{1}{p}\,\mathbf{y}^\top \mathbf{P}\,\operatorname{diag}(\mathbf{P})^{-2}\,\mathbf{P}\,\mathbf{y}\,,$$

where

$$\mathbf{P} = \mathbf{I}_p - \mathbf{H}\,\mathbf{A}^{-1}\mathbf{H}^\top\,,$$

and $\operatorname{diag}(\mathbf{P})$ is $\mathbf{P}$ with all off-diagonal terms set to zero. LOO is related to GCV and should perform similarly but is awkward to handle mathematically (because of the $\operatorname{diag}(\mathbf{P})$ term) and is only available in `rbf_rt_1`. In addition there's maximum marginal likelihood (MML), also known as the *evidence* [3], which is

$$\text{MML} = p \ln \sigma^2 - \ln |\mathbf{P}| + \frac{\mathbf{y}^\top \mathbf{P}\,\mathbf{y}}{\sigma^2}\,,$$

where σ^2 is an unbiased estimate of the noise (UEV). This is also slightly awkward because, being Bayesian, it only works for regularised networks, but it is offered as one of the alternative criteria for `rbf_rr_2`. See [6] for details.

References

[1] D.S. Broomhead and D. Lowe. Multivariate functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

[2] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[3] D.J.C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, 1992.

[4] M.J.L. Orr. Introduction to radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1996. `www.anc.ed.ac.uk/~mjo/papers/intro.ps`.

[5] M.J.L. Orr. Matlab routines for subset selection and ridge regression in linear neural networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1996. `www.anc.ed.ac.uk/~mjo/software/rbf.zip`.

[6] M.J.L. Orr. Recent advances in radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1999. `www.anc.ed.ac.uk/~mjo/papers/recad.ps`.

[7] J.O. Rawlings. *Applied Regression Analysis*. Wadsworth & Brooks/Cole, Pacific Grove, CA, 1988.

Methods

The Matlab functions in this section all implement methods of constructing RBF networks for supervised learning problems and all share the same set of input and output arguments. The currently available methods are listed below.

matlab function	*basic method*
`rbf_fs_2`	forward subset selection
`rbf_rr_2`	ridge regression
`rbf_rt_1`	regression trees
`rbf_rt_2`	regression trees

Running a Method

The pattern for calling any of the methods is the same.

```
[C, R, w, info, conf] = method(X, y, conf);
```

where `method` should be replaced by one of the function names in the table above. The various input and output arguments are as follows. Essentially, you supply the training data (`X, y`) and (optionally) configure the method (`conf`) and then the method hands back the centres and widths (`C, R`) of the hidden units in an RBF network plus some additional information (`info`) and confirmation of all configuration options used (`conf`), including those you didn't explicitly set but which took on default values.

`X, y` These input arguments specify the training set. If there are p cases and the input space is n-dimensional then `X` is an n-by-p matrix of input points and `y` is a p-by-1 vector of output values. Thus the i-th column of `X` corresponds to the input vector of the i-th case in the training set while the i-th component of `y` is the corresponding output value. Note that multi-dimensional outputs are not supported (other than by creating separate models for each output dimension).

`C, R, w` These output arguments specify the RBF network that the method has built: the centres `C`, the radii `R` and the weights `w`. If the network contains m RBF units then `C` is an n-by-m matrix, each column corresponding to one centre in input space. `R` is also an n-by-m matrix with each column corresponding to a set of n scaling parameters, one for each dimension, which determine the size of the m-th RBF function (see the manual page for `rbf_fm` for further details). `w` is either an m- or an $(m+1)$-dimensional vector depending on whether the method has included a bias unit in the network. `C`, `R` and `w` can be used to make predictions from the network (see the section on *Prediction* below).

`conf` This is both an input argument used to set method parameters and an output argument used to get them. It's a Matlab structure with fields corresponding to each parameter. Some of these are shared by all methods and some are special to individual methods. Every parameter has a default value and on input only

those taking non-default values need to have their `conf` fields set. The `conf` input argument can be entirely omitted in which case every parameter will take its default value. In the output `conf` all parameter fields are set and the structure can be examined to find out what values the method has used, including those it has set by default. See the section on *Configuration* below.

`info` This output argument provides additional information about the network built by the method. It's a structure with fields for each piece of information, some of which are common to all methods and some of which are specific to given methods (see the individual manual pages for details). `info` acts as a kind of catch-all for everything other than `C`, `R` and `w`. Some parts of `info` may be necessary to make predictions from the network, depending on how the method has been configured. See the section on *Prediction* below.

Prediction

To make predictions from the centres `C` and radii `R` returned by a method the first task is to build a design matrix for the test set inputs, `Xt`, with `rbf_dm` and then multiply this matrix by the weights `w`. A typical sequence would be

```
[C, R, w] = method(X, y);
H = rbf_dm(Xt, C, R);
yt = H * w;
```

where `yt` are the predicted outputs and, once again, `method` should be replaced with one of the method names. However, this sequence relies on agreement between some of the default parameter values used by the methods and by `rbf_dm`. If you configure a method to have a basis function type other than Gaussian with sharpness 2 or allow a bias unit in the network, then this information has to be conveyed to `rbf_dm`, via its own configuration argument, otherwise it will incorrectly assume the defaults (Gaussian type, sharpness 2 and no bias unit).

One way to do this is to use the fields in `conf` describing the type and sharpness of the basis functions and the field in `info` indicating the presence of a bias unit and to configure `rbf_dm` (see its manual page). However, for convenience, `info` always has a field called `dmc` which can be used directly to configure `rbf_dm`.

```
[C, R, w, info] = method(X, y, conf);
H = rbf_dm(Xt, C, R, info.dmc);
yt = H * w;
```

Even if you are using all the defaults, which makes the inclusion of `info.dmc` unnecessary, it's probably wise to get into the habit of doing things this way. Then you won't have to remember to reconfigure `rbf_dm` if you change your mind about using the defaults.

Configuration

As mentioned above, one way to find out about the configurable parameters of a method is to run the method with a fifth output argument, as in

```
[C, R, w, info, conf] = method(X, y, conf);
```

and then examine the fields of this structure each of which corresponds to a method parameter. Alternatively, to get the method to just print out a list of configuration options, run

```
method conf
```

The list of parameters will include their field names, a short description of their purpose, the type of value expected, a list of allowed values and a default value. If the default value is given as the empty matrix, this means that it depends on values set for other parameters (or input arguments) and in this case you must resort to actually running the method to find out what the default is. An error occurs if you try to set a value for a method parameter which does not satisfy its expected type or is not one of its allowed values. An exception to this rule is when the allowed values is given as the empty matrix. In that case the value need only have the correct type.

If you only want reminded about one particular parameter and can remember its field name you can use the command

```
method conf name
```

For example, here is a fragment from an interactive session which prints out the model selection criteria available for use in `rbf_rt_1`.

```
>> rbf_rt_1 conf msc
rbf_rt_1: configuration field 'msc'
Model selection criterion:
    field: msc
     type: string
  options: 'bic', 'gcv', 'loo'
  default: 'bic'
```

This shows that the the parameter `msc` must be a string, can only be `'bic'`, `'gcv'` or `'loo'` and that the default is `'bic'`. For more details about model selection see the corresponding section in the *Tutorial Introduction*.

Demonstrations

Each method has an associated demo program which walks you through the steps of applying the method to one or two data sets. To run a demo just type

```
method demo
```

but substitute `method` with the name of the method you want to demo.

The demo program pauses at various points and asks a question, perhaps just whether you want to continue or not, along with a list of possible answers (e.g. "yes or no?"). The first in the list is always the default and can be invoked by simply pressing the RETURN key. Otherwise, type enough characters of the initial letters of one of the alternatives to distinguish it from the others, (e.g. "n"), followed by RETURN, to invoke that answer. You can always type "h" (or "help") to get help or "q" (or "quit") to quit the demo.

rbf_fs_2

Synopsis

```
[C, R, w, info, conf] = rbf_fs_2(X, y, conf);
```

Description

This method performs forward selection [2] using orthogonal least squares [1] to build a model which uses only a subset of the available RBFs. Its optional features include ridge regression with a single regularisation parameter [3, 4] and the identification of the best overall scale size for the RBFs from supplied trial values [5].

Input and Output Arguments

These are the same for all methods and are covered in the introductory section on methods (see page 11).

Configuration

The fields of the configuration structure `conf`, which is used to set and get method parameters, are as follows.

`cen` Sets the centres and number of the RBF units and should be an n-by-m matrix where n is the dimension of the input space and m is the number of hidden units. Thus the columns of `cen` are the centres $\{\mathbf{c}_j\}_{j=1}^m$ discussed in the *Tutorial Introduction*. The number of rows in `cen` must agree with the number of rows of `X`, the training set inputs. By default `cen` is set to `X`. This is probably quite a good default to use unless there are a very large number of inputs, in which case it might be preferable to make `cen` a random subset of the columns of `X` to reduce the computational burden.

`rad` Sets the nominal size of the RBF units (the actual sizes also depend on `scales`). An n-by-m matrix specifying n scales for each of m RBFs is acceptable (n is the input space dimensionality). However, also acceptable is an n-by-1 vector, a 1-by-m vector or even a scalar if the sizes are the same for all RBFs, all dimensions or both. In the case where `rad` is a full n-by-m matrix then its columns are, apart from a global scale factor (see `scales`), the vectors $\{\mathbf{r}_j\}_{j=1}^m$ discussed in the *Tutorial Introduction*. In other cases `rad` is expanded to n rows and m columns either by copying an n-by-1 vector m times, a 1-by-m vector n times, or a scalar $n \times m$ times. By default `rad` is n-by-1 and the scale in each dimension is set to the span of the training set inputs ($r_k = \max_i(x_{ik}) - \min_i(x_{ik})$). This is okay as a default as long as you don't also use the default value for `scales` in which case the RBFs will probably be too wide. The default `rad` should be used in conjunction with some scale values smaller than 1.

`scales` A vector of positive scales to be applied to the RBF radii set by `rad`. Each value in `scales` is used to create a separate set of RBFs from which to select a subset for the network. The network with the lowest model selection score (see `msc`) wins and is returned by the method (and its scale returned in `info.scale`). By default `scales` just has one value, the number 1, so `rad` alone determines the hidden unit sizes. However, it is usually better to give `scales` a number of alternatives and to let the method choose the best one.

`bias` A switch which is either 1 (on) or 0 (off) and controls whether a bias unit can be selected. When `bias` is on then a bias unit is included, along with the RBFs, in the set of regressors from which the subset is selected, so a bias unit may appear in the final network. To check whether it has been, examine the value of `info.bias`. The default setting is off which means there will definitely not be a bias unit. This is fine if the mean of the training set outputs (`y`) is near zero since the bias unit is then unlikely to be picked. In other cases it may help to allow one.

`msc` A string specifying the type of model selection criterion. The options are: `'bic'` (Bayesian information criterion), `'gcv'` (generalised cross-validation), `'fpe'` (final prediction error) and `'uev'` (unbiased estimate of variance). See the section on model selection criteria in the *Tutorial Introduction*. The default is `'gcv'`. Another good choice is `'bic'` but the others are likely to overfit.

`thresh` Specifies the tolerance used to define convergence. Should be a positive integer value. As RBFs are added to the network the model selection criterion is tracked. When it decreases by less than the threshold amount (set by `thresh`) for a number (set by `conf.wait`) of consecutive iterations then the algorithm stops adding further RBFs to the network. To illustrate how the threshold works, suppose `thresh` is set to 10000, which is the default value. Then the model selection score has to decrease by more than one part in 10000 in order to stay above the threshold.

`wait` Specifies the number of iterations the decrease in model selection criterion has to stay below the threshold (set by `conf.thresh`) before the selection process is halted. Should be a small positive integer. It is possible for the decrease of model selection criterion to dip below the threshold for one or two iterations before increasing again and this parameter is designed to help prevent premature termination in such cases. The default value is 2.

`type` A string specifying the type of radial basis function. See the `rbf_dm` utility's manual page for the options. The default is `'g'`, which means Gaussian.

`exp` A number specifying the sharpness of the radial basis functions. See the `rbf_dm` utility's manual page for the options. The default is 2.

`lam` A number specifying the value of the regularisation parameter. If re-estimation is on (see `reest`) then this value is the initial one, otherwise it is held constant. By default `lam` is 10^{-9} and re-estimation is off. This set-up is designed to avoid numerical problems with ill-conditioned inverses which may arise when using the default values for `rad` and `scales` which create RBFs which are too large for low-dimensional data sets.

`reest` A switch which takes the values 1 (on) or 0 (off) and controls whether the regularisation parameter is re-estimated between iterations or remains constant. Turn it on to get *regularised forward selection* [3]. By default re-estimation is turned off.

`verb` A switch which takes the values 1 (on) or 0 (off). When on it causes the method to print out information as it processes a data set. Mainly used for debugging. By default, verbosity is turned off.

`timer` A string specifying part of the name of a window created and updated by the utilities `get_tmr` and `inc_tmr`. The purpose of this timer window is to give feedback on the progress made by the method as it processes each value in the list of scales. Useful for large data sets which take a long time to process. The default value is the empty string which prevents a timer being displayed.

Additional Information

The fields of the output argument `info`, which supplies additional information to supplement the centre positions (`C`), RBF radii (`R`) and weights (`w`), are as follows.

`scale` The best scale value found from amongst the alternatives in `conf.scales`.

`F` The full *design matrix* (see the manual page for `rbf_dm`) from which the subset was selected. Each column corresponds to one RBF, except the last column which, if `conf.bias` is on, is the bias unit.

`H` The design matrix of the selected RBFs. This saves having to recompute it from `rbf_dm(X, C, R, info.dmc)` or `info.F(:,info.subset)`.

`A` The weight variance matrix. In the absence of regularisation this is equivalent to `inv(H'*H)` where `H` is `info.H`. The relationship is more complicated when $\lambda \neq 0$ [4].

`U` An triangular matrix associated with orthogonal least squares [4].

`lam` The final value of the regularisation parameter (λ). If there is no re-estimation (if `conf.reest` has been turned off) then `info.lam` and `conf.lam` will be the same.

`gam` The final value of the effective number of free parameters (γ). This will be less than or equal to the number of columns in `info.H` (equal only if `conf.lam` is zero and there is no re-estimation).

`err` The final value of the model selection criterion.

`bias` Specifies which component of `w` refers to the bias unit, if any. If `bias` is zero then the bias unit was not selected or (if `conf.bias` is turned off) not allowed to be selected.

`subset` A list of indicies into the columns of `F` specifying the regressors which were selected and inserted into `H`.

`stats.comps` The number of floating point operations consumed by the function.

`stats.ticks` The running time in seconds for the function.

`dmc` A structure which can be used to configure `rbf_dm` to construct a design matrix for prediction.

Examples

We continue the Hermite example begun in the *Tutorial Introduction* for which sample training and test sets can be obtained by

```
[x, y] = get_data('hermite');
test.name = 'hermite';
test.p = 1000;
test.ord = 1;
test.std = 0;
[xt, yt] = get_data(test);
```

First we run the method without any special configuration.

```
[c, r, w] = rbf_fs_2(x, y);
```

and use the centres and radii to predict the outputs of the training set.

```
Ht = rbf_dm(xt, c, r);
ft = Ht * w;
```

The results are plotted by

```
hold off
plot(xt, yt, 'k--')
hold on
plot(xt, ft, 'r-')
```

and shown in figure 1.

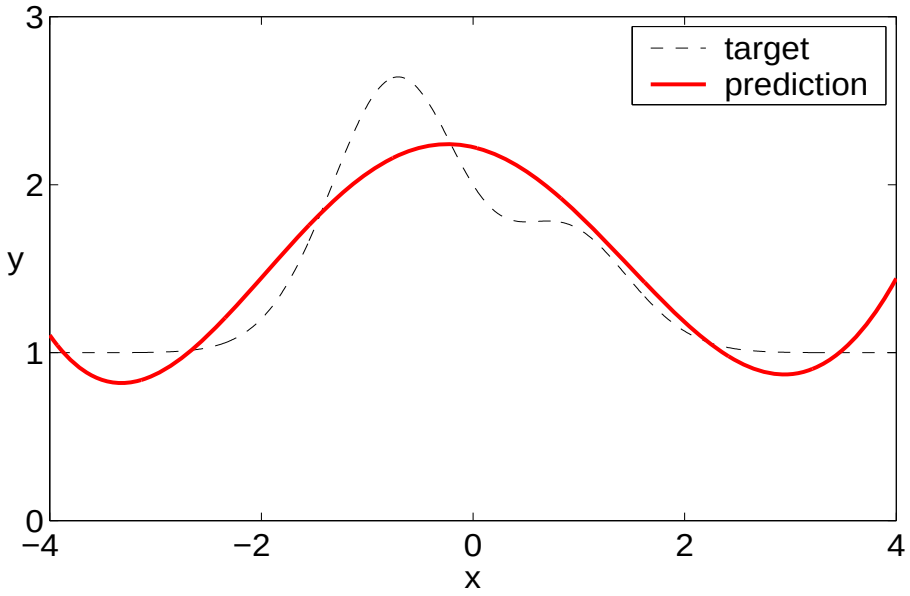

Figure 1: Hermite data with default scale.

Clearly, something is not right here, as the model appears to be too smooth (underfitting). The reason is that the RBFs from which the network is selected are too large and this is caused by the use of default RBF radii (see `conf.rad`) combined with the default scale size (see `conf.scales`).

To remedy the situation we run the method a second time, this time configuring a number of smaller alternative scales. Also, for no other reason than to demonstrate some of the method options, we change to Cauchy RBFs, allow a bias unit to be selected, and use BIC, instead of the default GCV, as the model selection criterion.

```
conf.scales = [1 0.5 0.2 0.1];
conf.type = 'cauchy';
conf.bias = 1;
conf.msc = 'bic';
[c, r, w, info] = rbf_fs_2(x, y, conf);
```

This time, because we are not using the default RBF type and allowing a bias unit we have to communicate this information to `rbf_dm` via its configuration structure. The most convenient way to do that is to make use of the `dmc` field of `info`. Actually, we recommend always configuring `rbf_dm` with `info.dmc`, even when the default configurations would have achieved the same result. This avoids the situation where you stop using defaults in `rbf_fs_2` (or any other method) but forget to configure `rbf_dm`.

```
Ht = rbf_dm(xt, c, r, info.dmc);
ft = Ht * w;
```

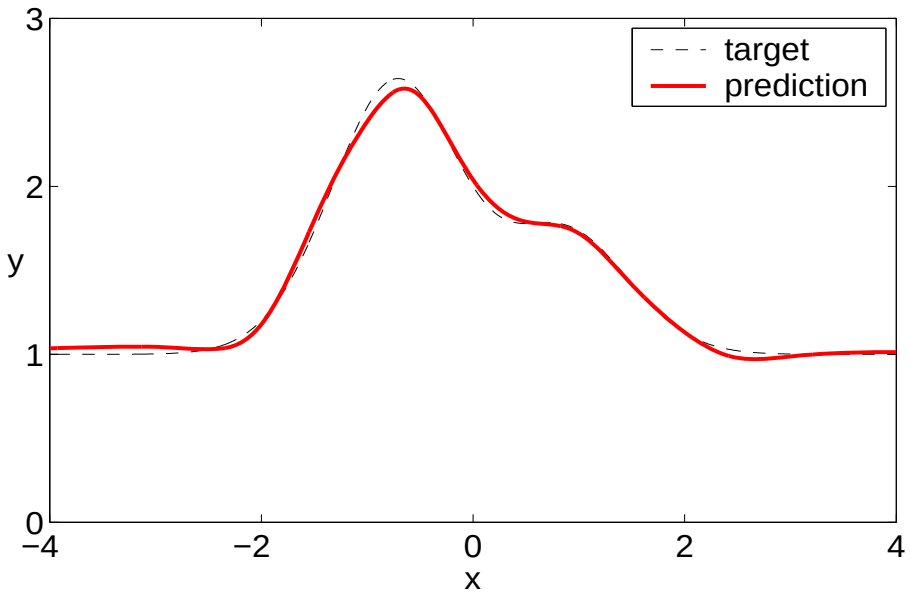

Figure 2: The result of allowing smaller scales.

The results are plotted as before and shown in figure 2. This is a much better fit and we can see why if we examine `info.scale` which contains the best scale value from amongst the alternatives in `conf.scale`.

```
>> disp(info.scale)
    0.1000
```

The preferred scale 0.1 is much smaller than the default 1.0.

Tips

If, after running the method and predicting some test set outputs, you find that the fit is really terrible then there a number of things you can do. You might be using RBFs which are too large, in which case adding some extra small values to `conf.scales` will help. Alternatively, try setting a small regularisation parameter (`conf.lam`) or turning on `conf.reest`.

Run the demo program with

```
rbf_fs_2 demo
```

To print a list of configuration parameters use

```
rbf_fs_2 conf
```

To print information about one particular parameter (e.g. `cen`) use

```
rbf_fs_2 conf cen
```

References

[1] S. Chen, C.F.N. Cowan, and P.M. Grant. Orthogonal least squares learning for radial basis function networks. *IEEE Transactions on Neural Networks*, 2(2):302–309, 1991.

[2] A.J. Miller. *Subset Selection in Regression*. Chapman and Hall, 1990.

[3] M.J.L. Orr. Regularisation in the selection of radial basis function centres. *Neural Computation*, 7(3):606–623, 1995.

[4] M.J.L. Orr. Introduction to radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1996. `www.anc.ed.ac.uk/~mjo/papers/intro.ps`.

[5] M.J.L. Orr. Recent advances in radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1999. `www.anc.ed.ac.uk/~mjo/papers/recad.ps`.

rbf_rr_2

Synopsis

```
[C, R, w, info, conf] = rbf_rr_2(X, y, conf);
```

Description

This method uses ridge regression to control model complexity and optimises the regularisation parameter with respect to the model selection criterion [1]. In addition it searches for the best overall RBF scale size from a number of trial values [2].

Input and Output Arguments

These are the same for all methods and are covered in the introductory section on methods (see page 11).

Configuration

The fields of the configuration structure `conf`, which is used to set and get method parameters, are as follows.

`cen` Sets the centres and number of the RBF units and should be an n-by-m matrix where n is the dimension of the input space and m is the number of hidden units. Thus the columns of `cen` are the centres $\{\mathbf{c}_j\}_{j=1}^m$ discussed in the *Tutorial Introduction*. The number of rows in `cen` must agree with the number of rows of `X`, the training set inputs. By default `cen` is set to `X`. This is probably quite a good default to use unless there are a very large number of inputs, in which case it might be preferable to make `cen` a random subset of the columns of `X` to reduce the computational burden.

`rad` Sets the nominal size of the RBF units (the actual sizes also depend on `scales`). An n-by-m matrix specifying n scales for each of m RBFs is acceptable (n is the input space dimensionality). However, also acceptable is an n-by-1 vector, a 1-by-m vector or even a scalar if the sizes are the same for all RBFs, all dimensions or both. In the case where `rad` is a full n-by-m matrix then its columns are, apart from a global scale factor (see `scales`), the vectors $\{\mathbf{r}_j\}_{j=1}^m$ discussed in the *Tutorial Introduction*. In other cases `rad` is expanded to n rows and m columns either by copying an n-by-1 vector m times, a 1-by-m vector n times, or a scalar $n \times m$ times. By default `rad` is n-by-1 and the scale in each dimension is set to the span of the training set inputs ($r_k = \max_i(x_{ik}) - \min_i(x_{ik})$). This is okay as a default as long as you don't also use the default value for `scales` in which case the RBFs will probably be too wide. The default `rad` should be used in conjunction with some scale values smaller than 1.

`scales` A vector of positive scales to be applied to the RBF radii set by `rad`. Each value in `scales` is used to create a separate network whose regularisation parameter is independently optimised. The network with the lowest model selection score (see `msc`) wins and is returned by the method (and its scale returned in `info.scale`). By default `scales` just has one value, the number 1, so `rad` alone determines the hidden unit sizes. However, it is usually better to give `scales` a number of alternatives and to let the method choose the best one.

`lambdas` This parameter specifies a list of (usually small) positive trial values to initialise the re-estimation of λ, the regularisation parameter. If only one trial value is supplied, as is the case with the default, which is $\lambda = 1$, then there is a risk that the global minimum (of the model selection criterion) will be missed if there is a nearby local minimum. To counter this situation multiple initial values for λ can be set up in `lambdas`. The algorithm which implements `rbf_rr_2` efficiently optimises multiple trial values and the extra computational burden is not great.

`bias` This is a switch with two possible values 1 (on) and 0 (off). If the switch is on then the method includes one bias unit in the first column of the design matrix (see `info.H`). The default setting is off which is fine if the mean of the training set outputs (`y`) is near zero. However, if the mean is not close to zero then it may help to have a bias unit.

`msc` A string specifying the type of model selection parameter. The options are: `'bic'` (Bayesian information criterion), `'gcv'` (generalised cross-validation), `'mml'` (maximum marginal likelihood), `'fpe'` (final prediction error) and `'uev'` (unbiased estimate of variance). See the section on model selection criteria in the *Tutorial Introduction*. The default is `'bic'`. Other good choices are `'gcv'` and `'mml'` but the others are likely to overfit.

`thresh` Specifies the tolerance used to define convergence. Should be a positive integer value. The method works by re-estimating new values for the regularisation parameter, λ, which reduce the model selection criterion. When the decrease between successive updates becomes smaller than a threshold amount (set by `thresh`) then the method assumes that no further improvement is possible, halts the re-estimations and calculates the network weights. To illustrate how the threshold works, suppose `thresh` is set to 10000, which is its default value. Then the model selection criterion has to decrease by more than one part in 10000 in order to stay above the threshold.

`hard` Sets a hard limit to the number of iterations allowed until convergence is reached. The default, 100, is rarely reached, especially if `cyc` is turned on.

`cyc` A switch which takes two values, either 1 (on) or 0 (off), which controls the operation of an anti-cycling heuristic. Sometimes the optimisation of λ, the regularisation parameter, gets trapped, cycling back and fourth between almost the same two values. An almost bi-stable state is reached and the `hard` limit is passed before it can be escaped. The result is that λ fails to reach its optimal value. Turning on `cyc` instructs the method to employ a heuristic designed to detect and escape such states. By default it is turned on.

`type` A string specifying the type of radial basis function. See the `rbf_dm` utility's manual page for the options. The default is `'g'`, which means Gaussian.

`exp` A number specifying the sharpness of the radial basis functions. See the `rbf_dm` utility's manual page for the options. The default is 2.

`reest` A switch which takes the values 1 (on) or 0 (off) and controls whether the regularisation parameter is re-estimated or whether the fixed values in `lambdas` are used. Normally this switch should be turned on since optimisation of the regularisation parameter is the main strategy of the method. By default, it is on.

`edm` A string which is either `'em'` or `'dm'` and switches between two different versions of the re-estimation formula if the model selection criterion is MML (maximum marginal likelihood). The EM (expectation-maximisation) formulae are guarenteed to converge while the DM (David MacKay) formulae are not. However, our empirical observation is that DM not only always converges but always does so faster than EM, which is why the default value of `edm` is `'dm'`. The theory is covered in [2].

`verb` A switch which takes the values 1 (on) or 0 (off) and causes the method to print out information as it processes a data set. Mainly used for debugging. By default, verbosity is turned off.

`timer` A string specifying part of the names of two windows created and updated by the utilities `get_tmr` and `inc_tmr`. The purpose of these timer windows is to give feedback on the progress made by the method as it processes the values in `scales` (outer loop) and `lambdas` (inner loop). Useful for large data sets which take a long time to process. The default value is the empty string which prevents any timers being displayed.

Additional Information

The fields of the output argument `info`, which supplies additional information to supplement the centre positions (`C`), RBF radii (`R`) and weights (`w`), are as follows.

`scale` The best scale value found from amongst the alternatives in `conf.scales`.

`lam` The optimal value of the regularisation parameter (λ).

`gam` The number of free parameter (γ) for the best scale and optimal λ.

`H` The design matrix corresponding to the best scale. This saves computing it again from `rbf_dm(X, C, R, info.dmc)`.

`A` The weight variance matrix. Equivalent to `inv(H'*H + lam*eye(m))` where `H` is `info.H`, `lam` is `info.lam` and `m` is the number of columns in `H`.

`err` The final value of the model selection criterion.

`bias` Is 1 if the first component of `w` (and column of `info.H`) refers to a bias unit. If `bias` is zero then there is no bias unit.

`lams` This vector gives the sequence of λ values, starting with one of the values in `conf.lambdas`, which led to the optimum. It is not present if `conf.reest` is turned off or if the model selection criterion is MML.

`sig`, `var`, `sigs`, `vars` Unlike the other model selection criteria, MML re-estimates the noise variance σ^2 and the prior weight variance ς^2 rather than the regularisation parameter λ, although they are related ($\lambda = \sigma^2/\varsigma^2$). So, if MML is the chosen criterion then the optimal values of σ^2 and ς^2 (respectively `sig` and `var`) along with the sequence of values which lead up to them (respectively `sigs` and `vars`) are returned in `info`. These fields are not present if `conf.reest` is turned off.

`errs` This vector gives the sequence of model selection criterion values, ending with a minimum, corresponding to the sequence of λ values in `lams` or σ^2 and ς^2 values in `sig` and `var` (depending on the model selection criterion). It is not present if `conf.reest` is turned off.

`stats.comps` The number of floating point operations consumed by the function.

`stats.ticks` The running time in seconds for the function.

`dmc` A structure which can be used to configure `rbf_dm` to construct a design matrix for prediction.

Examples

We continue the Hermite example begun in the *Tutorial Introduction* for which sample training and test sets can be obtained by

```
[x, y] = get_data('hermite');
test.name = 'hermite';
test.p = 1000;
test.ord = 1;
test.std = 0;
[xt, yt] = get_data(test);
```

The method does not work well on low dimensional examples without any configuration because the default values for `conf.rad` and `conf.scales` lead to RBFs which are too large. We'll run the method with some small scale values to overcome this problem. We'll also use GCV instead of the default BIC for the model selection criterion (to simplify the calculations for figure 2 below) and provide several initial λ values to cover the likely range in which the optimum will occur.

```
conf.msc = 'GCV';
conf.scales = [0.2 0.1];
conf.lambdas = 10.^[-2:-2:-10];
[c, r, w, info] = rbf_fs_2(x, y, conf);
```

Prediction is accomplished by the usual method of constructing a design matrix for the test set and multiplying by the weights.

```
Ht = rbf_dm(xt, c, r, info.dmc);
ft = Ht * w;
```

The result are plotted in figure 1 with

```
hold off
plot(xt, yt, 'k--')
hold on
plot(xt, ft, 'r-')
```

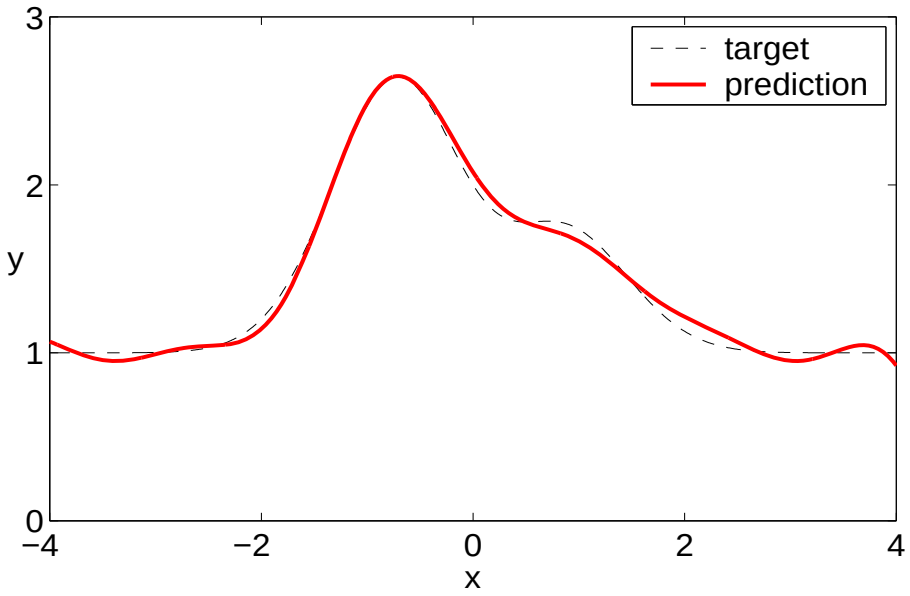

Figure 1: Hermite data modelled by `rbf_rr_2`.

The additional information in `info.lams` and `info.errs` is not particularly useful in a practical situation but does help to illustrate how the method works. In `info.lams` is the sequence of re-estimated λ values which led from one of the trial values (set up in `conf.lambdas`) to the lowest model selection score, while `info.errs` contains the corresponding scores. These are plotted in figure 2 with crosses.

```
hold off
plot(log10(info.lams), log10(info.errs), 'r+')
```

Figure 2 also plots the function which the method is effectively trying to minimise. To do this we first extracted the design matrix from `info.H` (this corresponds to the preferred scale, as do `info.lams` and `info.errs`) and then laboriously (and not very efficiently) calculated GCV for an array of λ values.

```
H = info.H; HH = H' * H; [p,m] = size(H);
lams = 10.^linspace(-10,0,50);
for i = 1:length(lams)
  A = inv(HH + lams(i)*eye(m));
  P = eye(p) - H * A * H';
  errs(i) = p * (y' * P) * (P * y) / trace(P)^2;
end
```

Then we plotting the function with

```
hold on
plot(log10(lams), log10(errs), 'm-', 'LineWidth', 2)
```

Note that the crosses lie on the curve, they start at one of the trial values set up in `conf.trials` ($\lambda = 10^{-4}$) and they end close to the global minimum. Also, because we used multiple trial values of the initial value of λ, the local minimum near $\lambda \approx 0.3$ was successfully avoided.

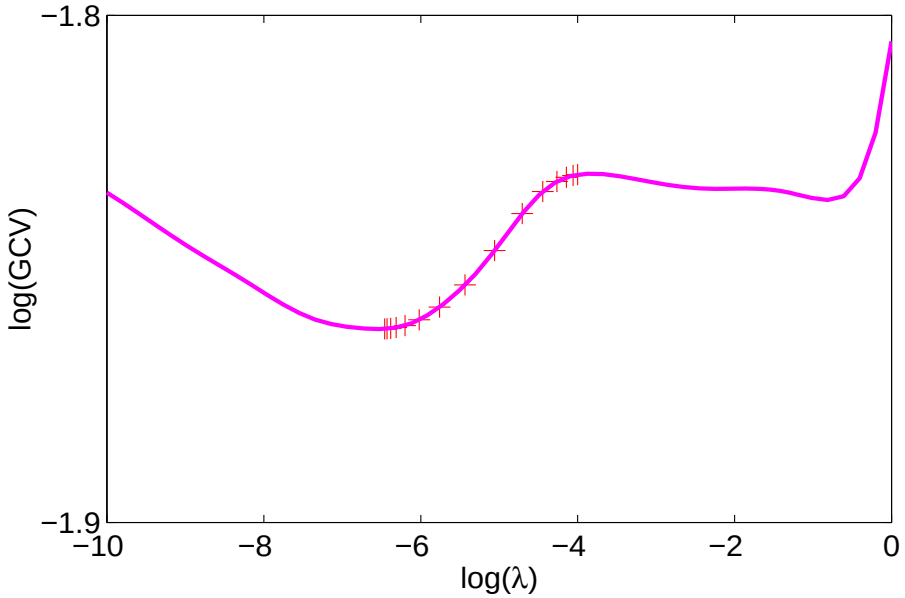

Figure 2: The relationship between λ and GCV.

Tips

It is possible for there to be no minimum of the model selection criterion when using BIC, GCV, FPE or UEV: it just keeps on decreasing as $\lambda \to 0$. The symptoms of this scenario, apart from a possibly very poor fit, are an extremely low value for `info.lams` and warnings (originating from the Matlab function `inv`) about unreliable results. Also, `info.err` may be infinite. The problem is usually caused by the RBFs being too numerous or too large.

For example, figure 3 shows the same plot as figure 2 except that we used the default scale when configuring `rbf_rr_2` (to deliberately make the RBFs too large). The crosses this time show the start of a sequence of λ values which eventually converged to a minimum GCV near $\lambda = 10^{-28}$. Possible solutions are too use less RBFs or to add some smaller values to `conf.scales`.

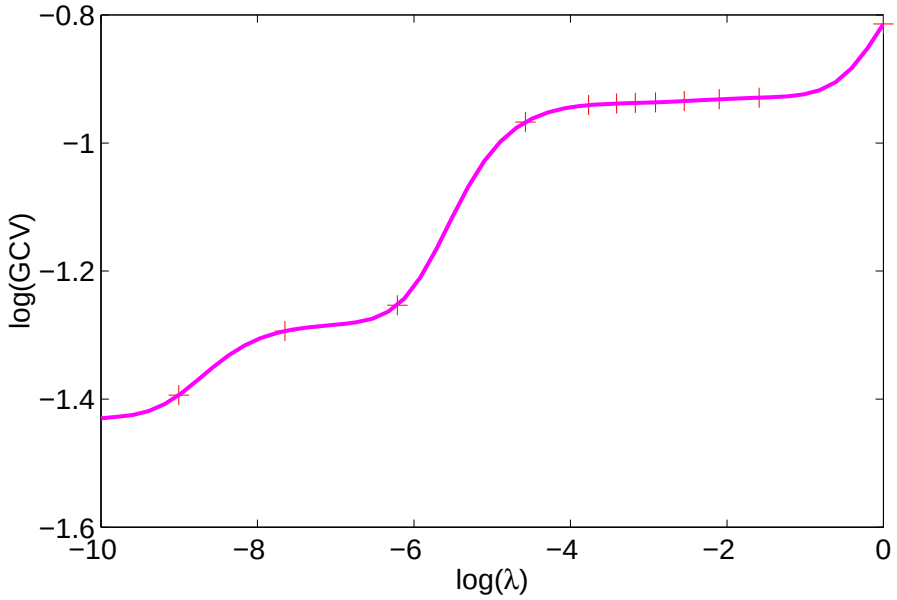

Figure 3: Sometimes there is no minimum.

Run the demo program with

```
rbf_rr_2 demo
```

To print a list of configuration parameters use

```
rbf_rr_2 conf
```

To print information about one particular parameter (e.g. `lambdas`) use

```
rbf_rr_2 conf lambdas
```

References

[1] M.J.L. Orr. Introduction to radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1996. `www.anc.ed.ac.uk/~mjo/papers/intro.ps`.

[2] M.J.L. Orr. Recent advances in radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1999. `www.anc.ed.ac.uk/~mjo/papers/recad.ps`.

rbf_rt_1

Synopsis

```
[C, R, w, info, conf] = rbf_rt_1(X, y, conf);
```

Description

This method first models the data with a regression tree then uses the nodes in the tree to determine the centres and radii of a set of RBFs. A subset of these are then selected for the final model. RBFs are considered for selection in a special order determined by the tree. Basically, large RBFs are considered first before smaller ones. The related method `rbf_rt_2` does not use any ordering, just plain forward selection (via `rbf_fs_2`) but, so far, has proved less effective than `rbf_fs_1`. See [4] for details.

Input and Output Arguments

These are the same for all methods and are covered in the introductory section on methods (see page 11).

Configuration

The fields of the configuration structure `conf`, which is used to set and get method parameters, are as follows.

`minmem` A list of small positive integers specifying alternative values for the minimum number of cases allowed in each regression tree node. A different tree is built for each value and each gives rise to a separate set of unscaled RBFs (see `scales` below). The trees are not pruned (because model complexity is controlled by RBF selection) so `minmem` is the only parameter which affects their depth. It can have a small effect on the performance of the method and in practice it usually pays to experiment with different sets of trial values to try and find one which works well on a given data set or collection of similar data sets. The default is the single value 5.

`scales` A vector of positive scales to be applied to the RBF radii determined by the regression tree node sizes. Each element of `scales` represents an alternative value with which to scale the size of the RBFs from the tree, and each gives rise to a separate set of RBFs from which to select the network. Since there may also be multiple trees (see `minmem`), the number of networks built by the method is equal to the product of the lengths of `scales` and `minmem`. The winning network is the one with the lowest model selection criterion score. Usually `scales` has a fairly significant effect on method performance and some experimentation to find values that work well is advisable. By default `scales` has two trial values, 1 and 2. Experience has shown that if the input space has a large number of dimensions then the best scale values are usually larger than these.

`msc` A string specifying the type of model selection parameter. The options are: `'bic'` (Bayesian information criterion), `'gcv'` (generalised cross-validation) and `'loo'` (leave-one-out cross-validation). See the section on model selection criteria in the *Tutorial Introduction*. The default is `'bic'`. The other choices, which are less conservative, do not appear to work quite as well.

`agcv` A positive number which specifies an adjustment to GCV, if that is the chosen model selection criterion (see `msc`). Some authors find that GCV is not conservative enough and have been led to modify it [1]. Let α be the number specified by `agcv`, then the version of GCV used is:

$$\text{GCV} \;=\; \frac{p}{(p - \alpha\,\gamma)^2}\,\text{SSE}\,.$$

The default is $\alpha = 1$ which produces standard GCV. Higher values lead to a more conservative criterion. In [1] values between $3 \leq \alpha \leq 5$ were used.

`type` A string specifying the type of radial basis function. See the `rbf_dm` utility's manual page for the options. The default is `'g'`, which means Gaussian.

`exp` A number specifying the sharpness of the radial basis functions. See the `rbf_dm` utility's manual page for the options. The default is 2.

`lambda` The fixed value of a regularisation parameter. Regularisation doesn't work particularly effectively with this method, probably because of the varied sizes of the RBFs. The default value is 0 which turns regularisation off. It might, however, be appropriate to set a small value for `lambda` (say, 10^{-10}) in order to avoid numerical problems if there are warning messages from the Matlab function `inv` about unreliable results.

`pseudo` A switch controlling the type of matrix inverse algorithm used and which takes either the value 1 (on) or 0 (off). By default `rbf_rt_1` uses the Matlab function `inv` to perform matrix inverses. This function complains if it's handed an ill-conditioned matrix (unless Matlab warnings are turned off). The problem can be addressed either by configuring a small amount of regularisation (see `lambda`) or by using `pinv`, the pseudo-inverse, instead of `inv`. This is what happens when `pseudo` is turned on. By default, it's off.

`speed` A switch controlling how model selection values are calculated and which takes either the value 1 (on) or 0 (off). When considering the next one or two RBFs to add to the growing model it is computationally more efficient to calculate the required matrix inverses by incremental operations [3]. This is what is done when this switch is on, as it is by default. Otherwise every inverse is calculated from scratch. It may be that incremental calculations introduce some numerical error, though probably not very much. Turn `speed` off if you're worried about this and don't mind the longer running time.

`rand` A switch controlling how ties are resolved and which takes either the value 1 (on) or 0 (off). Sometimes during the selection process, there are a number of equally good ways in which to expand the set of tree nodes whose RBFs are currently being considered for addition to the growing network (for the details of this mechanism see [4]). By default `rand` is turned on and such ties are resolved by randomly choosing one of the alternatives. This makes the

method non-deterministic (it can produce slightly different models in different runs on the same data). If this is undesirable, turn `rand` off. This causes the algorithm to always choose the first alternative and makes the method deterministic.

`edge` A switch controlling how tree nodes near the edge of the data space are turned into RBFs and which takes either the value 1 (on) or 0 (off). By default, when `edge` is off, the hyperrectangle associated with each node of the regression tree is converted into an unscaled RBF (see `scales`) by taking the RBF centre to be the middle of the hyperrectangle and the RBF radius (in each dimension) to be half the width of the hyperrectangle. An alternative scheme [2] is to shift the centres of RBFs corresponding to hyperrectangles which are near one or more edges of the data space towards those edges and double their radii in the corresponding dimensions. To use this alternative scheme turn `edge` on. In practice we have yet to find any conclusive evidence in favour of either method.

`verb` A switch which takes the values 1 (on) or 0 (off) and causes the method to be verbose and print out stuff as it goes along. Mainly used for debugging. By default, verbosity is turned off.

`rprt` A switch similar to `verb`. When on causes reports of the tree growing and RBF selection processes to be printed as the method is running. Used for debugging. Off by default.

`timer` A string specifying part of the name of two windows created and updated by the utilities `get_tmr` and `inc_tmr`. Their purpose is to give feedback on the progress made by the method as it processes the nested iterations over `minmem` (outer loop) and `scales` (inner loop). Useful for large data sets which take a long time to process. The default value is the empty string which prevents any timers being displayed.

Additional Information

The fields of the output argument `info`, which supplies additional information to supplement the centre positions (`C`), RBF radii (`R`) and weights (`w`), are as follows.

`minm` The best minimum membership value found from amongst the alternatives in `conf.minmem`.

`scale` The best scale value found from amongst the alternatives in `conf.scales`.

`err` The value of the model selection criterion for the best network.

`tree` This is a structure with many fields used internally by the function. It represents the tree which led to the best network and is suitable for input to `pred_tree`. Two fields which may be useful outside the function are listed below.

`tree.split.number` A vector equal in length to the number of input space dimensions and which lists the number of node splits in each dimension. The more splits in a dimension, the more important that component of the input is to

the regression relation. A component which is never split probably carries no information about the output.

`tree.split.order` A vector equal in length to the number of input space dimensions and which lists the order in which each component was first split. The earlier a component is split the more important it is to the regression relation. Components which are never split have a zero entry in this vector.

`rbf` This is a structure with many fields used internally by the function. Some of them may be useful outside the function as well.

`rbf.H` The design matrix of the selected RBFs. This is equivalent to `rbf_dm(X, C, R, info.dmc)`.

`rbf.A` The weight variance matrix. This is equivalent to `inv(H'*H+lam*eye(m))` where `H` is `info.rbf.H`, `lam` is `conf.lambda` and `m` is the number of columns in `H`.

`rbf.err` The value of the model selection criterion for the best network (same as `info.err`).

`rbf.gam` The effective number of parameters (γ). Equal to the number of hidden units unless `conf.lambda` is non-zero.

`stats.comps` The number of floating point operations consumed by the function.

`stats.ticks` The running time of the function in seconds.

`dmc` A structure which can be used to configure `rbf_dm` to construct a design matrix for prediction.

Examples

We continue the Hermite example begun in the *Tutorial Introduction* for which sample training and test sets can be obtained by

```
[x, y] = get_data('hermite');
test.name = 'hermite';
test.p = 1000;
test.ord = 1;
test.std = 0;
[xt, yt] = get_data(test);
```

Methods which use regression trees in general, and `rbf_rt_1` in particular, are really designed for multi-dimensional input spaces (they provide information about the relative importance of each input component). However, multi-dimensional input spaces are hard to depict in a 2D plot so, for the purposes of illustration, we persevere with this 1D example. Later we also show a 4D example and illustrate how to get an idea of the relative importance of the input components. The method is run on the Hermite data set by

```
[c, r, w, info] = rbf_rr_1(x, y);
```

The predictions are made by

```
Ht = rbf_dm(xt, c, r, info.dmc);
ft = Ht * w;
```

and plotted by

```
hold off
plot(xt, yt, 'k--')
hold on
plot(xt, ft, 'r-')
```

The use of `info.dmc` in `rbf_dm` is unnecessary, since we are using default RBF types and no bias (`rbf_rt_1` does not permit a bias unit) but recommended to avoid forgetting to configure `rbf_dm` if other types of RBF are used. The results are shown in figure 1.

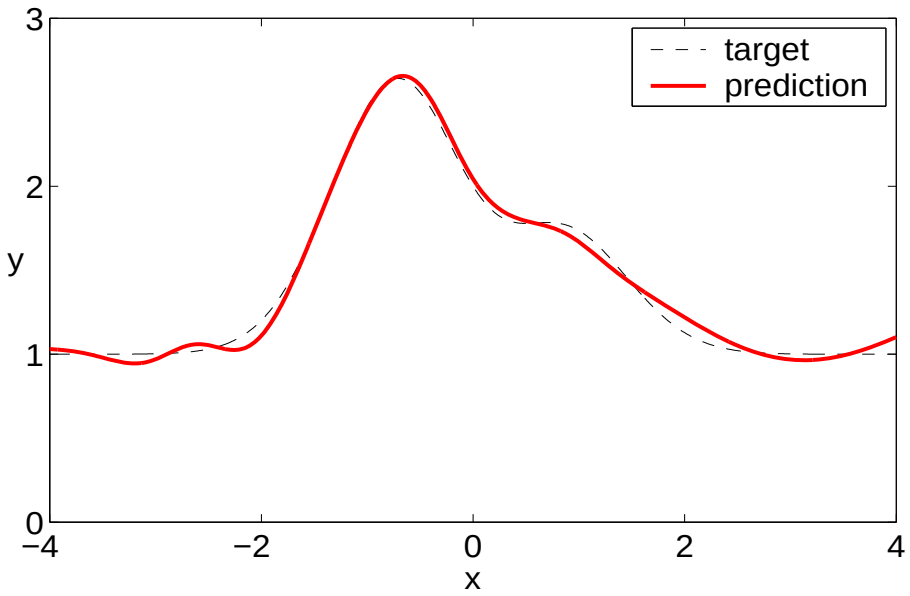

Figure 1: Hermite data with `rbf_rt_1`.

The fit is pretty good, especially since we used all the default configurations. In general the default `minmem` and `scales` are suitable for simple 1D problems like this but multi-dimensional problems usually require somewhat larger scales.

The tree returned as additional information in `info.tree` can itself be used for prediction using the function `pred_tree` as follows.

```
ft = pred_tree(info.tree, xt);
```

If we plot this prediction in the same way as we did for the prediction of `rbf_rt_1` we get figure 2. We show this only in order to help picture how the algorithm works. Actually using trees for prediction is not recommended for two reasons. Firstly, as can be seen from figure 2, the model estimated by the tree is discontinuous and

secondly, the tree has not been pruned (as is normally done with pure regression trees) because in rbf_rt_1 model complexity is controlled by RBF selection, not by the tree itself which merely generates candidate RBFs.

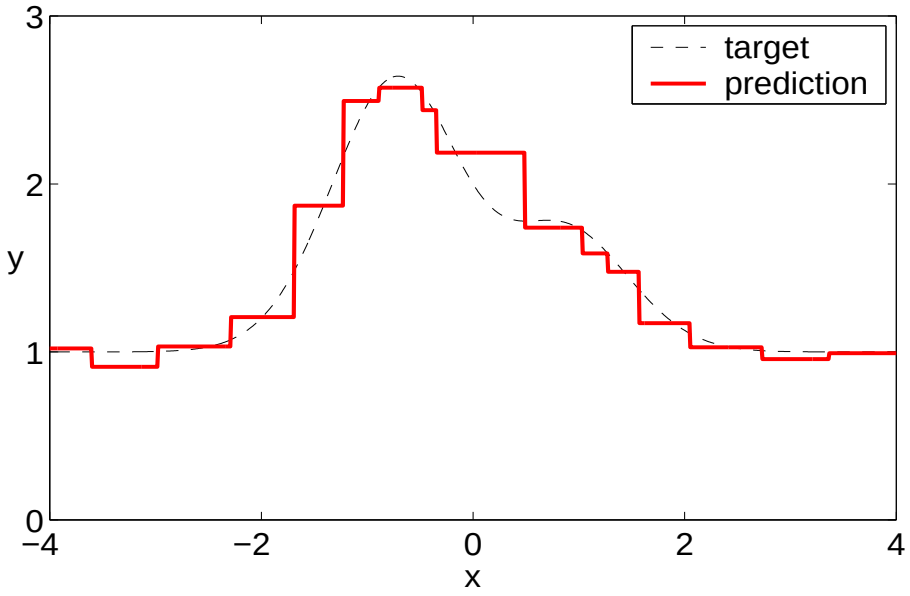

Figure 2: Hermite data predicted by the raw regression tree.

Finally we tackle a problem with 4 input dimensions in order to show a feature of this model which is not shared by rbf_fs_2 or rbf_rr_2: automatic relevance determination. The training data is obtained from get_data as follows.

```
[X, y] = get_data('friedman');
```

The method is configured with parameter values which have been found to work well on this problem and the graphical timers are turned on since it takes a while to finish (about one minute on my 266MHz Digital workstation).

```
conf.minmem = [3 4 5];
conf.scales = [7 8 9];
conf.timer = '4D';
[C, R, w, info] = rbf_rt_1(X, y, conf);
```

Next we display the split statistics.

```
>> disp(info.tree.splits.number)
     17     15      9     12
>> disp(info.tree.splits.order)
      1      4      3      2
```

For this particular data set there are no components which stand out as being particularly important or unimportant, although component 1 was both the first to be split and the most frequently split. In other multi-dimensional problems these split statistics can sometimes be used to identify irrelevant components which are seldom split and not amongst the first to be split.

Tips

Run the demo program with

```
rbf_rt_1 demo
```

To print a list of configuration parameters use

```
rbf_rt_1 conf
```

To print information about one particular parameter (e.g. `minmem`) use

```
rbf_rt_1 conf minmem
```

References

[1] J.H. Friedman. Multivariate adaptive regression splines (with discussion). *Annals of Statistics*, 19:1–141, 1991.

[2] M. Kubat. Decision trees can initialize radial-basis function networks. *IEEE Transactions on Neural Networks*, 9(5):813–821, 1998.

[3] M.J.L. Orr. Introduction to radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1996. `www.anc.ed.ac.uk/~mjo/papers/intro.ps`.

[4] M.J.L. Orr. Recent advances in radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1999. `www.anc.ed.ac.uk/~mjo/papers/recad.ps`.

rbf_rt_2

Synopsis

```
[C, R, w, info, conf] = rbf_rt_2(X, y, conf);
```

Description

This method first models the data with a regression tree then uses the nodes in the tree to determine the centres and radii of a set of RBFs. A subset of these are then selected for the final model by plain forward selection (using `rbf_fs_2`). The related method `rbf_rt_1` is identical except that it considers RBFs for selection in a special order. So far `rbf_rt_2` has proved inferior to `rbf_rt_1` in empirical tests. See [2] for details.

Input and Output Arguments

These are the same for all methods and are covered in the introductory section on methods (see page 11).

Configuration

The fields of the configuration structure `conf`, which is used to set and get method parameters, are as follows.

`minmem` A list of small positive integers specifying alternative values for the minimum number of cases allowed in each regression tree node. A different tree is built for each value and each gives rise to a separate set of unscaled RBFs (see `scales` below). The trees are not pruned (because model complexity is controlled by RBF selection) so `minmem` is the only parameter which affects their depth. It can have a small effect on the performance of the method and in practice it usually pays to experiment with different sets of trial values to try and find one which works well on a given data set or collection of similar data sets. The default is the single value 5.

`scales` A vector of positive scales to be applied to the RBF radii determined by the regression tree node sizes. Each element of `scales` represents an alternative value with which to scale the size of the RBFs from the tree, and each gives rise to a separate set of RBFs from which to select the network. Since there may also be multiple trees (see `minmem`), the number of networks built by the method is equal to the product of the lengths of `scales` and `minmem`. The winning network is the one with the lowest model selection criterion score. Usually `scales` has a fairly significant effect on method performance and some experimentation to find values that work well is advisable. By default `scales` has two trial values, 1 and 2. Experience has shown that if the input space has a large number of dimensions then the best scale values are usually larger than these.

`msc` A string specifying the type of model selection parameter. The options are: `'bic'` (Bayesian information criterion), `'gcv'` (generalised cross-validation) and `'fpe'` (final prediction error). `'uev'` (unbiased estimate of variance). See the section on model selection criteria in the *Tutorial Introduction*. The default is `'bic'`. Another good choice is `'gcv'` but the others are likely to lead to overfitting.

`type` A string specifying the type of radial basis function. See the `rbf_dm` utility's manual page for the options. The default is `'g'`, which means Gaussian.

`exp` A number specifying the sharpness of the radial basis functions. See the `rbf_dm` utility's manual page for the options. The default is 2.

`bias` A switch controlling the presence of a bias unit which has the value 1 (on) or 0 (off). When on, the forward selection algorithm (`rbf_fs_2`) is allowed to select a bias unit, otherwise not. By default it is off.

`edge` A switch controlling how tree nodes near the edge of the data space are turned into RBFs and which takes either the value 1 (on) or 0 (off). By default, when `edge` is off, the hyperrectangle associated with each node of the regression tree is converted into an unscaled RBF (see `scales`) by taking the RBF centre to be the middle of the hyperrectangle and the RBF radius (in each dimension) to be half the width of the hyperrectangle. An alternative scheme [1] is to shift the centres of RBFs corresponding to hyperrectangles which are near one or more edges of the data space towards those edges and double their radii in the corresponding dimensions. To use this alternative scheme turn `edge` on. In practice we have yet to find any conclusive evidence in favour of either method.

`trans` A string controlling which tree nodes get translated to candidate RBFs. The options are `'all'`, which means all tree nodes, and `'leaves'` which means only the terminal nodes.

`fsc` Internally, `rbf_rt_2` calls `rbf_fs_2` to perform forward selection and makes use of the latter's `cen` and `rad` configuration options to pass it the centres and radii of the tree generated RBFs. It also sets the `scales`, `msc`, `type`, `exp` and `bias` options of `rbf_fs_2` from it's own corresponding options, even if they have default values. The field `fsc` is to allow configuration of the other options in `rbf_fs_2`. For example, values for `fs.lam`, `fs.reest`, `fs.thresh` and `fs.wait` could be set, although generally speaking the defaults should suffice. See the manual page for `rbf_fs_2`.

`verb` A switch which takes the values 1 (on) or 0 (off) and causes the method to be verbose and print out stuff as it goes along. Mainly used for debugging. By default, verbosity is turned off.

`timer` A string specifying part of the name of two windows created and updated by the utilities `get_tmr` and `inc_tmr`. Their purpose is to give feedback on the progress made by the method as it processes the nested iterations over `minmem` (outer loop) and `scales` (inner loop). Useful for large data sets which take a long time to process. The default value is the empty string which prevents any timers being displayed.

Additional Information

The fields of the output argument `info`, which supplies additional information to supplement the centre positions (`C`), RBF radii (`R`) and weights (`w`), are as follows.

`minm` The best minimum membership value found from amongst the alternatives in `conf.minmem`.

`scale` The best scale value found from amongst the alternatives in `conf.scales`.

`err` The value of the model selection criterion for the best network.

`tree` This is a structure with many fields used internally by the function. It represents the tree which led to the best network and is suitable for input to `pred_tree`. Two fields which may be useful outside the function are listed below.

`tree.split.number` A vector equal in length to the number of input space dimensions and which lists the number of node splits in each dimension. The more splits in a dimension, the more important that component of the input is to the regression relation. A component which is never split probably carries no information about the output.

`tree.split.order` A vector equal in length to the number of input space dimensions and which lists the order in which each component was first split. The earlier a component is split the more important it is to the regression relation. Components which are never split have a zero entry in this vector.

`bias` If `conf.bias` is on and the bias unit has been selected in the best network then `info.bias` contains the index of the component in `w` which refers to the bias unit. Otherwise it is zero.

`info` In `rbf_rt_2` the networks are built by `rbf_fs_2` which returns an `info` structure. `info.info` is the additional information sent back from `rbf_fs_2` for the network which was selected by `rbf_rt_2` as best (whose centres, radii and weights are `C`, `R` and `w`). For example, `info.info.gam` is the number of free parameters. See the manual page for `rbf_fs_2`.

`stats.comps` The number of floating point operations consumed by the function.

`stats.ticks` The running time of the function in seconds.

`dmc` A structure which can be used to configure `rbf_dm` to construct a design matrix for prediction.

Examples

We continue the Hermite example begun in the *Tutorial Introduction* for which sample training and test sets can be obtained by

```
[x, y] = get_data('hermite');
test.name = 'hermite';
```

```
test.p = 1000;
test.ord = 1;
test.std = 0;
[xt, yt] = get_data(test);
```

The method is run with default configurations,

```
[c, r, w, info] = rbf_rr_1(x, y);
```

the test set is predicted,

```
Ht = rbf_dm(xt, c, r, info.dmc);
ft = Ht * w;
```

and the results plotted (see figure 1).

```
hold off
plot(xt, yt, 'k--')
hold on
plot(xt, ft, 'r-')
```

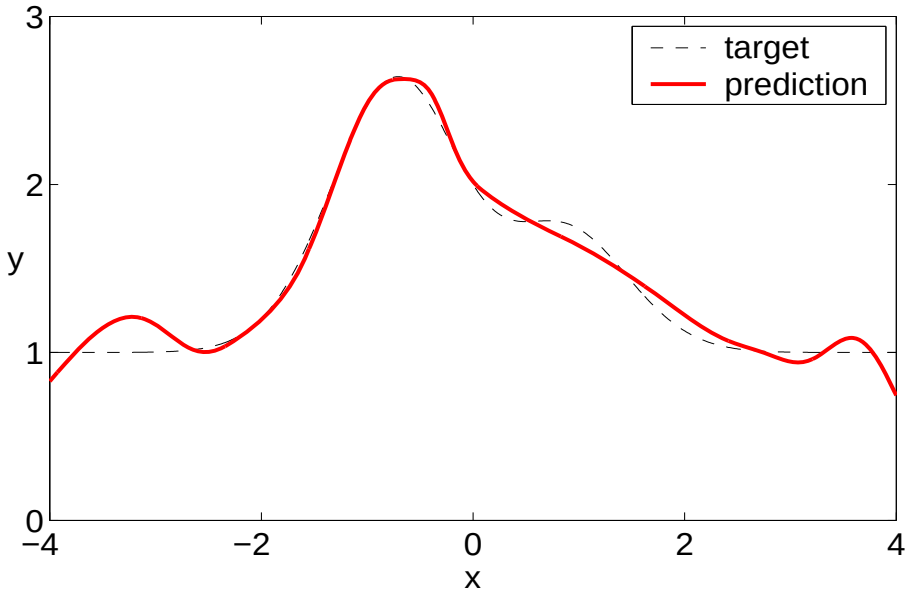

Figure 1: Hermite data with `rbf_rt_2`.

In the next example we'll run `rbf_rt_1` and `rbf_rt_2` head to head on 100 replications of the Hermite data to see which version has the best average test set error. Since 100 runs of both methods takes a little time (about 10 minutes on my 133MHz Windows 95 PC) we'll use a timer.

```
tmr = get_tmr(struct('name', 'fig 2', 'n', 2*100));
```

We'll use the same test set (`xt, yt`) as above but a different training set for each iteration of the main loop. Inside the loop both methods are run on the same training set and their prediction results are stored in the matrix `err`.

```
for rep = 1:100
  [x, y] = get_data('hermite');
  [c, r, w, info] = rbf_rt_1(x, y);
  Ht = rbf_dm(xt, c, r, info.dmc);
  ft = Ht * w;
  err(rep,1) = sqrt((yt - ft)' * (yt - ft) / length(yt));
  inc_tmr(tmr)
  [c, r, w, info] = rbf_rt_2(x, y);
  Ht = rbf_dm(xt, c, r, info.dmc);
  ft = Ht * w;
  err(rep,2) = sqrt((yt - ft)' * (yt - ft) / length(yt));
  inc_tmr(tmr)
end
```

The results are plotted in figure 2.

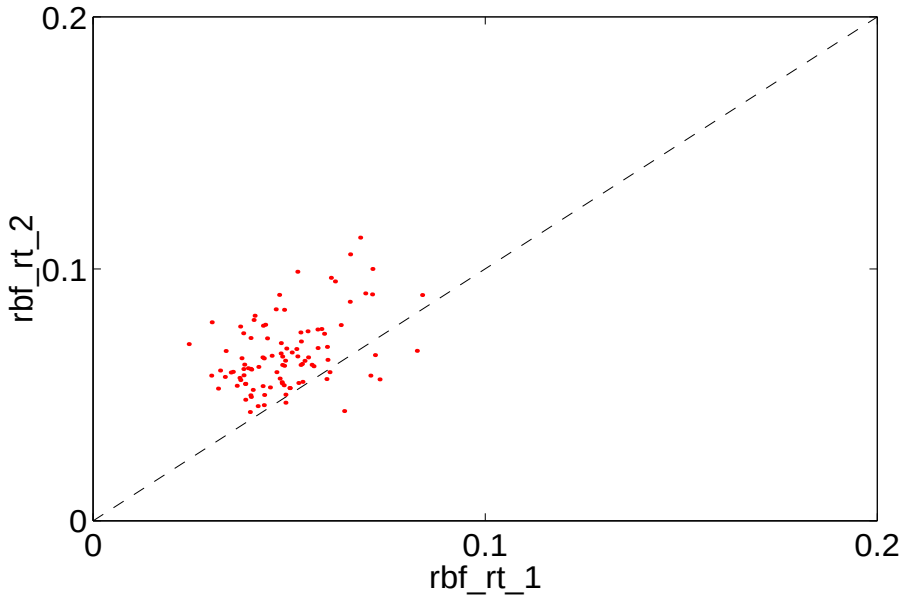

Figure 2: Relative performances of `rbf_rt_2` and `rbf_rt_2` on 100 replications of the Hermite data.

The basic plotting commands were

```
de = 0.1;
e1 = de*floor(min(min(err))/de);
e2 = de*ceil(max(max(err))/de);
hold off
plot([e1 e2], [e1 e2], 'k--')
hold on
plot(err(:,1), err(:,2), 'r.')
```

Clearly, since most points lie above the line, on average `rbf_rt_1` performs better than `rbf_rt_2`. Similar results have been obtained on other data sets leading us to the conclusion that the special selection order used in `rbf_rt_1` gives it an advantage.

Tips

Run the demo program with

```
rbf_rt_2 demo
```

To print a list of configuration parameters use

```
rbf_rt_2 conf
```

To print information about one particular parameter (e.g. `trans`) use (for example `trans`) with

```
rbf_rt_2 conf trans
```

References

[1] M. Kubat. Decision trees can initialize radial-basis function networks. *IEEE Transactions on Neural Networks*, 9(5):813–821, 1998.

[2] M.J.L. Orr. Recent advances in radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Division of Informatics, Edinburgh University, 1999. `www.anc.ed.ac.uk/~mjo/papers/recad.ps`.

Utilities

The Matlab functions in the *Methods* section are supported by a collection of utilities which implement common tasks such as setting up and incrementing timers (`get_tmr` and `inc_tmr`), generating data sets (`get_data`), computing design matrices for RBF networks (`rbf_dm`) and plotting Rasmussen diagrams (`ras_plot`).

Not all the utilities in the `util` folder of the distribution are covered in this manual, but those which are likely to be of general use have been given their own manual pages. They are listed in the table below.

matlab function	*used for*
`get_data`	generating data sets
`get_fig`	creating figure windows
`get_tmr`	creating graphical timers
`grow_tree`	growing regression trees
`inc_tmr`	incrementing timers
`pred_tree`	prediction using regression trees
`ras_plot`	plotting Rasmussen diagrams
`rbf_dm`	calculating RBF design matrices

In addition, there's a manual page for `rbf_ver` to highlight this important little script which prints the version number of this software package and the URL where the latest version can be found.

get_data

Synopsis

```
[X, y, conf] = get_data(conf, field);
```

Description

This utility is for quickly generating simulated data sets for testing learning methods. Internally, it stores the parameters of a few data sets and can recreate an instance of any of them simply by being given an identifying string. The parameters of any data set can be changed through fields in the configuration structure `conf` and this is how test sets (which usually have zero noise and ordered input values) are constructed. The function caches the last data set created so it can be retrieved if necessary.

Input Arguments

`conf` A structure used to configure the parameters of the data sets returned by the function. May be optionally omitted, in which case all parameters will take default values. See the section on *Configuration* for details.

`field` The name of a configuration field. Only used for printing configuration information (see *Tips* below).

Output Arguments

`X` A matrix of input vectors, one per column. The number of columns in `X` matches the length of `y`.

`y` A column vector of output values. The length of `y` matches the number of columns in `X`.

`conf` The full configuration structure with any parameters not explicitly set on input given their default values. Examine this structure to find out what the parameter defaults are. See also *Tips* below.

Configuration

The fields of the configuration structure `conf`, which is used to set and get optional parameters of the utility, are as follows.

`name` A string giving the name of the data set. Currently four alternatives are available: a 1D sine wave (`'sine1'`), a 2D sine wave (`'sine2'`), a 1D Hermite polynomial (`'mackay'` or `'hermite'`) [2] or a 4D simulated alternating current circuit with two different outputs (`'friedman'` or `'sacc'`) [1].

`p` The number of cases in the data set.

`x1, x2` The lower and upper limits for the input components. For multi-dimensional data sets these are column vectors of the same dimension as the input space. The components of `X` can be rescaled to all be in the range -1 to +1 (see `norm`) but this doesn't affect `x1` or `x2` which retain their original values.

`par` A vector specifying parameter values for the functions which generate the data sets. Only `'sine1'` and `'sine2'`, of the currently available data sets have such parameters. The following table lists all the functions and their parameters.

name	*function*	*parameters*
`'sine1'`	$y = \theta_1 \sin(\theta_2\, x)$	θ_1, θ_2
`'sine2'`	$y = \theta_1 \cos(\theta_2\, x_1) \sin(\theta_3\, x_2)$	θ_1, θ_2, θ_3
`'hermite'`	$y = 1 + (1 - x + 2\,x^2)\, e^{-x^2}$	none
`'friedman'`	$y_1 = \sqrt{x_2^2 + (x_1\, x_3 - (x_1\, x_4)^{-1})^2}$	none
`'friedman'`	$y_2 = \tan^{-1}((x_1\, x_3 - (x_1\, x_4)^{-1})/x_2)$	none

`comp` The index of the desired output component. Normally 1 since most data sets only have one output but can be 1 or 2 for `'friedman'`.

`std` The standard deviation of the Gaussian noise added to the outputs. In most cases this is a scalar but for `'friedman'`, which has two outputs, `std` is a two element vector giving the noise level for each. For uncorrupted test sets `std` needs to be set to zero.

`ord` A switch which is either 1 (on) or 0 (off). It controls whether the input values are ordered and evenly spaced out. By default `ord` is off for all data sets because the default action is to produce a training set where inputs are randomly sampled from within the input component extremes (see `x1, x2`). However, for creating test sets and particularly for plotting them, it can be useful to order the `X` values and space them out evenly. For this, `ord` should be switched on. For multidimensional data sets turning `ord` on causes the inputs to lie on a grid with the same number of cells in each dimension. If there are n dimensions the number of cases, p, may be adjusted so as to make $p^{1/n}$ a whole number. Therefor, in this situation, one should not rely on the `p` which is set in the input `conf` (or by default), but the one handed back in the output `conf`.

`norm` A switch which is either 1 (on) or 0 (off) and controls whether the input components are normalised or not. If the switch is on then each component is rescaled so its mean is close to zero and its values lie between -1 and 1. This doesn't affect `x1, x2` which retain their original values. At present `'friedman'` is the only data set which is normalised by default because its components have such diverse dynamic ranges.

Example

The default training set for the `'sine1'` data set is generated by

```
[x, y, conf] = get_data('sine1');
```

Some of the default parameter can be printed with

```
fprintf('conf.p = %d\n', conf.p)
fprintf('conf.x1,x2 = %.1f, %.1f\n', conf.x1, conf.x2)
fprintf('conf.par = ')
fprintf('%.1f ', conf.par)
fprintf('\n')
fprintf('conf.std = %.1f\n', conf.std)
```

which produces

```
conf.p = 100
conf.x1,x2 = -1.0, 1.0
conf.par = 0.8 6.0
conf.std = 0.1
```

The training data is plotted with red stars in figure 1.

```
hold off
plot(x, y, 'r*')
hold on
```

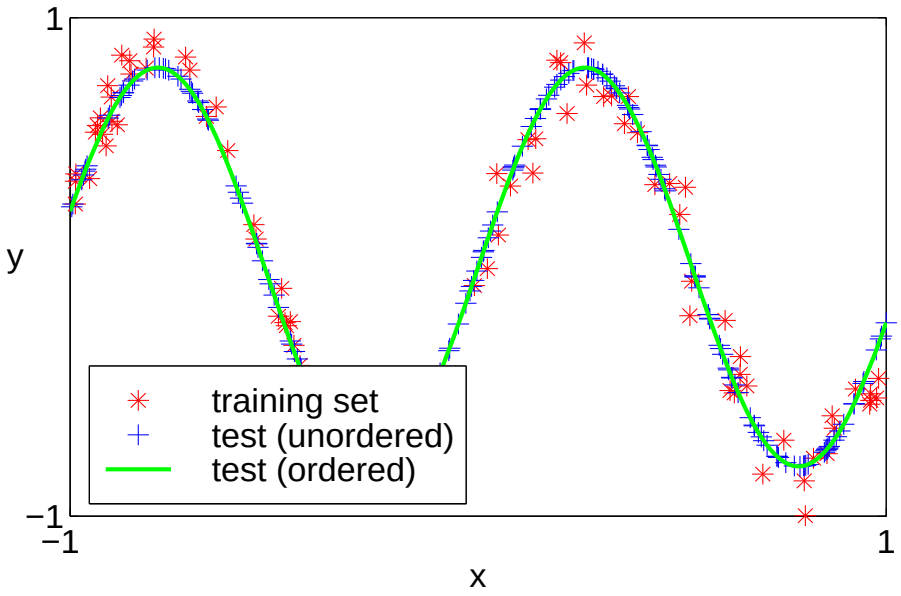

Figure 1: The `'sine1'` data set.

To produce a test set we set the noise to zero and double the number of cases. We could create a new configuration structure to set these parameters but it is convenient to just edit the one we already have.

```
conf.p = 200;
conf.std = 0;
[xt1, yt1] = get_data(conf);
plot(xt1, yt1, 'b+')
```

The test points are plotted in figure 1 with blue crosses. Notice that the x-values are in random positions but the y-values seem to lie on a smooth curve (because the noise is zero). The x-values are also unordered, which makes it difficult to use them for plotting a smooth curve. To order them we turn on `conf.ord`.

```
conf.ord = 1;
[xt2, yt2] = get_data(conf);
plot(xt2, yt2, 'g-')
```

This data is plotted by a smooth green curve in figure 1.

As a final example, figure 2 shows test and training data for the sine2 data set. This data was produced by

```
[X, y, conf] = get_data('sine2');
conf.ord = 400;
conf.std = 0;
conf.ord = 1;
[Xt, yt, conf] = get_data(conf);
```

and the plotting commands were

```
q = sqrt(conf.p);
x1 = linspace(conf.x1(1), conf.x2(1), q);
x2 = linspace(conf.x1(2), conf.x2(2), q);
Yt = zeros(q,q); Yt(:) = yt;
hold off
plot3(X(1,:), X(2,:), y, 'ko')
hold on
mesh(x1, x2, Yt)
```

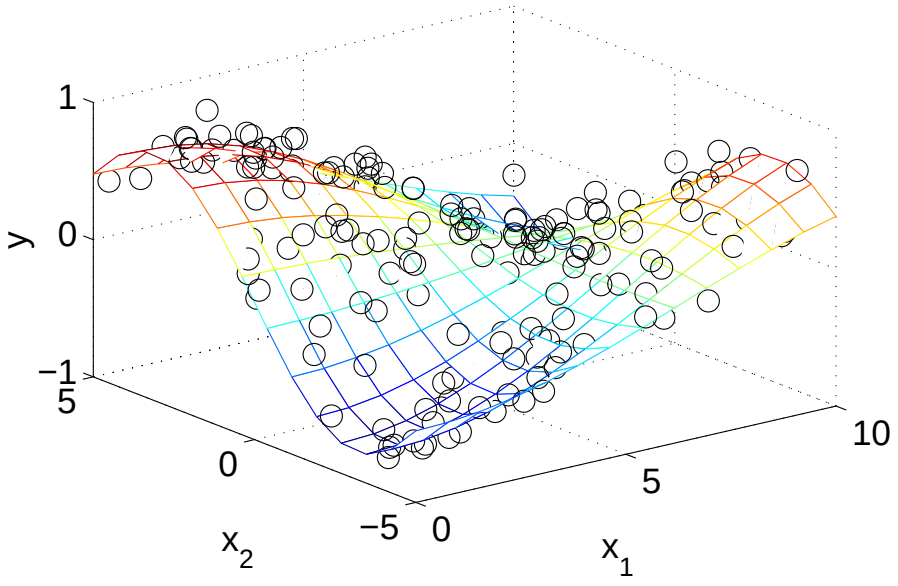

Figure 2: The `'sine2'` data set.

Tips

There's a shortcut for producing data with all the default configuration parameters. For example, if the data set is `'hermite'`, just run

```
[x, y] = get_data('hermite');
```

which is equivalent to

```
[x, y] = get_data(struct('name', 'hermite'));
```

You can print a list of configuration parameters with

```
get_data conf
```

or print information about a particular configuration parameter (for example `p`) with

```
get_data conf p
```

However, these don't tell you what the default values are because they depend on the data set name. The best way to find out what they are is to run the function for a particular data set and examine the fields of the output `conf` structure. For example,

```
>> [X, y, conf] = get_data('friedman');
>> disp(conf.p)
    100
```

References

[1] J.H. Friedman. Multivariate adaptive regression splines (with discussion). *Annals of Statistics*, 19:1–141, 1991.

[2] D.J.C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, 1992.

get_fig

Synopsis

```
[fh, conf] = get_fig(conf, field);
```

Description

This utility is a shortcut for the common operation of creating a figure window (or reusing one with the same name) with set sizes for screen position, screen size and printing size.

Input Arguments

`conf` A structure used to configure the parameters of the figures returned by the function. May be optionally omitted, in which case all parameters will take default values. See the section on *Configuration* for details.

`field` The name of a configuration field. Only used for printing configuration information (see *Tips* below).

Output Arguments

`fh` A figure handle.

`conf` The full configuration structure with all default values set. The structure can be examined to find out what the parameters and their defaults are. See also *Tips* below.

Configuration

The fields of the configuration structure `conf`, which is used to set and get optional parameters of the utility, are as follows.

`name` The title of the figure.

`pos` A row vector of length two specifying the screen position (in pixels) of the top left corner of the window. Default `[50, 50]`.

`size` A row vector of length two specifying the screen size (in pixels) of the window. Default `[600, 400]`.

`psize` A row vector of length two specifying the printing size (in cms) of the window. Default `[12, 8]`.

`num` A switch which takes one of two string values: `'on'` or `'off'`. When off this switch suppresses the inclusion of a number in the title of the window. By default the switch is off.

`menu` A switch which takes the value 1 (on) or 0 (off). When on, the menu bar at the top of the figure is suppressed. By default the switch is on.

Tips

Since often the only configuration parameter to be set is `conf.name`, a shortcut is available. For example, suppose the name is 'plot 1', then

```
fh = get_fig('plot 1');
```

is equivalent to

```
fh = get_fig(struct('name','plot 1'));
```

To print a list of configuration parameters use

```
get_fig conf
```

To print information about one particular parameter (for example, `name`), use

```
get_fig conf name
```

get_tmr

Synopsis

```
[fh, conf] = get_tmr(conf, field);
```

Description

This utility creates a long thin window which acts as a graphical timer to indicate the progress of a computation with a fixed number of iterations (see figure 1 for an example). The first return argument is a figure handle. The timer is incremented by the associated utility `inc_tmr` which is given the figure handle as its sole input argument. When the computation is finished the window can be closed with

```
close(fh)
```

Multiple timers can co-exist, which is useful for nested iterations, and `get_tmr` will try to position them down the left hand side of the screen so they don't overlap and obscure each other.

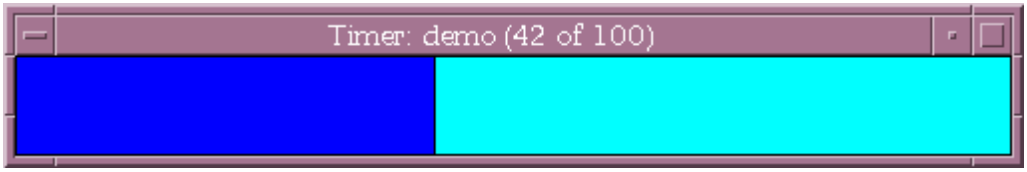

Figure 1: A graphical timer.

Input Arguments

`conf` A structure used to configure the parameters of the timer returned by the function. Must be included since the total number of increments must be specified. However, there is a shortcut if default values are sufficient for all other parameters (see *Tips* below). See the section on *Configuration* for details.

`field` The name of a configuration field. Only used for printing configuration information (see *Tips* below).

Output Argument

`fh` A figure handle that can be used as input to `inc_tmr`.

`conf` The full configuration structure with any parameters not explicitly set on input present in the structure with their default values. The structure can be examined to find out what the defaults are. However, there's a simpler way of doing this (see *Tips* below).

Configuration

The fields of the configuration structure `conf`, which is used to set and get optional parameters of the utility, are as follows.

`n` The number of iterations. This parameter must be specified.

`name` This field sets a title for the timer. By default the name is the empty string `''`.

`x` Horizontal position (in pixels) of the top left of the window. Default is 50.

`y` Vertical position (in pixels) of the top left of the window. The default depends on how many other timers are currently running.

`w` The width (in pixels) of the window, although it may be adjusted in order to be exactly divisible by the number of iterations. The default is 500.

`h` The height (in pixels) of the window. The default is 50.

Example

Here is an example of using two timers for a pair of nested loops.

```
t1 = get_tmr(struct('name','outer','n',10);
for i = 1:10
  t2 = get_tmr(struct('name','inner','n',10);
  for j = 1:10
    % some computation on i and j ...
    inc_tmr(t2)
  end
  close(t2)
  inc_tmr(t1)
end
close(t1)
```

Tips

If default values are sufficient for the other parameters there is a short cut for configuring the remaining parameter, `conf.n`. For example, instead of

```
fh = get_tmr(struct('n', 100));
```

you can just use

```
fh = get_tmr(100);
```

To print a list of configuration parameters use

```
get_tmr conf
```

To print information about a particular parameter (e.g. `w`) use

```
get_tmr conf w
```

grow_tree

Synopsis

```
[tree, conf] = grow_tree(X, y, conf);
```

Description

This function grows a regression tree [1] by recursive splitting of the input space. The tree is not pruned, which means this function alone is not suitable as a learning method because there is no mechanism to match the complexity of the model to the data. The utility is used internally by the methods `rbf_rt_1` and `rbf_rt_2` which control model complexity by means other than tree pruning.

Input Arguments

`X` A matrix which specifies the inputs of the training set. If there are p cases and the input space has n dimensions then `X` should have n rows and p columns, thus the columns of `X` are the vectors $\{\mathbf{x}_i\}_{i=1}^{p}$ (see the *Tutorial Introduction*).

`y` A vector specifying the outputs of the training set. If there are p cases then `y` should have p components, the scalars $\{y_i\}_{i=1}^{p}$ (see the *Tutorial Introduction*).

`conf` A structure used to configure the tree building algorithm. May be optionally omitted, in which case all parameters will take default values. See the section on *Configuration* for details.

Output Arguments

`tree` A structure representing the regression tree. Suitable as input to `pred_tree`.

`conf` The full configuration structure with any parameters not explicitly set on input given their default values. This structure can be examined to find out what the parameters and their defaults are. See also *Tips* below.

Configuration

The fields of the configuration structure `conf`, which is used to set and get optional parameters of the utility, are as follows.

`minm` A (usually small) positive integer specifying the minimum number of samples per tree node. No tree nodes with less than this number of samples will be created. The default is 5.

`place` A string which controls where the centres of tree nodes are placed and what their radii are in each dimension. These features of tree nodes are important because they are used to create the centres and radii of the hidden nodes in an RBF network by methods such as `rbf_rt_1` and `rbf_rt_2` but they don't

affect the recursive splitting algorithm in any way. The choices are `'centre'` or `'edge'`. In the first case centres are always places in the middle of a hyperrectangle and the width, in each dimension, is half the distance between opposite boundaries. In the `'edge'` case, hyperrectangles which have a single boundary on the edge of the data space in one or more dimensions have their centre shifted to lie on those boundaries and their width doubled in those dimensions. This implements a scheme proposed by [2]. However, there is no evidence yet to indicate an advantage for one choice over another. The default is `'centre'`.

`verb` A switch controlling verbosity which is either 1 (on) or 0 (off). Set to zero (the default) for the silent approach.

`rprt` A switch which is either 1 (on) or 0 (off) and controls whether a final report of the tree building activity is printed. Essentially just extra verbosity.

Example

To illustrate this function we'll use it as a crude learning method on the `'sine2'` data set of the `get_data` utility. By careful choice of the `minm` parameter a similar effect to pruning can be achieved and thus model complexity controlled. However, we must stress that `grow_tree` is primarily designed as a utility for `rbf_rt_1` and `rbf_rt_2`, not as a learning method in its own right.

First, we get the training data.

```
[X, y] = get_data('sine2');
```

Next we get the test data, making sure the inputs are ordered and the number of cases is a perfect square (this will come in handy when we later plot the function on a square grid).

```
test.name = 'sine2';
test.ord = 1600;
test.std = 0;
test.ord = 1;
[Xt, yt, test] = get_data(test);
```

Then we run `grow_tree` on the training data (using the default configuration) and predict using `pred_tree`.

```
tree = grow_tree(X, y, conf);
ft = pred_tree(tree, Xt);
```

To plot the original outputs `yt`, and the predicted outputs `ft`, we need to grid them and calculate the grid coordinates from `test.x1` and `test.x2`, configuration parameters of `get_data` which we didn't explicitly set but whose default values became available after `get_data` returned (they specify the extreme values of the input components).

```
q = sqrt(conf.p);
x1 = linspace(conf.x1(1), conf.x2(1), q);
x2 = linspace(conf.x1(2), conf.x2(2), q);
Yt = zeros(q,q); Yt(:) = yt;
Ft = zeros(q,q); Ft(:) = yt;
```

The original and predicted test data are plotted in figures 1 and 2 respectively.

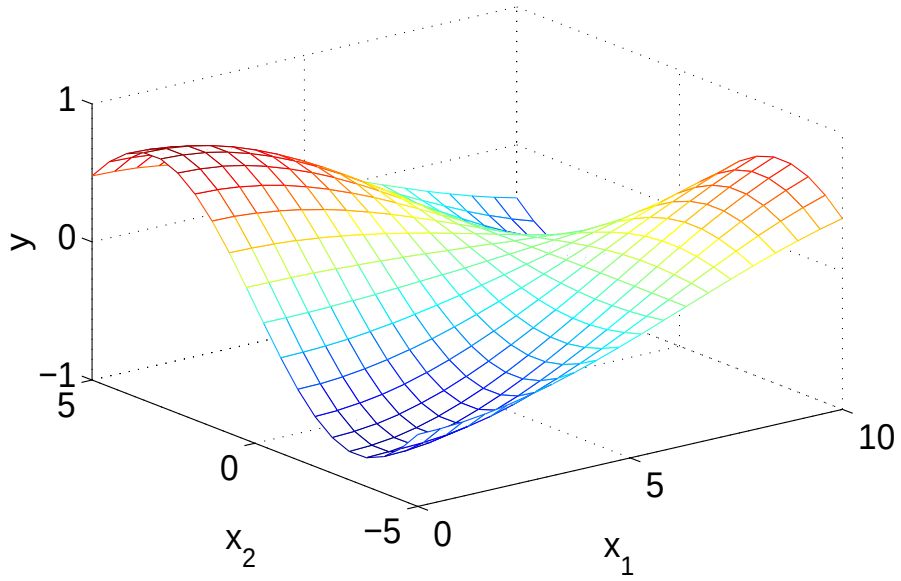

Figure 1: The original test data of 400 cases.

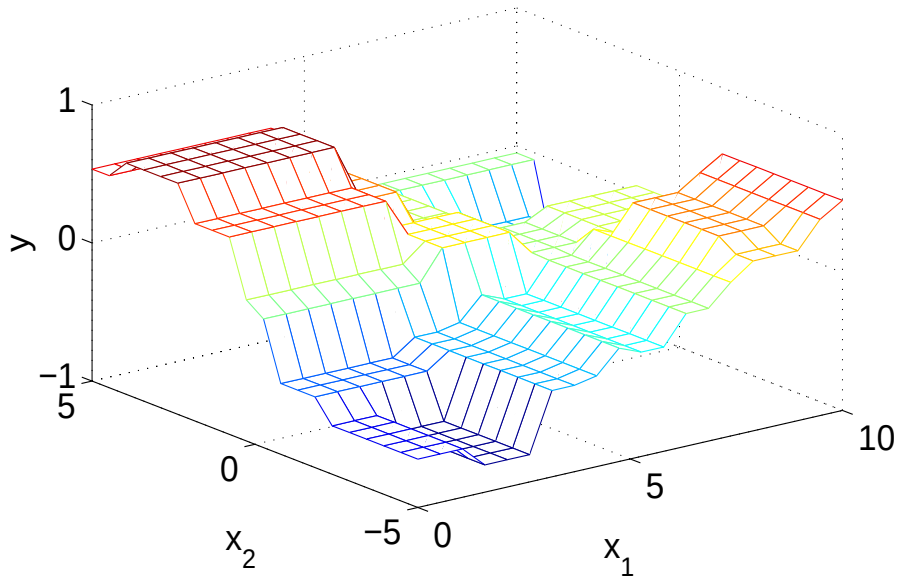

Figure 2: The test data predicted by `grow_tree` on the basis of 200 noisy training cases.

The plotting commands were

```
figure(1)
hold off
mesh(x1, x2, Yt)
figure(2)
hold off
mesh(x1, x2, Ft)
```

Note from figure 2 that the tree model is discontinuous. This is one of the drawbacks of using pure regression trees which must approximate the samples in each tree node with a constant function.

Tips

To print a list of configuration parameters use

```
grow_tree conf
```

To print information about a particular parameter (e.g. `minm`) use

```
grow_tree conf minm
```

References

[1] L. Breiman, J. Friedman, J. Olsen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, CA, 1984.

[2] M. Kubat. Decision trees can initialize radial-basis function networks. *IEEE Transactions on Neural Networks*, 9(5):813–821, 1998.

inc_tmr

Synopsis

```
inc_tmr(fh)
```

Description

This utility increments a graphical timer set up by the associated utility `get_tmr` and uses as its only input argument the figure handle returned by that utility. A timer is used to give feedback on the progress of an iterative computation.

Input Argument

`fh` A file handle returned from `get_tmr`.

Output Arguments

None.

Example

Here is an example of creating and incrementing two timers for a pair of nested loops.

```
t1 = get_tmr(struct('name','outer','n',10);
for i = 1:10
  t2 = get_tmr(struct('name','inner','n',10);
  for j = 1:10
    % some computation on i and j ...
    inc_tmr(t2)
  end
  close(t2)
  inc_tmr(t1)
end
close(t1)
```

plot_ras

Synopsis

```
plot_ras(data, methods, sizes, perfs, pvals, conf)
```

Description

This utility is for plotting Rasmussen diagrams. These are used to compare the average performance of methods on various sizes of training sets from the same source of data [2]. An example of such a diagram is shown in figure 1 using results from the DELVE archive.

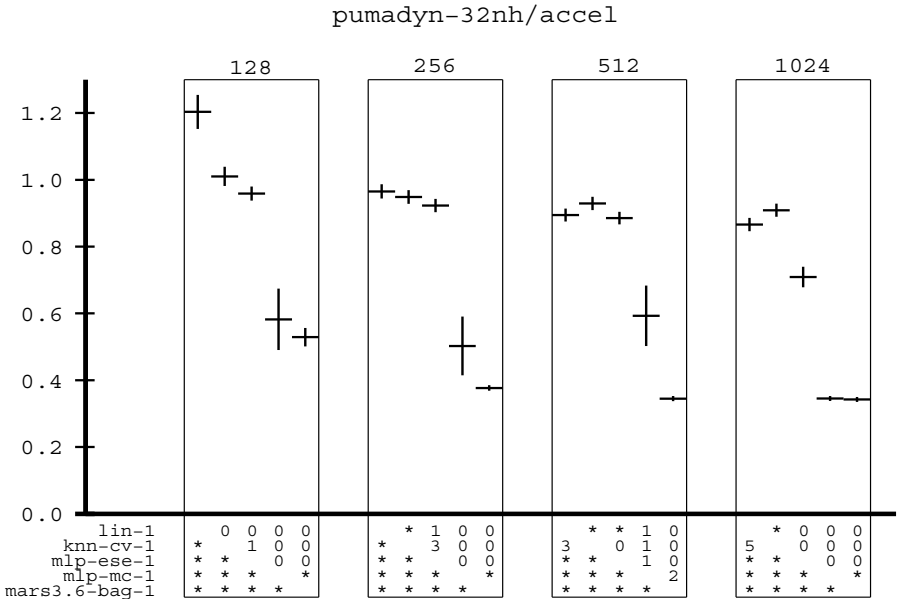

Figure 1: DELVE results plotted with `plot_ras`.

The title of the data set is at the top. In this case the data comes from the pumadyn family with 32 inputs and is non-linear with high noise. The name of the target component is accel.

Results for a given task are contained in a rectangle labelled at the top with the number of training set samples. The left-to-right ordering of columns within each rectangle is the same as the top-to-bottom ordering of method names lower down. The horizontal line for each method marks the mean normalised sum-squared-error and the vertical line depicts plus and minus the standard error.

Below each rectangle is an array of numbers. If method A performs better than method B and the probability that this difference could have occurred by chance (as calculated by a simple t-test) is less than or equal to 9 (when multiplied by 100 and rounded) then this figure appears in A's column and B's row. Otherwise a "*" character is inserted, or a blank if A and B are the same method. For example, the number 5 means that there is a 0.05 chance that A's better performance compared to B's could have arisen by chance. A method which has lots of small numbers in its column is performing well against the others.

Input Arguments

`data` A string which sets the title for the plot. In figure 1 the title string is `'pumadyn_32nh/accel'`.

`methods` A 1D cell array of method names (strings). In figure 1 the first name is `'lin_1'` and the last name is `'mars3.6-bag-1'`.

`sizes` A numeric vector specifying the different training set sizes. In figure 1 this is `[128 256 512 1024]`.

`perfs` This is a 3D matrix containing the mean and standard deviation of each method's scaled sum-square-error for each training set size. If there are s training set sizes and m methods, then the size of this matrix is s-by-m-by-2. In the third dimension the first component is the mean and the second is the standard deviation. These values are used to plot the upper half of the diagram.

`pvals` This is a matrix of probabilities for the numerical comparisons in the lower half of the diagram. You must calculate this from `perfs` or use the DELVE software (see *Tips* below). Values along the diagonal of `pvals` are ignored. Off-diagonal values should either contain a probability value between 0 and 1 or the number -1. The former case indicates that the method in the column performs better than the row method and that the probability of the difference being due to chance is the value given. The latter case indicates that either the column method performs less well than the row method or there is a significant probability that the difference between them is due to chance.

`conf` A structure used to configure parameters of the plot. May be omitted, in which case all parameters will take default values. See the section on *Configuration* for details.

Output Arguments

None.

Configuration

The fields of the configuration structure `conf`, which is used to set and get optional parameters of the utility, are as follows.

`width` The width of the plot in centimetres. The default is 12.

`height` The height of the plot in centimetres. The default is 8.

`ppc` The number of pixels per centimeter. The default is 50.

`tfs` The font size for the title string (the `data` input argument). The default is 14.

`afs` The font size for the axis annotations. The default is 12.

`mfs` The font size for the method names. The default is 10.

`ebc` A string specifying a set of colours for the error bars of each method. For example, `'rgb'` specifies red, green and blue for the first three methods. If there are more than three models the fourth will cycle back to red, and the next will be green and so on. The default is `'k'` which makes all error bars black.

`mnc` A string specifying a set of colours for the method names. Operates in a similar way to `ebc`. In fact, probably it is a good idea to have the same `ebc` and `mnc` so error bars and method names match in colour. The default is `'k'` which makes all names black.

`ebs` A vector of positive numbers specifying the widths of the lines used to plot the error bars for each method. For example, `[2 1 3]` specifies line widths of 2, 1 and 3 for the first three methods. If there are more than three methods then the next one cycles back to 2, then 1 and so on. The default is `[1]` which means all lines have width 1.

`maxse` A positive number specifying the maximum scaled error to be plotted on the vertical axis. Methods whose mean performance is worse than this will have their performance represented by an upward pointing arrow near the top of the plot.

Example

For the example we'll use a Rasmussen plot to compare three methods on some simulated data: a "stupid" method which just guesses the mean of the training set outputs for any input, `rbf_rt_1` and MARS. We'll compute results for the first two methods ourselves but get the MARS results from a paper [1].

First we set a name for our data set, the first argument to `plot_ras`.

```
data = 'friedman/Z';
```

`Z` is the name of the target being predicted. Next we will state our method names, the second argument to `plot_ras`.

```
methods = {'stupid', 'rbf_rt_1', 'mars'};
nmet = length(methods);
```

Next we specify a list of training set sizes, the third argument to `plot_ras`.

```
sizes = [100 200];
nsiz = length(sizes);
```

Now we'll begin the task of calculating the performance figures for the fourth input argument, `perfs`.

```
perfs = zeros(nsiz, nmet, 2);
```

We'll run the stupid method and rbf_rt_1 on replications of the training set and compute the mean and standard deviation of their scaled sum-square-error. First we obtain a test set. The utility get_data has been programmed with all the specifications of the data set, the simulated alternating current circuit from [1]. In order to scale the errors later, we also calculate the total variance of the test set outputs.

```
test.name = 'friedman';
test.p = 5000;
test.std = 0;
[xt, yt] = get_data(test);
tvar = test.p * std(yt)^2;
```

Next we configure the rbf_rt_1 method. The parameter values used are the results of some previous experimentation to determine values that work well on this data.

```
conf = struct('scales', [7 9], 'minmem', [3 4]);
```

The number of replications is set as follows.

```
nrep = 10;
```

Now we iterate over training set sizes (outer loop) and replications (inner loop) to get scaled errors for our first two methods.

```
results = zeros(nsiz, nrep, 2);
for i = 1:nsiz
  p = sizes(i);
  for j = 1:nrep
    [x, y] = get_data(struct('name', 'friedman', 'p', p));
    ft = sum(y) / length(y);  % Stupid method.
    results(i, j, 1) = (yt - ft)' * (yt - ft) / tvar;
    [c, r, w, info] = rbf_rt_1(x, y, conf);
    H = rbf_dm(xt, c, r, info.dmc);
    ft = H * w;
    results(i, j, 2) = (yt - ft)' * (yt - ft) / tvar;
  end
end
```

The hard work has been done by this stage and we can start to fill in some of the values in perfs.

```
for i = 1:nsiz
  perfs(i, 1, 1) = mean(results(i, :, 1));
  perfs(i, 1, 2) = std(results(i, :, 1));
  perfs(i, 2, 1) = mean(results(i, :, 2));
  perfs(i, 2, 2) = std(results(i, :, 2));
end
```

Finally, we add the results for the best MARS model on this data taken from table 9 of reference [1].

```
perfs(1, 3, 1) = 0.28;  % mean for 100 cases
perfs(1, 3, 2) = 0.17;  % std for 100 cases
perfs(2, 3, 1) = 0.12;  % mean for 200 cases
perfs(2, 3, 2) = 0.07;  % std for 200 cases
```

Calculating the probabilities in `pvals` is non-trivial (see the DELVE manual [2]) so we simply assign some values for the purposes of illustration.

```
pvals = ones(nsiz, nmet, nmet);
pvals(1, :, :) = [ 0  0  0; -1  0 -1; -1 -1 0];
pvals(2, :, :) = [ 0  0  0; -1  0 -1; -1 -1 0];
```

We'll use all the default configurations so can omit the last input argument (`conf`).

```
plot_ras(data, methods, sizes, perfs)
```

The result is in figure 2.

As a last example we replot the previous diagram (figure 2) but set some of the plot parameters to change it's appearance. The new diagram is in figure 3.

```
clear conf
conf.width = 8;
conf.mfs = 14;
conf.afs = 16;
conf.ebc = 'mrb';
conf.ebs = [2 3 2];
conf.mnc = 'mrb';
plot_ras(data, methods, sizes, perfs, [], conf)
```

Tips

If you want to use `ras_plot` in conjunction with the DELVE software (which is written in TCL) you'll have to find a way to get results from the DELVE script `mstats` into Matlab. The approach I took, for example, was to write a Perl script to call `mstats` and then strip away all but the required numbers from the rather verbose output from `mstats`. The Matlab calling function figures out the right arguments for `mstats` which it passes to the Perl script which passes back the numbers from `mstats` in the form of a string which the Matlab function can parse with `sscanf`. The perl script is available in the `perl` folder of the distribution and is called `mstats.pl`.

To print a list of configuration parameters use

```
plot_ras conf
```

To print information about one particular parameter (e.g. `afs`) use

```
plot_ras conf afs
```

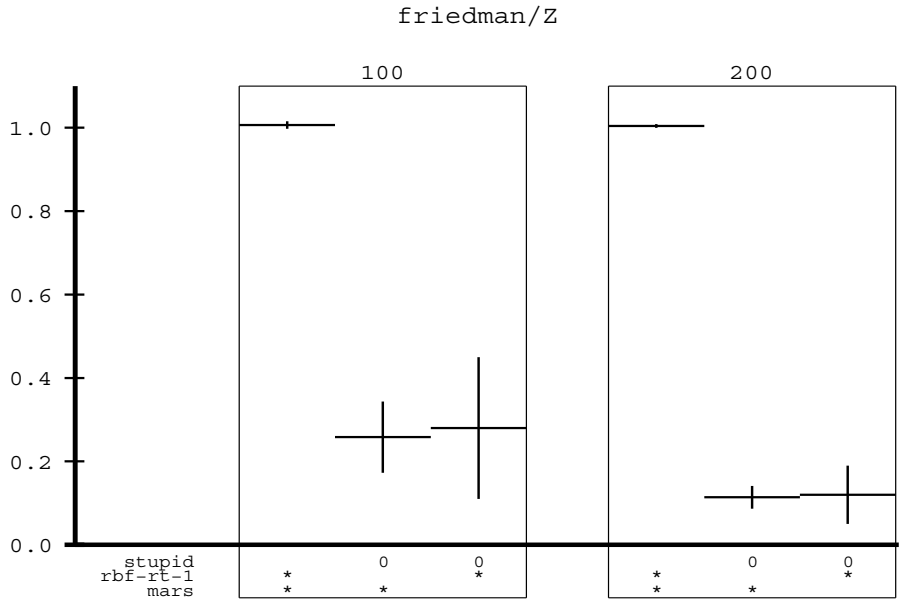

Figure 2: Rasmussen diagram for three methods on the simulated circuit data.

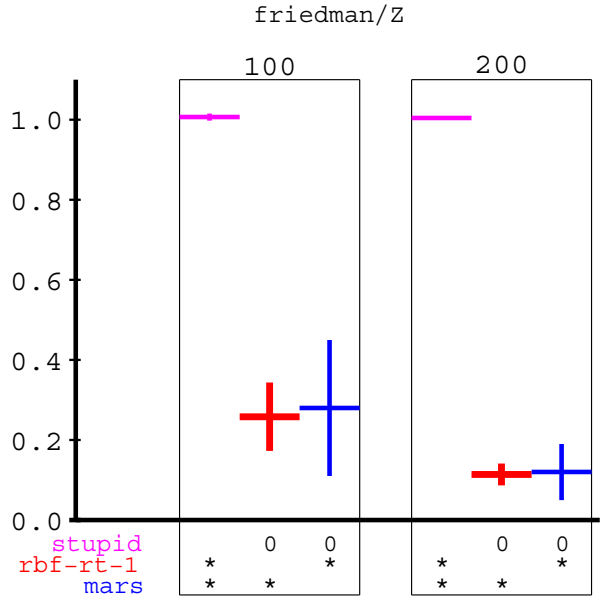

Figure 3: Same as figure 2 but with some plot parameters altered to change its appearance.

References

[1] J.H. Friedman. Multivariate adaptive regression splines (with discussion). *Annals of Statistics*, 19:1–141, 1991.

[2] C.E. Rasmussen, R.M. Neal, G.E. Hinton, D. van Camp, Z. Ghahramani, M. Revow, R. Kustra, and R. Tibshirani. *The DELVE Manual*, 1996. `http://www.cs.utoronto.ca/~delve/`.

pred_tree

Synopsis

```
y = pred_tree(tree, X);
```

Description

This function guesses the outputs `y` corresponding to input vectors in the columns of `X` on the basis of a regression tree built by the utility `grow_tree`. The pair of utilities `grow_tree` and `pred_tree` are not designed to be used as a learning method by themselves because `grow_tree` lacks a mechanism (such as tree pruning) to match the complexity of the model to the data. Rather, `grow_tree` is a utility used by other methods (`rbf_rt_1` and `rbf_rt_2`) which have separate complexity control mechanisms and `pred_tree` exists only to demonstrate what it is that `grow_tree` does (see the example for `grow_tree`).

Input Arguments

`tree` A structure representing the regression tree and generated by `grow_tree`.

`X` A matrix which specifies the inputs of the test set. If there are p cases and the input space has n dimensions then `X` should have n rows and p columns, thus the columns of `X` are the input vectors $\{\mathbf{x}_i\}_{i=1}^{p}$.

Output Argument

`y` A vector specifying the predicted outputs. Each column of the input argument `X` is paired to the corresponding component of `y`.

Configuration

There is no configuration structure for this function.

Example

Please see the example for `grow_tree`.

rbf_dm

Synopsis

```
H = rbf_dm(X, C, R, conf)
```

Description

The term *design matrix* comes from statistics, in particular from linear regression, and relates to the ability, in some experimental situations, to deliberately choose, or design, the inputs of the training set. In practical applications of neural networks the inputs are usually not under experimental control but we use the term nevertheless.

In the case of models which are linear with respect to the unknown coefficients w_j,

$$f(\mathbf{x}) \;=\; \sum_{j=1}^{m} w_j \, h_j(\mathbf{x}) \,,$$

the system of linear equations to be solved, in a least squares sense, is

$$\begin{aligned} h_1(\mathbf{x}_1)\, w_1 + h_2(\mathbf{x}_1)\, w_2 + \cdots + h_m(\mathbf{x}_1)\, w_m &= y_1 \,, \\ h_1(\mathbf{x}_2)\, w_1 + h_2(\mathbf{x}_2)\, w_2 + \cdots + h_m(\mathbf{x}_2)\, w_m &= y_2 \,, \\ \cdots &= \cdots \\ h_1(\mathbf{x}_p)\, w_1 + h_2(\mathbf{x}_p)\, w_2 + \cdots + h_m(\mathbf{x}_p)\, w_m &= y_p \,, \end{aligned}$$

and the solution, assuming no regularisation, is

$$\mathbf{w} \;=\; \left(\mathbf{H}^\top \mathbf{H}\right)^{-1} \mathbf{H}^\top \mathbf{y} \,.$$

Here, $\mathbf{H}$ is the design matrix, and its entries are the coefficients on the left hand side of the system of equations, $H_{ij} = h_j(\mathbf{x}_i)$, the responses of the regressors to the inputs in the training set. In the context of neural networks the regressor functions are identified with the transfer functions of the hidden units and the coefficients are called weights. This explains why the letters "H", "h" and "w" are used in the notation.

In the even more specialised world of RBF networks the hidden unit functions are radial, as opposed to logistic or polynomial, for example. Each hidden unit is associated with a centre $\mathbf{c}_j$ and a vector of scales (known as the radius) $\mathbf{r}_j$. Together these define a scaled distance from the j-th centre to the input

$$z_j(\mathbf{x}) \;=\; \sqrt{\sum_{k=1}^{n} \frac{(x_k - c_{jk})^2}{r_{jk}^2}} \,.$$

The hidden unit transfer function is then

$$h_j(\mathbf{x}) = \phi(z_j(\mathbf{x})) \,,$$

where ϕ is some monotonic function over positive numbers. The function `rbf_dm` implements a number of alternatives for ϕ (see `conf.type` and the *Examples* section below).

Input Arguments

`X` A matrix of n rows and p columns which specifies the training set input vectors, $\{\mathbf{x}_i\}_{i=1}^{p}$, one per column. Thus p is the number of cases and n the number of dimensions in the input space.

`C` A matrix of n rows and m columns which specifies the centres of the hidden units, $\{\mathbf{c}_j\}_{j=1}^{m}$, one per column. The input space dimensionality is n and m is the number of centres.

`R` Nominally, a matrix of n rows and m columns specifying a scale for each dimension (of which there are n) for each hidden unit (of which there are m). The columns of `R` are thus the vectors $\{\mathbf{r}_j\}_{j=1}^{m}$. However, `R` can also be a row vector, a column vector or a scalar depending on wether the radii are the same in each dimension, in each hidden unit or both. If `R` is a row vector of length m the full n-by-m matrix is obtained by stacking n copies of `R` on top of each other, so each dimension has the same scale. In this case the RBFs are all spherical, but may be of different sizes. If `R` is a column vector of length n the full matrix is obtained by stacking m copies of `R` side by side, so each hidden unit has the same radius vector `r`. If `R` is a scalar then this is copied over n rows and m columns so that the RBFs are all spherical and all the same size.

`conf` This is a structure used to configure the type of RBF and the presence of an optional bias unit. See below for details.

Output Argument

`H` The design matrix with p rows and m columns where p is the number of cases in the training set (see input argument `X`) and m is the number of hidden units (see input arguments `C` and `R`). If there is a bias unit (see `conf.bias`) `H` will have an extra column ($m + 1$ in total).

Configuration

The fields of the configuration structure `conf`, which is used to set and get optional parameters of the utility, are as follows.

`type` A string specifying the type radial function. Currently, there are four alternatives. The table below lists their names, the string values used to select them and their characteristic function, $\theta(z)$. See *Description* above for an explanation of the function θ and the distance z.

type	*string(s)*	θ
Gaussian	'g', 'gaussian'	e^{-z^2}
Cauchy	'c', 'cauchy'	$1/(1+z^2)$
multiquadric	'm', 'multiquadric'	$(1+z^2)^{1/2}$
inverse	'i', 'inverse'	$(1+z^2)^{-1/2}$

The table assumes a sharpness of 2 (see the configuration option `exp`). The default type is Gaussian.

`exp` This option is a positive scalar which can be used to sharpen or widen the radial function. The distance z in the function θ of the table above (for `type`) is their raised to the power 2. This is the default value of `exp`. Increasing it sharpens the radial function, decreasing it widens the function (see the examples).

`bias` An integer specifying the presence or absence of a bias unit and its position in the columns of `H`. Most methods support the optional inclusion of a bias unit which adds the extra term b to the model

$$f(\mathbf{x}) \;=\; b + \sum_{j=1}^{m} w_j \, h_j(\mathbf{x}) \,,$$

where m is the number of hidden units in the network and b, like any weight w_j, is one of the unknowns to be estimated. This is equivalent to an extra basis function which takes the value 1 for all $\mathbf{x}$ and an extra weight with the value b so the model can be rewritten

$$f(\mathbf{x}) \;=\; \sum_{j=1}^{m+1} w_j \, h_j(\mathbf{x}) \,.$$

The value of `conf.bias` indicates which column of `H` the bias unit is in; in other words, which value of j indexes $h_j = 1$ and $w_j = b$. If `bias` is zero, which is the default, there is no bias unit and the design matrix `H` has just m columns, not $m + 1$.

Examples

The manual pages for the various methods are full of examples of `rbf_dm` being used for prediction. Here we first illustrate the various types of RBF available. For this purpose we first create an array of ordered and equally spaced x-values which we can use for plotting.

```
x = linspace(-4, 4, 1000);
```

Next, we sample 4 different types of RBF on this set of inputs. All have their centre at zero and are 1 unit wide.

```
m = rbf_dm(x, 0, 1, struct('type', 'm'));
i = rbf_dm(x, 0, 1, struct('type', 'i'));
c = rbf_dm(x, 0, 1, struct('type', 'c'));
g = rbf_dm(x, 0, 1, struct('type', 'g'));
```

The functions are plotted in figure 1.

```
hold off
plot(x, m, 'm-')
hold on
plot(x, i, 'r-')
plot(x, c, 'c-')
plot(x, g, 'g-')
```

The order of names in the legend matches the height of the functions in the plot. The multiquadric is first in the list and higher than all the others. The Gaussian is last in order and lowest in height. All are monotonic functions of the distance from the centre but only the multiquadric increases, the others decrease.

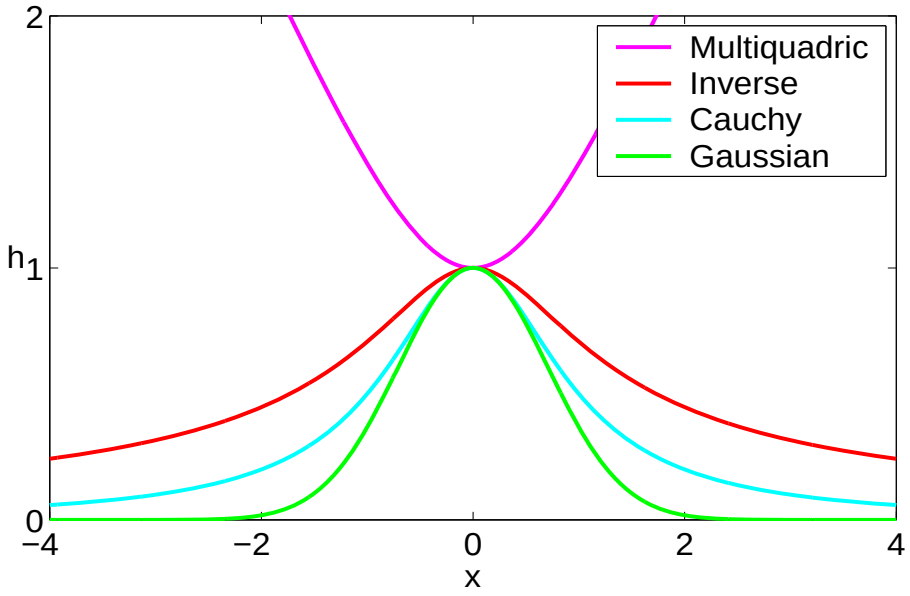

Figure 1: Different types of RBFs.

To show the effect of changing the sharpness we vary the parameter `exp`.

```
g1 = rbf_dm(x, 0, 1, struct('exp', 1));
g2 = rbf_dm(x, 0, 1, struct('exp', 2));
g3 = rbf_dm(x, 0, 1, struct('exp', 3));
g4 = rbf_dm(x, 0, 1, struct('exp', 4));
```

The functions are shown in figure 2.

```
hold off
plot(x, g4, 'b-')
hold on
plot(x, g3, 'c-')
plot(x, g2, 'g-')
plot(x, g1, 'y-')
```

The order in the legend is from narrowest (`exp = 4`) to widest (`exp = 1`). Note that for values of `exp` less than 2 all RBF types have discontinuous derivatives at their centre.

For the final example we will plot a small function model with 3 RBFs in a 2D input space. For this we need a grid of x-values for which we can use the utility `get_data` and its `'sine2'` data set.

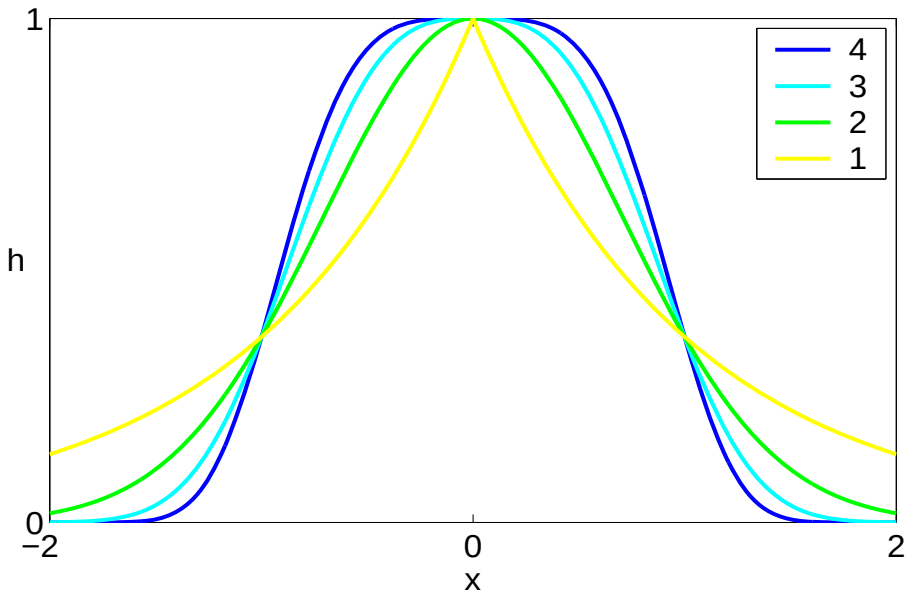

Figure 2: Different sharpness for the Gaussian RBF.

```
conf.name = 'sine2';
conf.p = 1600;
conf.ord = 1;
[X, y, conf] = get_data(conf);
```

The input ranges for the `'sine2'` data set are $0 \leq x_1 \leq 10$ and $-5 \leq x_2 \leq 5$. The three RBF centres are placed at $\mathbf{c}_1 = (5,\, 2)$, $\mathbf{c}_2 = (7,\, 0)$ and $\mathbf{c}_3 = (2,\, -3)$.

```
C = [5 7 2; 2 0 -3];
```

The first RBF has a unit radius in component 2 but is twice as wide in component 1. The second RBF has a unit radius in the first dimension but is three times that size in the second. The last RBF has a scale of 1 in each direction.

```
R = [2 1 1; 1 3 1];
```

Now that we have the inputs `X`, centres `C` and radii `R` we can get the design matrix. We use all the default configurations (Gaussian type, sharpness 2, no bias unit).

```
H = rbf_dm(X, C, R);
```

Lastly, we set the weights of each RBF:

```
w = [2; 1; 0.5];
```

Now we can calculate the function over the grid of x-values.

```
f = H * w;
```

Before we plot it we also have to grid `f` and get values for the grid coordinates.

```
q = sqrt(conf.p);
F = zeros(q, q);
F(:) = f;
v1 = linspace(conf.x1(1), conf.x2(1), q);
v2 = linspace(conf.x1(2), conf.x2(2), q);
```

Finally we can plot the function (see figure 3). The first and second RBFs merge together in the background while the third, with its smaller weight, sits apart in the foreground.

```
hold off
mesh(v1, v2, F);
```

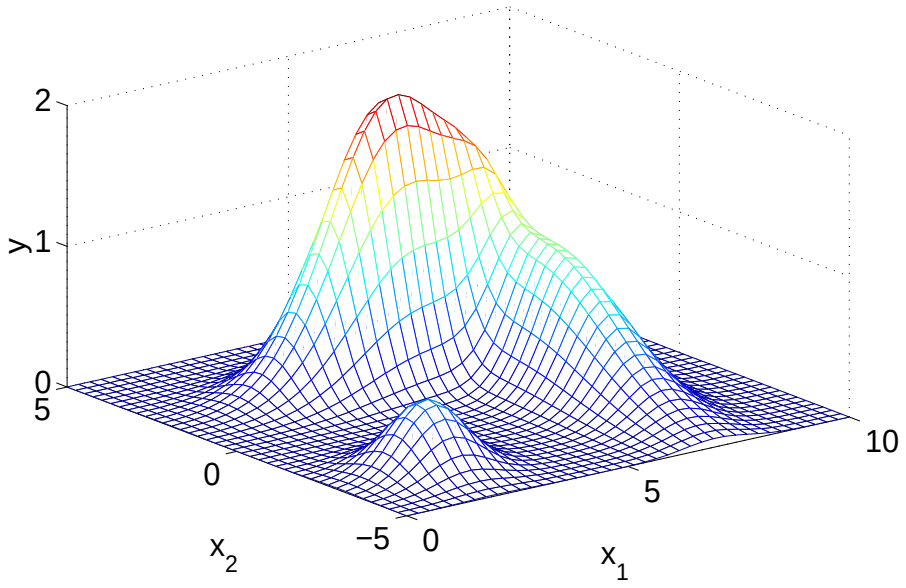

Figure 3: The 2D function modelled by a small RBF network.

Tips

To print a list of configuration parameters use

```
rbf_dm conf
```

To print information about a particular parameter (e.g. `type`) use

```
rbf_dm conf type
```

rbf_ver

Synopsis

```
rbf_ver
```

Description

This script prints the version number of the software package and the URL where you can download the latest version.