Part II

Practical Toolbox

About the Regression Toolbox

One reason why data-oriented methods have become so popular during the last years is the availability of high-capacity computers and efficient software tools. There is a wealth of alternative tools available — different kinds of tools for different needs.

There exist many ways to categorize the software tools. One characterization goes along specialization: For example, general-purpose programming languages like `C++` and `Java` make it possible to implement any algorithm — if there is enough time available. Specialized program products (like `SIMCA`, etc.) make it easy to do the standard operations on data appropriately; however, one is bound to the ready-to-use routines. `Matlab` is there in between, offering a programming framework for implementing generic algorithms that can directly operate on high-level concepts from linear algebra. Based on this `Matlab` platform, various more or less sophisticated *toolboxes* have been implemented that are tailored for special purposes: For example, the following toolboxes are available, either in public domain or commercially:

- `Statistics Toolbox` (indeed, various versions exist) for statistical data analysis
- `PLS Toolbox` for PCA/PLS modeling, etc.
- `Chemometrics Toolbox` for model calibration, etc.
- `System Identification Toolbox` for analysis of dynamic systems.

Additionally, there exist dozens of toolboxes that concentrate on some specific approach, like `Neural Networks Toolbox`, `Optimization Toolbox`, `FastICA Toolbox`, etc. It seems that as these toolboxes have been deloped by professional domain-area experts, they often are rather unpenetrable: They are difficult to understand, meaning that they are difficult to modify or experiment with.

That is why, there is need for yet another toolbox. The implemented `Regression Toolbox` for `Matlab` specially supports the theoretical derivations discussed in Part 1. The routines are not polished or optimized. But because the used codes are so simple, they can be understood also by a non-expert, and they can be easily experimented with, and extended if needed.

Just as in the theoretical discussions before, the main goal in the Toolbox is to present all methods in a homogeneous framework, and to show how simple all the algorithms (in principle) are. The routines in this Toolbox are not intended for professional use.

Installation

The `Regression Toolbox` version 1.1 can be downloaded through the address `http://saato014.hut.fi/hyotyniemi/publications/01_report125.htm`.

There are two compressed files, one for the commands, and one for the accompanying data for experimenting. The compressed files have to be uncompressed using `WinZip`, for example, and the files should be expanded to a separate toolbox folder.

Within the `Matlab` environment, the search path has to be modified so that the toolbox folder is included (preferably in the beginning of the search path). After that, the commands should be operational. The toolbox was developed in the `Matlab` version `5.3` environment, and the compatibility with other versions is *not* guaranteed; however, only the very basic functionality of `Matlab` is used.

Standard `Matlab` style command line `help` scripts are supplied for all routines. No special user interface is available; the commands are run from command line (because of this, the Toolbox *may* be operational also in other versions of `Matlab`).

Commands at a glance

The `Regression Toolbox` consists of the following commands (summarized here in not in alphabetical but in "logical" order). Each of the commands is explained in more detail in the attached Reference Guide. The `help` command of `Matlab` is available for on-line use.

Preprocessing commands

- `regrCenter`: Mean centering of data
- `regrScale`: Normalization, variable variances getting scaled
- `regrWeight`: Weighting of data samples
- `regrWhiten`: "Whitening" of data: covariance becomes identity matrix
- `regrFixval`: Iterative fixing of missing data values.

Cluster management

- `regrFDA`: Fisher Discriminant Analysis for distinguishing between clusters
- `regrForm`: Histogram equalization (or deformation) model construction
- `regrDeform`: Histogram equalization model application
- `regrEM`: Expectation Maximization clustering
- `regrKM`: K-Means clustering of data
- `regrOutl`: Visual outlier detection.

Structure refinement

- `regrPCA`: Standard Principal Component Analysis
- `regrPLS`: Partial Least Squares analysis, formulated as an eigenproblem
- `regrCR`: Continuum regression basis determination
- `regrCCA`: Canonical Correlation Analysis
- `regrICA`: Independent Component Analysis
- `regrRBFN`: Radial Basis Function Network construction.

Model construction and regression

- `regrMLR`: Multi-Linear Regression
- `regrMLRC`: Multi-Linear Regression with linear constraints
- `regrOLS`: Orthogonal Least Squares algorithm
- `regrTLS`: Total Least Squares regression
- `regrRR`: Ridge Regression
- `regrRBFR`: Radial Basis Function Regression.

Functions for dynamic systems

- `regrBal`: Balancing and reducing a dynamic state-space system
- `regrCyb`: Iterative "cybernetic regression"
- `regrIdent`: Black-box identification of ARX models
- `regrSSI`: SubSpace Identification of dynamic systems
- `regrSSSI`: Stochastic SubSpace Identification of dynamic systems.

Iterative demonstration algorithms

- `regrCYB`: "Cybernetic" adaptation of PCA
- `regrFACTOR`: Factor analysis applying the neocybernetic approach
- `regrHAH`: Hebbian - Anti-Hebbian regression
- `regrPPCA`: PCA using the power method
- `regrGHA`: PCA using the Generalized Hebbian Algorithm
- `regrIICA`: "Interactive" ICA.

Analysis and visualization

- `regrP`: Fit data against a Gaussian distribution
- `regrCrossval`: Cross-validation of the model
- `regrShowClust`: Visualize the structure of clustered data
- `regrKalman`: Implement discrete-time Kalman filter
- `regrKalm`: Implement stochastic discrete-time Kalman filter
- `regrAskOrder`: Visual tool for model order determination.

Test material

- `dataXY`: Generate random input-output data
- `dataClust`: Generate random clustered data
- `dataIndep`: Generate data consisting of independent signals
- `dataDigits`: Handwritten digits (Warning: Large file)
- `dataDyn`: Generate random dynamic data
- `dataHeatExch`: Heat exchanger data
- `dataEmotion`: Voice signal data (Warning: Large file).

dataClust

Function generates random clustered data.

Syntax

```
[X] = dataclust(n,N,kk,cm,cd)
[X] = dataclust(n,N,kk,cm)
[X] = dataclust(n,N,kk)
[X] = dataclust(n,N)
[X] = dataclust(n)
[X] = dataclust
```

Input parameters

- `n`: Data dimension (default 3)
- `N`: Number of clusters (default 2)
- `kk`: Data samples in each cluster (default 100)
- `cm`: Deviation of the cluster centers (default 1)
- `cd`: Cluster spread, “longest” axis vs. “shortest” (default 1)

Return parameter

- `X`: Data matrix (size $kk \cdot N \times n$)

Comments

The cluster centers are normally distributed around origin, the centers having standard deviation `cm`.

The individual clusters have internal normal distributions determined by parameter `cd`: If this ratio between the distribution principal axes is 1, the clusters are circular. Otherwise, the standard deviations in randomly selected orthogonal directions are determined so that the deviation widths are equally spaced on the logarithmic scale, the ratio between widest and narrowest deviation being `cd`. The determinant of the covariance matrix is always 1.

dataDigits

Challenging data: Handwritten digits (thanks to Jorma Laaksonen, Dr.Tech.)

Syntax

```
datadigits
```

Comments

Running the command defines the 500×256 matrix `DIGITS` containing 500 samples of handwritten digits in a 16×16 grid. Each row represents one sample, packed in a vector row by row; this means that the data can be visualized in the following way:

```
digit = DIGITS(index,:);
feature = zeros(16,16);
for j = 1:16
   feature(j,:) = digit((j-1)*16+1:j*16);
end
colormap(gray);
imagesc(feature);
```

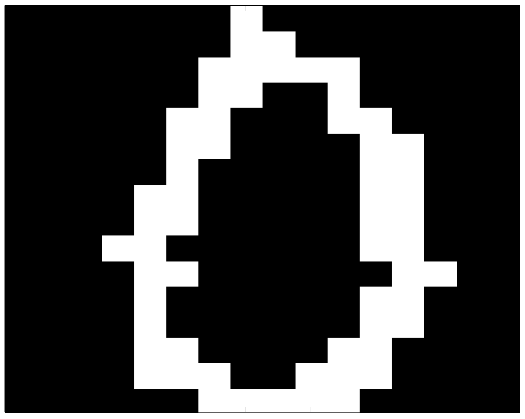

Figure B.4: Index 1: Example of the digit “0”

dataDyn

Generate random data coming from a state-space system.

Syntax

```
[U,Y] = datadyn(n,nu,m,k,sigma)
[U,Y] = datadyn(n,nu,m,k)
[U,Y] = datadyn(n,nu,m)
[U,Y] = datadyn(n,nu)
[U,Y] = datadyn(n)
[U,Y] = datadyn
```

Input parameters

- `n`: State dimension (default 1)
- `nu`: Input dimension (default 1); see below
- `m`: Output dimension (default 1)
- `k`: Number of data samples (default 100)
- `sigma`: Standard deviation of the noise (default 0)

Return parameters

- `U`: Input sequence (size $k \times \nu$)
- `Y`: Output sequence (size $k \times m$)

Comments

Function generates sequences of random dynamical data, starting from zero initial condition. Parameter `sigma` determines all the noise processes: The standard deviation of the state noise and the measurement noise.

If the input parameter `nu` is zero, the system has no inputs, being driven by a stochastic process; if `nu` is vector or matrix, this data is directly interpreted as the input data.

dataEmotion

Command file defines sound signal samples.

Syntax

```
dataemotion
```

Comments

There are no expicit inputs or outputs in this command file; running it constructs matrix `DATA` in the workspace. `DATA` contains five sequences of sound signals from different sources; these sources are presented in `DATA` as separate columns.

```
dataemotion;
sound(DATA(:,1),16000);
sound(DATA(:,2),16000);
sound(DATA(:,3),16000);
sound(DATA(:,4),16000);
sound(DATA(:,5),16000);
```

Because no compression of the signals has been carried out, this file is rather large.

dataHeatExch

Command file defines heat exchanger data.

Syntax

```
dataheatexch
```

Comments

There are no expicit inputs or outputs in this command file; running it constructs two matrices `X` and `Y` in the workspace, where the input data `X` stands for temperature measurements along the heat exchanger (see Fig. B.5), and the matrix `Y` is interpreted as follows:

- Y_1: Temperature of the incoming cold flow
- Y_2: Temperature of the incoming hot flow
- Y_3: Temperature of the outgoing cold flow
- Y_4: Temperature of the outgoing hot flow.

This data is used, for example, as the training material for the *Regreswsion Course,* and different regression methods can be experimented with and their properties can be compared: A model should be constructed for estimating y when x is given. Note that the causality structure is here blurred — the values in y cannot be interpreted as being functions of x, but prediction models can still be implemented.

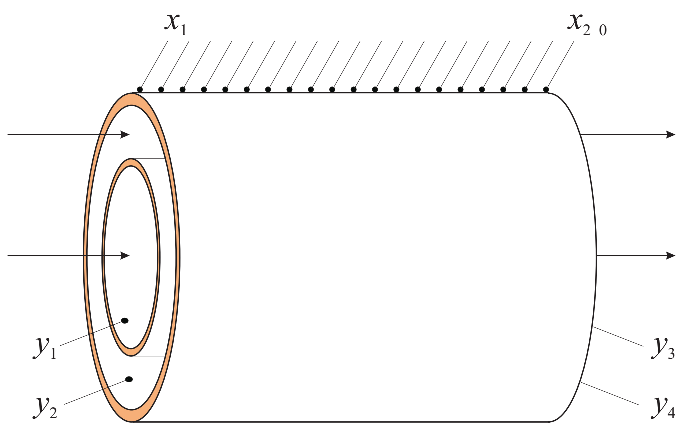

Figure B.5: “Instrumentation” of the heat exchanger

dataIndep

Function mixes independent data sources.

Syntax

```
[X] = dataindep(k,func1,func2,func3,func4,func5,func6)
[X] = dataindep(k,func1,func2,func3,func4,func5)
[X] = dataindep(k,func1,func2,func3,func4)
[X] = dataindep(k,func1,func2,func3)
[X] = dataindep(k,func1,func2)
[X] = dataindep(k,func1)
[X] = dataindep(k)
[X] = dataindep
```

Input parameters

- `k`: Number of data samples (default 1000)
- `func`*i*: If `func`*i* is string, it is evaluated as a function of time index k (note that in the text this variable is called κ). There are some functions directly available:
  - `'f1'`: Harmonic, $x_i(k) = \sin(k/5)$
  - `'f2'`: Saw-tooth, $x_i(k) = (\mathrm{rem}(k, 27) - 13)/9$
  - `'f3'`: "Strange curve", $x_i(k) = ((\mathrm{rem}(k, 23) - 11)/9)^5$
  - `'f4'`: Impulsive noise, $x_i(k) = \mathrm{binrand}(k) \cdot \log(\mathrm{unifrand}(k))$, where "binrand" and "unifrand" denote random binary and uniform sequences, giving two alternative values -1 or 1 and continuum of values between 0 and 1, respectively

  Default is `func1='f1'`, `func2='f2'`, `func3='f3'`, and `func4='f4'`.

Return parameter

- `X`: Data matrix (size $k \times n$, where n is the number of selected functions)

Comments

Linear mixing, mean centering, and whitening is applied to the set of signals automatrically.

dataXY

Function generates random input-output data.

Syntax

```
[X,Y] = dataxy(k,n,m,dofx,dofy,sn,sm)
[X,Y] = dataxy(k,n,m,dofx,dofy)
[X,Y] = dataxy(k,n,m)
[X,Y] = dataxy(k)
[X,Y] = dataxy
```

Input parameters

- `k`: Number of samples (default 100)
- `n`: Input data dimension (default 5)
- `m`: Output data dimension (default 4)
- `dofx`: Non-redundant input data dimension (default 3)
- `dofy`: Non-redundant output data dimension (default 2)
- `sn`: Input noise level (default 0.001)
- `sm`: Output noise level (default 0.1)

Return parameters

- `X`: Input data matrix (size $k \times n$)
- `Y`: Output data matrix (size $k \times m$)

Comments

This data is specially intended for visualizing the differences between MLR, PCR, and PLS regression methods. There is redundancy in X, making problems of MLR visible; but not all of the input variation explains the output, so that the difference between PCR and PLS is also demonstrated.

regrAskOrder

Interactively determine the model order.

Syntax

```
[N] = regraskorder(LAMBDA)
```

Input parameter

- LAMBDA: Vector of latent vector weights

Return parameter

- N: Selected model order

Comments

Function plots the values in LAMBDA and lets the user select how many of the latent variables will be used in the model construction.

This function is intended to be used only by other routines (`regrPCA`, `regrCCA`, `regrPLS`, `regrICA`, `regrTLS`, `regrBal`, `regrSSI`, and `regrSSSI`) if the model order is not explicitly determined.

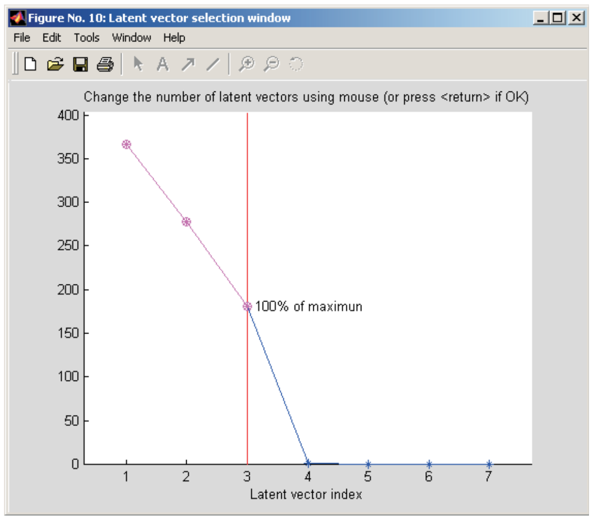
Figure No. 10: Latent vector selection window
File Edit Tools Window Help
Change the number of latent vectors using mouse (or press <return> if OK)
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
Latent vector index
100% of maximun

Figure B.6: Selection of the model order. Note that the figure "X% of maximum" only means how much of the absolute *achievable* maximum is captured

regrBal

State-space dynamic system balancing and reduction.

Syntax

```
[Ared,Bred,Cred,theta,sigma] = regrbal(A,B,C,N)
[Ared,Bred,Cred,theta,sigma] = regrbal(A,B,C)
```

Input parameters

- `A`, `B`, `C`: System matrices
- `N`: Number of remaining states (optional)

Return parameters

- `Ared`, `Bred`, `Cred`: Reduced system matrices
- `theta`: State transformation matrix, `z=theta'*x`
- `sigma`: Hankel singular values

Comments

If the reduced model order is not given, graphical interaction with the user is used (see `regrAskorder`).

If none of the states is dropped, the model is just balanced.

regrCCA

Canonical Correlation Analysis (CCA) model construction.

Syntax

```
[theta,phi,lambda] = regrcca(X,Y,N)
[theta,phi,lambda] = regrcca(X,Y)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `N`: Number of latent variables (optional)

Return parameters

- `theta`: Input block canonical variates
- `phi`: Output block canonical variates
- `lambda`: Canonical correlation coefficients

Comments

If the number of latent vectors N is not explicitly given, it is queried interactively (see `regrAskOrder`).

Based on this command, the Canonical Correlation Regression can easiest be implemented as

```
F = regrmlr(X,Y,regrcca(X,Y));
Ytest = Xtest*F;
```

regrCenter

Function for mean centering the data.

Syntax

```
[X,barX] = regrcenter(DATA,barX)
[X,barX] = regrcenter(DATA)
```

Input parameter

- `DATA`: Data to be modeled
- `barX`: Point in space included in the model (optional)

Return parameters

- `X`: Transformed data matrix
- `barX`: Center of DATA

Comments

Returning to the original coordinates can be carried out as

```
Data = X + barX;
```

regrCR

Continuum Regression basis determination.

Syntax

```
[theta,lambda] = regrcr(X,Y,alpha,N)
[theta,lambda] = regrcr(X,Y,alpha)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `alpha`: Continuum parameter from ≈ 0 (MLR) to 1 (PCR) through 0.5 (PLS)
- `N`: Dimension of the latent structure (optional)

Return parameters

- `theta`: Latent basis vectors
- `lambda`: Corresponding eigenvalues

Comments

The determination of the CR latent basis is *not* carried out exactly as explained on page 98; the reason is that the powers of an $k \times k$ matrix should be calculated, resulting in an huge eigenvalue problem; on the contrary, a shortcut using singular value decomposition is applied.

If the number of latent vectors N is not explicitly given, it is queried interactively (see `regrAskOrder`).

Based on this command, the actual Continuum Regression can easiest be implemented as

```
F = regrmlr(X,Y,regrcr(X,Y,alpha));
Ytest = Xtest*F;
```

regrCrossVal

Function for cross-validation of linear regression models.

Syntax

```
[E] = regrcrossval(X,Y,expr,seqs)
[E] = regrcrossval(X,Y,expr)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `expr`: String form expression resulting in F, given X and Y
- `seqs`: How many cross-validation rounds (default k, meaning “leave-one-out” cross-validation approach)

Return parameter

- `E`: Validation error matrix (size $k \times m$)

Comments

Cross-validation leaves a set of samples out from the training set, constructs the model using the remaining samples, and tries to estimate the left-out samples using the model.

Parameter `seqs` determines how many continuous separate validation sets are used; default is k, meaning that the model construction is carried out k times, always with $k-1$ samples in the training set.

In this routine, it is assumed that the string `expr`, when evaluated, returns the mapping matrix F, so that $Y = X \cdot F$. References within the string to input and output blocks must be `X` and `Y`, respectively, no matter what are the actual symbols.

Remember that validation of the constructed model with fresh samples is crucial — otherwise MLR would always give the seemingly best model, explicitly minimizing the cost criterion for the training data, even though its robustness is weak.

regrCYB

Function that implements adaptation of data structures in the "neocybernetic model".

Syntax

```
[A,B,Uhat,Xbar] = regrCYB(U,A,B,lambda,S,nonlin,maskA,maskB)
[A,B] = regrCYB(U,A,B)
```

Input parameters

- `U`: Input data block (size $k \times \nu$)
- `A`: Feedback matrix A ($n \times n$)
- `B`: Feedforward matrix B ($n \times \nu$)
- `lambda`: Forgetting factor (scalar)
- `S`: Sparsity - how many variables used
- `nonlin`: If non-zero, nonlinear cut modeling
- `maskA`: Masking matrix for A
- `maskB`: Masking matrix for B

Return parameters

- `A`: Modified feedback matrix A
- `B`: Modified feedforward matrix B
- `Uhat`: Estimate of the input
- `Xbar`: State variable sequence in balance

Comments

This routine implements the "neocybernetic model" in a simplified form (for more information, see `http://www.control.hut.fi/cybernetics`). In the linear and non-sparse-coded form principal subspace analysis is carried out; if the masking matrix `maskA` in adaptation of `A` is triangular, principal component analysis results (accepting the default, the last feature will represent the most significant principal component direction). Principal subspace analysis is now iterative, no explicit data covariance matrix is constructed; in this sense, this method can even be useful for analysis of very high-dimensional data, for example, when doing image analysis:

```
dataDigits;          % Loading challenging data
n = 10;              % Dimension of latent basis
A = 0.01*eye(n);
B = 0.01*randn(n,size(DIGITS,2));
```

```
                    % Iteration repeated until convergence
[A,B] = regrCYB(DIGITS,A,B,0.8);
[A,B] = regrCYB(DIGITS,A,B,0.8);
...
```

After this, the pattern vectors are stored as columns in the data structure `(inv(A)*B)'`. In the algorithm, the internal dynamics of a neocybernetic model is abstracted away. What is more, the structure is streamline somewhat: Internal iterations are eliminated, making the `Matlab` implementation relatively fast. However, these modifications can result in problems if nonlinearity is employed. Specially, when employing the *cut* nonlinearity (`nonlin = 1`), it can be beneficial to "invert" some of the features; assume that feature number i is stuck in negative weights, so that there never holds $x_i > 0$, one can make it active by writing

```
invert = zeros(n,1); invert(i) = 1;
B = B - 2*(invert*ones(1,size(B,2))).*B;
A = A - 2*(invert*ones(1,size(A,2))).*A;
A = A - 2*(ones(size(A,1),1)*invert').*A;
```

regrDeForm

Function for equalizing data distributions.

Syntax

```
[X,W] = regrdeform(DATA,defmatrix)
```

Input parameters

- `DATA`: Data to be manipulated (size $k \times n$)
- `defmatrix`: Matrix containing deformation information (see `form`)

Return parameters

- `X`: Data with (approximately) deformed distribution (size $k \times n$)
- `W`: Validity vector containing “1” if measurement within assumed distribution, “0” otherwise

Comments

Note that the histogram of the deformed data follows the intended distribution only with the resolution that was determined by the number of *bins* in the equalization model construction function `form`. That is, if only 10 bins, for example, are used, histograms plotted with higher resolution will still be unevenly distributed (even for the training data).

regrEM

Function for clustering using Expectation Maximization algorithm.

Syntax

```
[clusters] = regrem(X,N,centers,equal)
[clusters] = regrem(X,N,centers)
[clusters] = regrem(X,N)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `N`: Number of clusters
- `centers`: Initial cluster center points (by default determined by K-means, see `km`)
- `equal`: “1” if distributions within clusters are assumed equal (default “0”)

Return parameter

- `clusters`: Vector (size $k \times 1$) showing the clusters (1 to N) for samples

Comments

Sometimes, for difficult combinations of clusters, the procedure may get stuck in the clustering given by K-means algorithm; that is why, the initial centers can be given also explicitly. Also, setting `equal` to “1” makes the algorithm more robust; this can be utilized if there is only affinity difference between clusters.

Only the set membership information is given out; the actual properties for cluster 1, for example, can be calculated as

```
cl = regrem(X,N)
count1 = sum(find(cl==1));
center1 = mean(X(find(cl==1),:))';
covariance1 = X(find(cl==1),:)'*X(find(cl==1),:)/count1 ...
             - center1*center1';
```

regrFACTOR

Function that implements adaptation of data structures in the "neocybernetic model".

Syntax

```
  [Exu,Uhat,Q] = regrFACTOR(U,Exu,Q,lambda,nonlin,XXref)
  [Exu,Uhat,Q] = regrFACTOR(U,Exu,Q)
```

Input parameters

- `U`: Input data block (size $k \times m$)
- `Exu`: Model matrix (format $n \times m$)
- `Q`: "Stiffnesses" (diagonal matrix $n \times n$)
- `lambda`: Forgetting factor (scalar, default no learning)
- `nonlin`: Nonlinearity applied (default "1"=true)
- `XXref`: Scalar/vector of x_i variances (default levels at 1)

Return parameters

- `Exu`: Modified model matrix
- `Uhat`: Balance data estimate outside the system
- `Q`: Related to error covariances (diagonal matrix $n \times n$)

Comments

This routine implements the "neocybernetic model" in a simplified form (for more information, see `http://www.control.hut.fi/cybernetics`). Principal subspace analysis is carried out, and within that subspace, *sparse coding* is searched for. The code below shows this coding for the digit data:

```
% Load hand-written digits
dataDigit; U = DIGITS;

% Parameters
[k,m] = size(U);
n = 16; lambda = 0.9;

% Initialize model structures
Exu = 0.01*randn(n,m);
Q = 1*eye(n);

while 1 == 1
  [Exu,Uhat,Q] = regrFACTOR(U,Exu,Q,lambda,1);
  % Visualization
  for i = 1:n
    feature = zeros(16,16);
    for j = 1:16, feature(j,:) = Exu(i,(j-1)*16+1:j*16); end
    subplot(sqrt(n),sqrt(n),i); imagesc(feature); colormap('hot'); drawnow;
  end
end
```

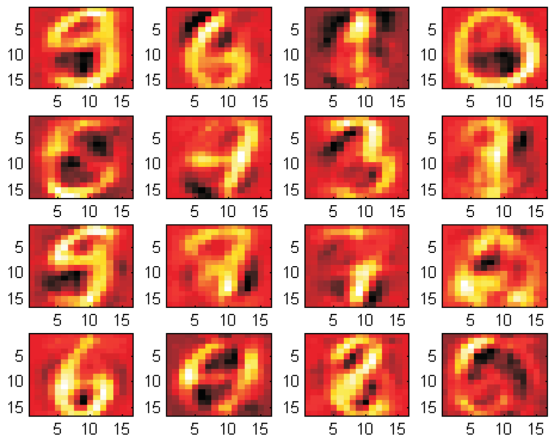

Figure B.7: 16 factors extracted from the handwritten digits

regrFDA

Fisher Discriminant Analysis (FDA) for discriminating between classes.

Syntax

```
[theta,lambda] = regrfda(X,clusters,N)
[theta,lambda] = regrfda(X,clusters)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `clusters`: Cluster index between 1 and N for all k samples (size $k \times 1$)
- `N`: Number of discriminant axes (optional)

Return parameters

- `theta`: Discriminant axes (size $n \times N$)
- `lambda`: Corresponding eigenvalues

Comments

If the number of discriminant axes N is not explicitly given, it is queried interactively (see `regrAskOrder`).

The vector `clusters` can be constructed, for example, by the EM algorithm (see `em`):

```
X = dataclust(3,2,50,5,5);
clusters = regrem(X,2);
theta = (X,clusters,1);
```

`regrFixVal`

Function tries to fix missing values matching data against model.

Syntax

```
[Xhat,Yhat] = regrfixval(X,Y,F,Wx,Wy)
```

Input parameters

- `X`: Data matrix to be fixed (size $k \times n$)
- `Y`: Output matrix to be fixed (size $k \times m$)
- `F`: Model matrix, assuming that `Y = X*F`
- `Wx`: Matrix containing "1" for each valid data in X (size $k \times n$)
- `Wy`: Matrix containing "1" for each valid data in Y (size $k \times m$)

Return parameters

- `Xhat`: Fixed input data matrix (size $k \times n$)
- `Yhat`: Fixed output matrix (size $k \times m$)

Comments

Function tries to fix uncertain elements in X and Y (as pointed out by zeros in the matrices `Wx` and `Wy`, respectively) to have more plaussible values, so that the reconstruction error $E = Y - XF$ would be minimized. This procedure can be repeated as meny times as needed:

```
for i = 1:10
   F = mlr(Xhat,Yhat);
   [Xhat,Yhat] = regrfixval(Xhat,Yhat,F,Wx,Wy);
end
```

regrForm

Function for “equalizing”, finding mapping between distributions.

Syntax

```
[defmatrix] = regrform(DATA,normdist)
[defmatrix] = regrform(DATA,bins)
[defmatrix] = regrform(DATA)
```

Input parameters

- `DATA`: Data to be manipulated (size $k \times n$)
- `normdist` for *vector argument*: Intended distribution form
- `bins` for *scalar argument*: Number of “bins”, data sections
  (default distribution being Gaussian between -3σ to 3σ, number of bins being 10 if no argument is given)

Return parameters

- `defmatrix`: Matrix containing deformation information. For each variable (column) of `DATA`, there are the starting points of the data bins, and between these there is the “steepness” of the distribution within the bin; there are this kind of bin/steepness pairs as many as there are bins (and, additionally, the end point of the last bin), altogether the matrix size being $2 \cdot \texttt{bins} + 1 \times n$.

Comments

Too special distributions, or if there are too many bins as compared to the number of data, may result in difficulties in the analysis. These failure situations are stochastic, being caused by a random process dividing samples in bins: This means that retrial may help.

regrGHA

Principal Component Analysis using Generalized Hebbian Algorithm.

Syntax

```
[theta,lambda] = regrgha(X,N,gamma,epochs)
[theta,lambda] = regrgha(X,N,gamma)
[theta,lambda] = regrgha(X,N)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `N`: Number of latent variables to be extracted
- `gamma`: Step size (default $\gamma = 0.001$)
- `epochs`: Number of iterations (default 10)

Return parameters

- `theta`: Extracted eigenvectors (size $n \times N$)
- `lambda`: Corresponding eigenvalues

Comments

As compared to `regrPCA` or even `regrPPCA`, convergence here is slow; however, no explicit covariance matrix needs to be constructed.

regrICA

Eigenproblem-form Independent Component Analysis (ICA).

Syntax

```
[theta,lambda] = regrica(X,alpha,N)
[theta,lambda] = regrica(X,alpha)
[theta,lambda] = regrica(X)
```

Input parameters

- `X`: Input (whitened) data block, mixture of signals (size $m \times n$)
- `alpha`: Type of signals being searched (default 1)
- `N`: Number of latent vectors (default all)

Return parameters

- `theta`: Independent basis vectors
- `lambda`: Corresponding eigenvalues

Comments

This function calculates independent components using an eigenproblem-oriented approach called FOBI. The (prewhitened) data is modified so that the anomalies in the distribution become visible in the second-order properties; this is accomplished in the following way:

$$x'(\kappa) = \|x(\kappa)\|^{\alpha} \cdot x(\kappa)$$

It can be shown that for $\alpha = 1$ the standard, kurtosis-based independent component analysis results. Different values of α may emphasize different statistical moments.

For $\alpha = 1$, the kurtosis in direction θ_i can be calculated as $\lambda_i - n - 2$.

regrIdent

Identify SISO system recursively.

Syntax

```
[f] = regrident(u,y,n,lambda,f)
[f] = regrident(u,y,n,lambda)
[f] = regrident(u,y,n)
```

Input parameters

- `u`: Scalar input sequence (size $k \times 1$)
- `y`: Scalar output sequence (size $k \times 1$)
- `n`: System dimension
- `lambda`: Forgetting factor (default 1, no forgetting)
- `f`: Initial parameter vector (default zero)

Return parameters

- `f`: Final parameter vector

Comments

Very simple SISO identification algorithm based on the ARX model structure. The structure of the model is

$$y(\kappa) = a_1 y(\kappa-1) + \cdots + a_n y(\kappa-n) + b_1 u(\kappa-1) + \cdots + b_n u(\kappa-n),$$

and the parameter vector is

$$f = \begin{pmatrix} a_1 \\ \vdots \\ a_n \\ b_1 \\ \vdots \\ b_n \end{pmatrix}.$$

`regrIICA`

“Interactive” Independent Component Analysis (ICA).

Syntax

```
[theta] = regriica(X)
```

Input parameter

- `X`: Input (whitened) data block, mixture of signals (size $k \times n$)

Return parameters

- `theta`: Independent basis vectors

Comments

The method presented in Sec. 7.3.3 results, in principle, in $n \cdot (n^2 + n)/2$ provisional independent components. This function lets the user interactively select those that are most interesting.

regrKalm

Discrete time stochastic Kalman filter.

Syntax

```
[Xhat,Yhat] = regrkalm(Y,A,C,Rxx,Ryy,Rxy,x0)
[Xhat,Yhat] = regrkalm(Y,A,C,Rxx,Ryy,Rxy)
```

Input parameters

- `Y`: Output data block (size $k \times m$)
- `A`, `C`, `Rxx`, `Ryy`, and `Rxy`: System matrices
- `x0`: Initial state (default $\mathbf{0}$)

Return parameters

- `Xhat`: State sequence estimate
- `Yhat`: Corresponding output estimate

Comments

The assumed stochastic system structure is

$$\begin{cases} x(\kappa+1) = Ax(\kappa) + \epsilon(\kappa) \\ y(\kappa) = Cx(\kappa) + e(\kappa), \end{cases}$$

where the white noise sequences $\epsilon(\kappa)$ and $e(\kappa)$ are characterized by the covariance matrices $\mathrm{E}\{\epsilon(\kappa)\epsilon^T(\kappa)\} = R_{xx}$, $\mathrm{E}\{e(\kappa)e^T(\kappa)\} = R_{yy}$, and $\mathrm{E}\{\epsilon(\kappa)e^T(\kappa)\} = R_{xy}$.

This function augments the stochastic model and calls `regrKalman` function.

regrKalman

Discrete time Kalman filter.

Syntax

```
[Xhat,Yhat] = regrkalman(U,Y,A,B,C,D,Rxx,Ryy,Rxy,x0)
[Xhat,Yhat] = regrkalman(U,Y,A,B,C,D,Rxx,Ryy,Rxy)
```

Input parameters

- `U`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `A`, `B`, `C`, `D`, `Rxx`, `Ryy`, and `Rxy`: System matrices
- `x0`: Initial state (default $\mathbf{0}$)

Return parameters

- `Xhat`: State sequence estimate
- `Yhat`: Corresponding output estimate

Comments

The assumed stochastic system structure is

$$\begin{cases} x(\kappa+1) = Ax(\kappa) + Bu(\kappa) + \epsilon(\kappa) \\ y(\kappa) = Cx(\kappa) + Du(\kappa) + e(\kappa), \end{cases}$$

where the white noise sequences $\epsilon(\kappa)$ and $e(\kappa)$ are characterized by the covariance matrices $\mathrm{E}\{\epsilon(\kappa)\epsilon^T(\kappa)\} = R_{xx}$, $\mathrm{E}\{e(\kappa)e^T(\kappa)\} = R_{yy}$, and $\mathrm{E}\{\epsilon(\kappa)e^T(\kappa)\} = R_{xy}$. The implementation of the procedure is very elementary — for example, the Riccati equation is solved using a fixed-length iteration, meaning that for badly conditioned matrices the results can be inaccurate.

`regrKM`

Function for determining the clusters using the K-means algorithm.

Syntax

```
[clusters] = regrkm(X,N,centers)
[clusters] = regrkm(X,N)
```

Input parameters

- `X`: Data to be modeled (size $k \times n$)
- `N`: Number of clusters
- `centers`: Initial cluster center points (optional)

Return parameter

- `clusters`: Vector (size $k \times 1$) showing the clusters for samples

Comments

If no initial cluster centers are given, the N first samples are used as centers.

regrMLR

Regression from X to Y, perhaps through a latent basis.

Syntax

```
[F,error,R2,stds] = regrmlr(X,Y,theta)
[F,error,R2,stds] = regrmlr(X,Y)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `theta`: Latent basis, orthogonal or not (optional)

Return parameters

- `F`: Mapping matrix, $\hat{Y} = X \cdot F$
- `error`: Prediction errors (size $k \times m$)
- `R2`: Data fitting criterion R^2
- `stds`: Estimated standard deviations of the parameters ($n \times m$)

Comments

This is the basic regression function in the Toolbox, used by most of the other regression methods.

If no θ is given, least-squares mapping from X to Y is constructed; otherwise, the data is first projected onto the latent basis, no matter how it has been constructed.

The matrix `stds` has the same structure as the model matrix F, revealing the estimated accuracies of the parameters. Depending on the assumed probability distribution of the error, one can study whether the parameter value "0" is plausible. If there is the latent structure θ defined, standard deviations are not calculated.

regrMLRC

Regression from X to Y, perhaps through a latent basis, when there are additional linear constraints for parameters.

Syntax

```
[F,error,R2,stds] = regrMLRC(X,Y,G,g,theta)
[F,error,R2,stds] = regrMLRC(X,Y,G,g)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `G` and `g`: Linear constraints in the form $GF = g$
- `theta`: Latent basis, orthogonal or not (optional)

Return parameters

- `F`: Mapping matrix, $\hat{Y} = X \cdot F$
- `error`: Prediction errors (size $k \times m$)
- `R2`: Data fitting criterion R^2
- `stds`: Estimated standard deviations of the parameters ($n \times m$)

Comments

If no θ is given, least-squares mapping from X to Y is constructed; otherwise, the data is first projected onto the latent basis, no matter how it has been constructed. In both cases, the final mapping between the input and the output fulfills the given constraints.

`regrOLS`

Orthogonal Least Squares (OLS).

Syntax

```
[F,error] = regrols(X,Y)
```

Input parameters

- `X`:Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)

Return parameters

- `F`: Mapping matrix, $\hat{Y} = X \cdot F$
- `error`: Prediction errors

Comments

Model construction is carried out using the QR factorization of X.

regrOutl

Interactive elimination of outliers.

Syntax

```
[W] = regroutl(X,W)
[W] = regroutl(X)
```

Input parameters

- X: Data to be modeled (size $k \times n$)
- W: Old vector (size $k \times 1$) containing "1" for valid samples (optional)

Return parameters

- W: New vector containing "1" for valid data (size $k \times 1$)

Comments

Function eliminates outliers interactively. Feedback is given on the graphical screen — mouse clicks toggle the status of the nearest point ("1" for valid data and "0" for invalid), and the vector of these values W is returned.

Vector W can also be refined if it is given as input argument.

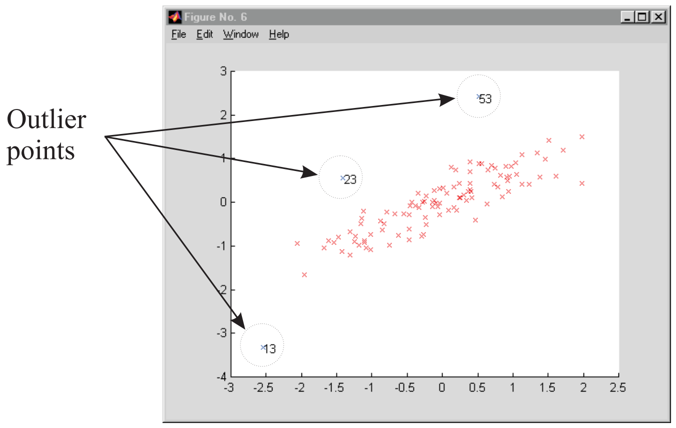

Figure B.8: Outliers, lone samples

regrP

Fitting data against a Gaussian distribution.

Syntax

```
[p] = regrP(X1,X2)
```

Input parameters

- `X1`: Input data block determining the distribution (size $k_1 \times n$)
- `X2`: Data block to be fitted (size $k_2 \times n$)

Return parameters

- `p`: Probabilities for data in `X2` to match `X1`

Comments

For each vector in `X2`, a probability value is returned — how well that vector matches the assumedly Gaussian distributions determined by data in `X1`, or "how probably a vector truly belonging to that distribution is farther from the center than vector in `X2` is".

Assuming that x has normal distribution, the measure $x^T \mathrm{E}\{xx^T\}^{-1} x$ has χ^2 distribution, and this understanding is exploited.

`regrPCA`

Principal Component Analysis (PCA) model construction.

Syntax

```
[theta,lambda,Q,T2] = regrpca(X,N)
[theta,lambda,Q,T2] = regrpca(X)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `N`: Number of latent variables (optional)

Return parameters

- `theta`: Latent basis
- `lambda`: Variances in latent basis directions
- `Q`,`T2`: Fitting criteria Q and T^2 (sizes $k \times 1$)

Comments

If the number of latent vectors N is not explicitly given, it is queried interactively (see `regrAskOrder`).

Based on this command, the Principal Component Regression can easiest be implemented as

```
F = regrmlr(X,Y,regrpca(X));
Ytest = Xtest*F;
```

`regrPLS`

Partial least Squares (PLS) model construction.

Syntax

```
[theta,phi,lambda] = regrpls(X,Y,N)
[theta,phi,lambda] = regrpls(X,Y)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `N`: Number of latent variables (optional)

Return parameters

- `theta`: Input block latent basis
- `phi`: Output block latent basis
- `lambda`: Correlation coefficients (unnormalized)

Comments

If the number of latent vectors N is not explicitly given, it is queried interactively (see `regrAskOrder`).

Based on this command, the actual regression can easiest be implemented as

```
F = regrmlr(X,Y,regrpls(X,Y));
Ytest = Xtest*F;
```

regrPPCA

Principal Component Analysis using iterative power method.

Syntax

```
[Z,L,iter] = regrppca(X,N,iterlimit)
[Z,L,iter] = regrppca(X,N)
[Z,L,iter] = regrppca(X)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `N`: Number of latent variables (optional)
- `iterlimit`: Maximum number of iterations (optional)

Return parameters

- `theta`: Eigenvectors
- `lambda`: Corresponding eigenvalues
- `iter`: Number of iterations needed

Comments

If there are eigenvectors with practically same eigenvalues, the convergence may be slow.

regrRBFN

Radial Basis Function regression model construction.

Syntax

```
[rbfmodel,error] = regrrbfn(X,Y,clusters,sigma)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `clusters`: Cluster index between 1 and N for all k samples, constructed, for example, by K-means (see `km`)
- `sigma`: Distribution of the Gaussians

Return parameters

- `rbfmodel`: Matrix containing the model (see below)
- `error`: Prediction errors

Comments

The structure of `rbfmodel` is the following:

```
rbfmodel = [centers;sigmas;weights]
```

where

- `centers`: Vector (size $n \times N$) containing cluster centers
- `sigmas`: Standard deviations in the clusters (size $1 \times N$)
- `weights`: Mappings (size $m \times N$) from clusters to outputs

This routine can be used as:

```
model = regrrbfn(X,Y,regrkm([X,Y],N));
Yhat = regrrbfr(X,model);
```

regrRBFR

Radial Basis Function regression.

Syntax

```
[Yhat] = regrrbfr(X,rbfmodel)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `rbfmodel`: Matrix containing the model

Return parameters

- `Yhat`: Estimated output data block (size $k \times m$)

Comments

See `regrRBFN`.

`regrReconc`

Data reconciliation.

Syntax

```
[v] = regrreconc(v,R,G,g)
```

Input parameters

- `v`: Data vector (size $n \times 1$)
- `R`: Measurement error covariance matrix (size $n \times n$)
- `G` and `g`: Linear constraints in the form $Gv = g$

Return parameter

- `v`: Modified data vector

Comments

After this operation, the data vector fulfills the given set of linear constraints.

`regrRR`

Ridge Regression.

Syntax

```
[F,error] = regrrr(X,Y,q)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `q`: “Stabiliaztion factor”

Return parameters

- `F`: Mapping matrix, so that $\hat{Y} = X \cdot F$
- `error`: Prediction errors

`regrScale`

Function for normalizing data variances.

Syntax

```
[X,backmap] = regrscale(DATA,w)
[X,backmap] = regrscale(DATA)
```

Input parameters

- `DATA`: Data to be manipulated (size $k \times n$)
- `w`: Intended variable variances or “importances” (optional)

Return parameters

- `X`: Scaled data matrix (size $k \times n$)
- `backmap`: Matrix for getting back to original coordinates

Comments

The original coordinates are restored as

```
DATA = X*backmap;
```

`regrShowClust`

Function for visualization of clusters.

Syntax

```
regrshowclust(X,c)
regrshowclust(X)
```

Input parameters

- `X`: Data matrix (size $k \times n$)
- `c`: Cluster indices for data (optional)

Comments

Samples classified in different classes are shown in different colours. If only one argument is given, one cluster is assumed.

The user is interactively asked to enter (in `Matlab` list form) the principal axes onto which the data is projected. If only one number is given, the horizontal axis is time k; if two or three axes are given, a two-dimensional or three-dimensional plot is constructed. The three-dimensional view can be rotated using mouse.

regrSSI

Combined stochastic-deterministic SubSpace Identification (simplified)

Syntax

```
[A,B,C,D,Rxx,Ryy,Rxy] = regrssi(U,Y,maxdim,N)
[A,B,C,D,Rxx,Ryy,Rxy] = regrssi(U,Y,maxdim)
```

Input parameters

- `U`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)
- `maxdim`: Assumed maximum possible system dimension
- `N`: System dimension (optional)

Return parameters

- `A`, `B`, `C`, `D`: System matrices
- `Rxx`, `Ryy`, `Rxy`: Noise covariances

Comments

If the number of states N is not explicitly given, it is queried interactively (see `regrAskOrder`). The implementation of the ideas of subspace identification is very simple, so that internally it is simple PCA that is applied, and Ridge Regression is used for carrying out the internal mappings.

The assumed system structure is compatible with the following structure:

$$\begin{cases} x(\kappa+1) = Ax(\kappa) + Bu(\kappa) + \epsilon(\kappa) \\ y(\kappa) = Cx(\kappa) + Du(\kappa) + e(\kappa), \end{cases}$$

where the white noise sequences $\epsilon(\kappa)$ and $e(\kappa)$ are characterized by the covariance matrices $\mathrm{E}\{\epsilon(\kappa)\epsilon^T(\kappa)\} = R_{xx}$, $\mathrm{E}\{e(\kappa)e^T(\kappa)\} = R_{yy}$, and $\mathrm{E}\{\epsilon(\kappa)e^T(\kappa)\} = R_{xy}$.

regrSSSI

Stochastic SubSpace Identification (simplified)

Syntax

```
[A,C,Rxx,Ryy,Rxy] = regrsssi(Y,maxdim,N)
[A,C,Rxx,Ryy,Rxy] = regrsssi(Y,maxdim)
```

Input parameters

- `Y`: Output data block (size $k \times m$)
- `maxdim`: Assumed maximum possible system dimension
- `N`: System dimension (optional)

Return parameters

- `A`, `C`: System matrices
- `Rxx`, `Ryy`, `Rxy`: Noise covariances

Comments

If the number of latent states N is not explicitly given, it is queried interactively (see `regrAskOrder`). The implementation of the ideas of subspace identification is very simple, so that internally it is simple PCA that is applied, and Ridge Regression is used for carrying out the internal mappings.

The assumed system structure is compatible with the stochastic Kalman filter (see `regrKalman`):

$$\begin{cases} x(\kappa+1) = Ax(\kappa) + \epsilon(\kappa) \\ y(\kappa) = Cx(\kappa) + e(\kappa), \end{cases}$$

where the white noise sequences $\epsilon(\kappa)$ and $e(\kappa)$ are characterized by the covariance matrices $\mathrm{E}\{\epsilon(\kappa)\epsilon^T(\kappa)\} = R_{xx}$, $\mathrm{E}\{e(\kappa)e^T(\kappa)\} = R_{yy}$, and $\mathrm{E}\{\epsilon(\kappa)e^T(\kappa)\} = R_{xy}$.

regrTLS

Total Least Squares regression.

Syntax

```
[F,error] = regrtls(X,Y)
```

Input parameters

- `X`: Input data block (size $k \times n$)
- `Y`: Output data block (size $k \times m$)

Return parameters

- `F`: Mapping matrix, $\hat{Y} = X \cdot F$
- `error`: Prediction errors (size $k \times m$)

`regrWeight`

Function for weighting the data.

Syntax

```
[X] = regrweight(DATA,w)
```

Input parameters

- `DATA`: Data to be manipulated (size $k \times n$)
- `w`: Data sample importances (size $k \times 1$)

Return parameters

- `X`: Weighted data matrix (size $k \times n$)

Comments

This function makes it possible to condition heteroscedastic data. Assuming that *invs* contains the inverses of the sample-wise a priori error variances, the following formulations can be employed:

```
theta = regrpca(regrweight(X,sqrt(invs)));
F = regrmlr(regrweight(X,sqrt(invs)),regrweight(Y,sqrt(invs)),theta);
```

`regrWhiten`

Function for whitening the data.

Syntax

```
[X,backmap] = regrwhiten(DATA)
```

Input parameters

- `DATA`: Data to be manipulated (size $k \times n$)

Return parameters

- `X`: Scaled data matrix (size $k \times n$)
- `backmap`: Matrix for getting back to original coordinates

Comments

The original coordinates are restored as

```
DATA = X*backmap;
```