LS-SVMlab Toolbox User's Guide

version 1.4

K. Pelckmans, J.A.K. Suykens, T. Van Gestel, J. De Brabanter, L. Lukas, B. Hamers, B. De Moor, J. Vandewalle

Katholieke Universiteit Leuven
Department of Electrical Engineering, ESAT-SCD-SISTA
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{ kristiaan.pelckmans, johan.suykens }@esat.kuleuven.ac.be

http://www.esat.kuleuven.ac.be/sista/lssvmlab/

October 2002

Acknowledgements

Research supported by *Research Council K.U.Leuven*: GOA-Mefisto 666, IDO (IOTA oncology, genetic networks), several PhD/postdoc & fellow grants; *Flemish Government*: FWO: PhD/postdoc grants, G.0407.02 (support vector machines), projects G.0115.01 (microarrays/oncology), G.0240.99 (multilinear algebra), G.0080.01 (collective intelligence), G.0413.03 (inference in bioi), G.0388.03 (microarrays for clinical use), G.0229.03 (ontologies in bioi), G.0197.02 (power islands), G.0141.03 (identification and cryptography), G.0491.03 (control for intensive care glycemia), G.0120.03 (QIT), research communities (ICCoS, ANMMM); AWI: Bil. Int. Collaboration Hungary, Poland, South Africa; IWT: PhD Grants, STWW-Genprom (gene promotor prediction), GBOU-McKnow (knowledge management algorithms), GBOU-SQUAD (quorum sensing), GBOU-ANA (biosensors); Soft4s (softsensors) *Belgian Federal Government*: DWTC (IUAP IV-02 (1996-2001) and IUAP V-22 (2002-2006)); PODO-II (CP/40: TMS and sustainibility); *EU*: CAGE; ERNSI; Eureka 2063-IMPACT; Eureka 2419-FliTE; *Contract Research/agreements*: Data4s, Electrabel, Elia, LMS, IPCOS, VIB; JS is a professor at K.U.Leuven Belgium and a postdoctoral researcher with FWO Flanders. TVG is postdoctoral researcher with FWO Flanders. BDM and JWDW are full professors at K.U.Leuven Belgium.

Contents

1 Introduction **4**

2 A birds eye view on LS-SVMlab **5**
2.1 Classification and Regression 5
2.1.1 Classification Extensions 6
2.1.2 Tuning, Sparseness, Robustness 6
2.1.3 Bayesian Framework 8
2.2 NARX Models and Prediction 8
2.3 Unsupervised Learning 8
2.4 Solving Large Scale Problems with Fixed Size LS-SVM 9

3 LS-SVMlab toolbox examples **10**
3.1 Classification 10
3.1.1 Hello world... 10
3.1.2 The Ripley data set 12
3.1.3 Bayesian Inference for Classification 14
3.1.4 Multi-class coding 16
3.2 Regression 17
3.2.1 A Simple Sinc Example 17
3.2.2 Bayesian Inference Applications for Regression 19
3.2.3 Robust Regression 20
3.2.4 Multiple Output Regression 21
3.2.5 A Time-Series Example: Santa Fe Laser Data Prediction 22
3.2.6 Fixed size LS-SVM 23
3.3 Unsupervised Learning using kernel based Principal Component Analysis 25

A MATLAB functions **26**
A.1 General Notation 26
A.2 Index of Function Calls 27
A.2.1 Training and Simulation 27
A.2.2 Object Oriented Interface 28
A.2.3 Training and Simulating Functions 29
A.2.4 Kernel Functions 30
A.2.5 Tuning, Sparseness and Robustness 31
A.2.6 Classification Extensions 32
A.2.7 Bayesian Framework 33
A.2.8 NARX models and Prediction 34
A.2.9 Unsupervised learning 35
A.2.10 Fixed Size LS-SVM 36
A.2.11 Demos 37
A.3 Alphabetical List of Function Calls 38
A.3.1 `AFE` 38

A.3.2 bay_errorbar . . . 39
A.3.3 bay_initlssvm . . . 41
A.3.4 bay_lssvm . . . 42
A.3.5 bay_lssvmARD . . . 44
A.3.6 bay_modoutClass . . . 46
A.3.7 bay_optimize . . . 48
A.3.8 bay_rr . . . 50
A.3.9 code, codelssvm . . . 52
A.3.10 crossvalidate . . . 55
A.3.11 deltablssvm . . . 57
A.3.12 denoise_kpca . . . 58
A.3.13 eign . . . 59
A.3.14 initlssvm, changelssvm . . . 60
A.3.15 kentropy . . . 63
A.3.16 kernel_matrix . . . 64
A.3.17 kpca . . . 65
A.3.18 latentlssvm . . . 66
A.3.19 leaveoneout . . . 67
A.3.20 leaveoneout_lssvm . . . 69
A.3.21 lin_kernel, MLP_kernel, poly_kernel, RBF_kernel . . . 71
A.3.22 linf, mae, medae, misclass, mse, trimmedmse . . . 72
A.3.23 plotlssvm . . . 74
A.3.24 predict . . . 75
A.3.25 prelssvm, postlssvm . . . 77
A.3.26 ridgeregress . . . 78
A.3.27 robustlssvm . . . 79
A.3.28 roc . . . 80
A.3.29 simlssvm . . . 82
A.3.30 sparselssvm . . . 84
A.3.31 trainlssvm . . . 85
A.3.32 tunelssvm & gridsearch . . . 87
A.3.33 validate . . . 90
A.3.34 windowize & windowizeNARX . . . 92

Chapter 1

Introduction

Support Vector Machines (SVM) is a powerful methodology for solving problems in nonlinear classification, function estimation and density estimation which has also led to many other recent developments in kernel based learning methods in general [3, 16, 17, 34, 33]. SVMs have been introduced within the context of statistical learning theory and structural risk minimization. In the methods one solves convex optimization problems, typically quadratic programs. Least Squares Support Vector Machines (LS-SVM) are reformulations to standard SVMs [21, 28] which lead to solving linear KKT systems. LS-SVMs are closely related to regularization networks [5] and Gaussian processes [37] but additionally emphasize and exploit primal-dual interpretations. Links between kernel versions of classical pattern recognition algorithms such as kernel Fisher discriminant analysis and extensions to unsupervised learning, recurrent networks and control [22] are available. Robustness, sparseness and weightings [23] can be imposed to LS-SVMs where needed and a Bayesian framework with three levels of inference has been developed [29, 32]. LS-SVM alike primal-dual formulations are given to kernel PCA [24], kernel CCA and kernel PLS [25]. For ultra large scale problems and on-line learning a method of Fixed Size LS-SVM is proposed, which is related to a Nyström sampling [6, 35] with active selection of support vectors and estimation in the primal space.

The present *LS-SVMlab toolbox User's Guide* contains Matlab/C implementations for a number of LS-SVM algorithms related to classification, regression, time-series prediction and unsupervised learning. References to commands in the toolbox are written in `typewriter` font.

A main reference and overview on least squares support vector machines is

J.A.K. Suykens, T. Van Gestel, J. De Brabanter, B. De Moor, J. Vandewalle,
Least Squares Support Vector Machines,
World Scientific, Singapore, 2002 (ISBN 981-238-151-1).

The LS-SVMlab homepage is

http://www.esat.kuleuven.ac.be/sista/lssvmlab/

Chapter 2

A birds eye view on LS-SVMlab

The toolbox is mainly intended for use with the commercial Matlab package. However, the core functionality is written in C-code. The Matlab toolbox is compiled and tested for different computer architectures including Linux and Windows. Most functions can handle datasets up to 20000 data points or more. LS-SVMlab's interface for Matlab consists of a basic version for beginners as well as a more advanced version with programs for multi-class encoding techniques and a Bayesian framework. Future versions will gradually incorporate new results and additional functionalities.

The organization of the toolbox is schematically shown in Figure 2.1. A number of functions are restricted to LS-SVMs (these include the extension "`lssvm`" in the function name), the others are generally usable. A number of demos illustrate how to use the different features of the toolbox. The Matlab function interfaces are organized in two principal ways: the functions can be called either in a *functional way* or using an *object* oriented structure (referred to as the **model**) as e.g. in Netlab [14], depending on the user's choice[1].

2.1 Classification and Regression

Function calls: `trainlssvm, simlssvm, plotlssvm, prelssvm, postlssvm;`
Demos: `Subsections 3.1, 3.2, demofun, democlass.`

The Matlab toolbox is built around a fast LS-SVM training and simulation algorithm. The corresponding function calls can be used for classification as well as for function estimation. The function `plotlssvm` displays the simulation results of the **model** in the region of the training points.

To avoid failures and ensure performance of the implementation, three different implementations are included. The most performant is the CMEX implementation (`lssvm.mex*`), based on C-code linked with Matlab via the CMEX interface. More reliable (less system specific) is the C-compiled executable (`lssvm.x`) which passes the parameters to/from Matlab via a buffer file. Both use the fast conjugate gradient algorithm to solve the set of linear equations [8]. The C-code for training takes advantage of previously calculated solutions by caching the firstly calculated kernel evaluations up to 64 Mb of data. Less performant but stable, flexible and straightforward coded is the implementation in Matlab (`lssvmMATLAB.m`) which is based on the Matlab matrix division (backslash command \).

Functions for single and multiple output regression and classification are available. Training and simulation can be done for each output separately by passing different kernel functions, kernel and/or regularization parameters as a column vector. It is straightforward to implement other kernel functions in the toolbox.

[1] See http://www.kernel-machines.org/software.html for other software in kernel based learning techniques.

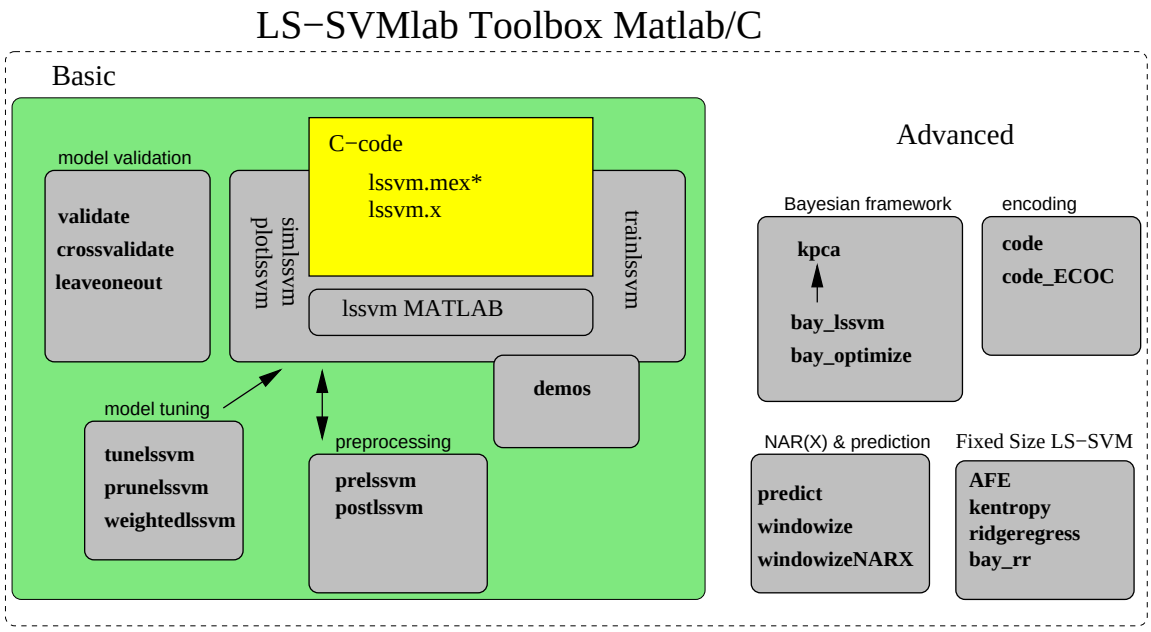


Figure 2.1: *Schematic illustration of the organization of LS-SVMlab. Each box contains the names of the corresponding algorithms. The function names with extension "*`lssvm`*" are LS-SVM method specific. The dashed box includes all functions of a more advanced toolbox, the large grey box those that are included in the basic version.*

The performance of a model depends on the scaling of the input and output data. An appropriate algorithm detects and appropriately rescales continuous, categorical and binary variables (`prelssvm`, `postlssvm`).

2.1.1 Classification Extensions

Function calls: `codelssvm, code, deltablssvm, roc, latentlssvm`;
Demos: `Subsection 3.1, democlass`.

A number of additional function files are available for the classification task. The latent variable of simulating a model for classification (`latentlssvm`) is the continuous result obtained by simulation which is discretised for making the final decisions. The Receiver Operating Characteristic curve [9] (`roc`) can be used to measure the performance of a classifier. Multiclass classification problems are decomposed into multiple binary classification tasks [30]. Several coding schemes can be used at this point: minimum output, one-versus-one, one-versus-all and error correcting coding schemes. To decode a given result, the Hamming distance, loss function distance and Bayesian decoding can be applied. A correction of the bias term can be done, which is especially interesting for small data sets.

2.1.2 Tuning, Sparseness, Robustness

Function calls: `tunelssvm, validate, crossvalidate, leaveoneout, robustlssvm, sparselssvm`;
Demos: `Subsections 3.1.2, 3.1.4, 3.2.3, 3.2.5, demofun, democlass, demomodel`.

A number of methods to estimate the generalization performance of the trained model are included. The estimate of the performance based on a fixed testset is calculated by `validate`. For

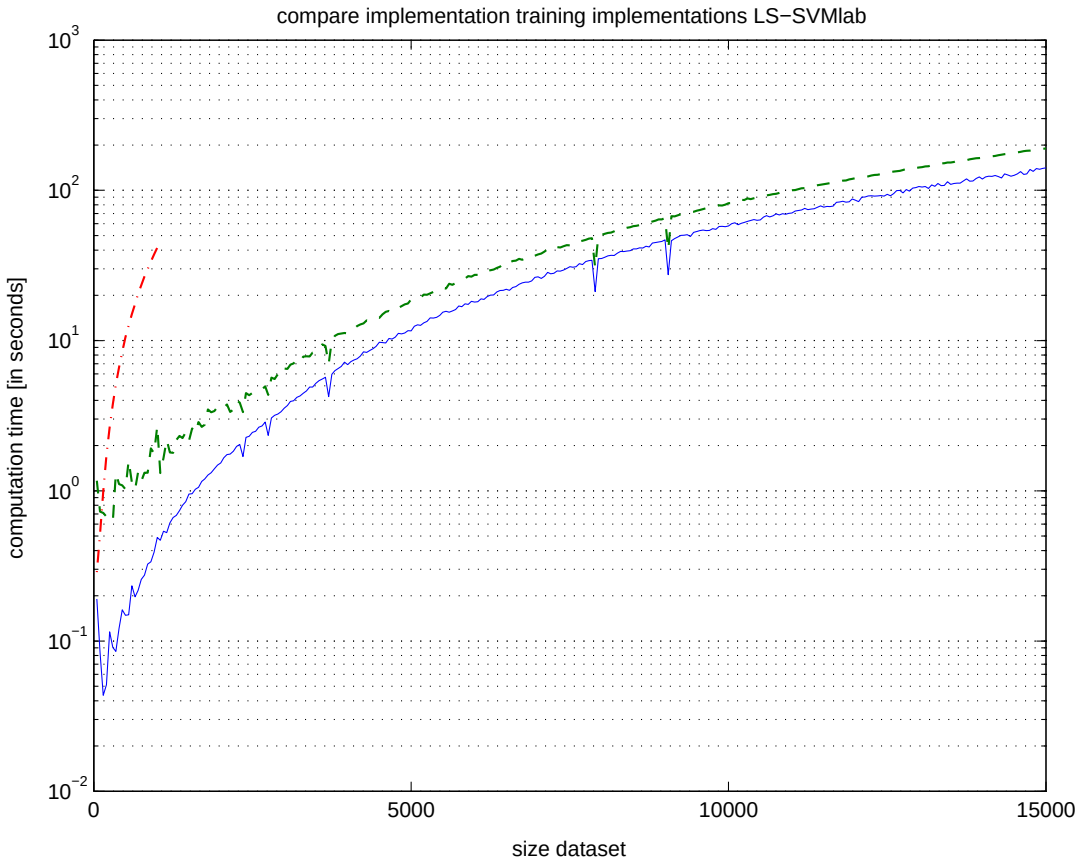


Figure 2.2: *Indication of the performance for the different training implementations of LS-SVMlab. The solid line indicates the performance of the CMEX interface. The dashed line shows the performance of the CFILE interface and the dashed-dotted line indicated the performance of the pure MATLAB implementation.*

classification, the rate of misclassifications (`misclass`) can be used. Estimates based on repeated training and validation are given by `crossvalidate` and `leaveoneout`. The implementation of these include a bias correction term. A robust crossvalidation score function [4] is called by `rcrossvalidate`. These performance measures can be used to tune the hyper-parameters (e.g. the regularization and kernel parameters) of the LS-SVM (`tunelssvm`). Reducing the model complexity of a LS-SVM can be done by iteratively pruning the less important support values (`sparselssvm`) [23]. In the case of outliers in the data or non-Gaussian noise, corrections to the support values will improve the model (`robustlssvm`) [23].

2.1.3 Bayesian Framework

Function calls: `bay_lssvm, bay_optimize, bay_lssvmARD, bay_errorbar, bay_modoutClass, kpca, eign`;
Demos: `Subsections 3.1.3, 3.2.2`.

Functions for calculating the posterior probability of the model and hyper-parameters at different levels of inference are available (`bay_lssvm`) [26, 32]. Errors bars are obtained by taking into account model- and hyper-parameter uncertainties (`bay_errorbar`). For classification [29], one can estimate the posterior class probabilities (this is also called the *moderated output*) (`bay_modoutClass`). The Bayesian framework makes use of the eigenvalue decomposition of the kernel matrix. The size of the matrix grows with the number of data points. Hence, one needs approximation techniques to handle large datasets. It is known that mainly the principal eigenvalues and corresponding eigenvectors are relevant. Therefore, iterative approximation methods such as the Nyström method [31, 35] are included, which is also frequently used in Gaussian processes. Input selection can be done by Automatic Relevance Determination (`bay_lssvmARD`) [27]. In a backward variable selection, the third level of inference of the Bayesian framework is used to infer the most relevant inputs of the problem.

2.2 NARX Models and Prediction

Function calls: `predict, windowize`;
Demo: `Subsection 3.2.5`.

Extensions towards nonlinear NARX systems for time series applications are available [25]. A NARX model can be built based on a nonlinear regressor by estimating in each iteration the next output value given the past output (and input) measurements. A dataset is converted into a new input (the past measurements) and output set (the future output) by `windowize` and `windowizeNARX` for respectively the time series case and in general the NARX case with exogenous input. Iteratively predicting (in recurrent mode) the next output based on the previous predictions and starting values is done by `predict`.

2.3 Unsupervised Learning

Function calls: `kpca, denoise_kpca`;
Demo: `Subsection 3.3`.

Unsupervised learning can be done by kernel based PCA (`kpca`) as described by [19], for which recently a primal-dual interpretation with support vector machine formulation has been given in [24], which has also be further extended to kernel canonical correlation analysis [25] and kernel PLS.

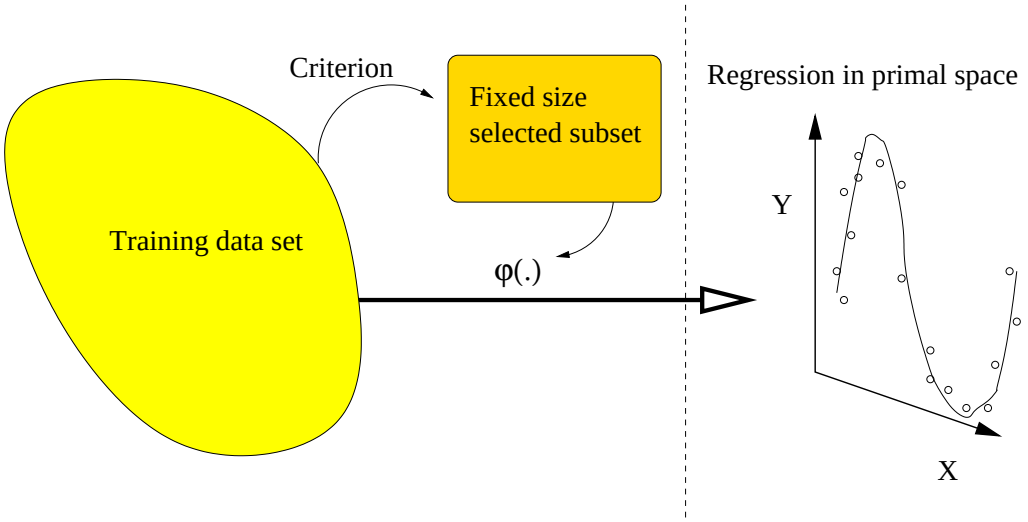


Figure 2.3: *Fixed Size LS-SVM is a method for solving large scale regression and classification problems. The number of support vectors is pre-fixed beforehand and the support vectors are selected from a pool of training data. After estimating eigenfunctions in relation to a Nyström sampling with selection of the support vectors according to an entropy criterion, the LS-SVM model is estimated in the primal space.*

2.4 Solving Large Scale Problems with Fixed Size LS-SVM

Function calls: `demo_fixedsize, AFE, kentropy`;
Demos: `Subsection 3.2.6, demo_fixedsize`.

Classical kernel based algorithms like e.g. LS-SVM [21] typically have memory and computational requirements of $O(N^2)$. Recently, work on large scale methods proposes solutions to circumvent this bottleneck [25, 19].

For large datasets it would be advantageous to solve the least squares problem in the primal weight space because then the size of the vector of unknowns is proportional to the feature vector dimension and not to the number of datapoints. However, the feature space mapping induced by the kernel is needed in order to obtain non-linearity. For this purpose, a method of fixed size LS-SVM is proposed [25] (Figure 2.3). Firstly the Nyström method [29, 35] can be used to estimate the feature space mapping. The link between Nyström sampling, kernel PCA and density estimation has been discussed in [6]. In fixed size LS-SVM these links are employed together with the explicit primal-dual LS-SVM interpretations. The support vectors are selected according to a quadratic Renyi entropy criterion (`kentropy`). In a last step a regression is done in the primal space which makes the method suitable for solving large scale nonlinear function estimation and classification problems. A Bayesian framework for ridge regression [11, 29] (`bay_rr`) can be used to find a good regularization parameter. The method of fixed size LS-SVM is suitable for handling very large data sets, adaptive signal processing and transductive inference.

An alternative criterion for subset selection was presented by [1, 2], which is closely related to [35] and [19]. It measures the quality of approximation of the feature space and the space induced by the subset (see Automatic Feature Extraction or `AFE`). In [35] the subset was taken as a random subsample from the data (`subsample`).

Chapter 3

LS-SVMlab toolbox examples

3.1 Classification

At first, the possibilities of the toolbox for classification tasks are illustrated.

3.1.1 Hello world...

A simple example shows how to start using the toolbox for a classification task. We start with constructing a simple example dataset according to the right formatting. Data are represented by doubles, each row of the matrix contains one datapoint:

```
>> X = 2.*rand(30,2)-1;
>> Y = sign(sin(X(:,1))+X(:,2));
>> X

X =

    0.9003   -0.9695
   -0.5377    0.4936
    0.2137   -0.1098
   -0.0280    0.8636
    0.7826   -0.0680
    0.5242   -0.1627
    ....       ....
   -0.4556    0.7073
   -0.6024    0.1871

>> Y

Y =

    -1
    -1
     1
     1
     1
     1
     ...
     1
    -1
```

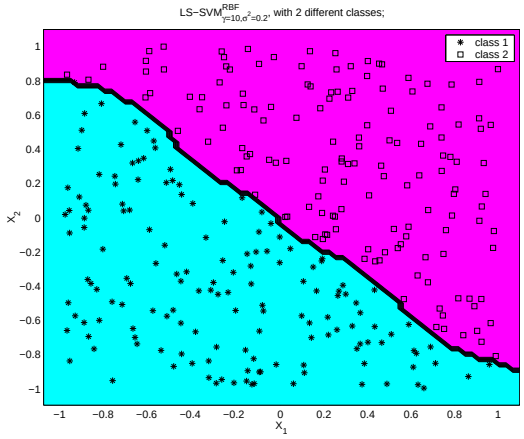


Figure 3.1: Figure generated by `plotlssvm` in the simple classification task.

In order to make an LS-SVM model, we need 2 parameters: γ (`gam`) is the regularization parameter, determining the trade-off between the fitting error minimization and smoothness. In the common case of the RBF kernel, σ^2 (`sig2`) is the bandwidth:

```
>> gam = 10;
>> sig2 = 0.2;
>> type = 'classification';
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel'});
```

The parameters and the variables relevant for the LS-SVM are passed as one cell. This cell allows for consistent default handling of LS-SVM parameters and syntactical grouping of related arguments. This definition should be used consistently throughout the use of the LS-SVM in the following calls: e.g. `simlssvm`, `plotlssvm` and others. The corresponding object oriented interface to LS-SVMlab leads to shorter function calls (see `demomodel`).

By default, the data are **preprocessed** by applying the function `prelssvm` to the raw data and the function `postlssvm` on the predictions of the model. This option can explicitly be stated in the call:

```
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel','preprocess'});
```

or be switched off:

```
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel','original'});
```

Remember to consistently use the same option in all successive calls.

To evaluate new points for this model, the function `simlssvm` is used. One can determine the `rate` of misclassifications represented by a number between 0 and 1. The last command composes a plot of the obtained LS-SVM (Figure 3.1).

```
>> Xt = 2.*rand(10,2)-1;
>> Yt = sign(sin(Xt(:,1))+Xt(:,2));
>> Ytest = simlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,b},Xt);
>> rate = misclass(Yt,Ytest);
>> plotlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,b});
```

3.1.2 The Ripley data set

The well-known Ripley dataset problem consists of two classes where the data for each class have been generated by a mixture of two Gaussian distributions (Figure 3.2a).

First, let us build an LS-SVM on the dataset and determine suitable hyperparameters:

```
>> % load dataset ...
>> type = 'classification';
>> L_fold = 10; % L-fold crossvalidation
>> [gam,sig2] = tunelssvm({X,Y,type,1,1,'RBF_kernel'},...
              'gridsearch',{},'crossvalidate',{X,Y,L_fold,'misclass'});
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel'});
>> plotlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,b});
```

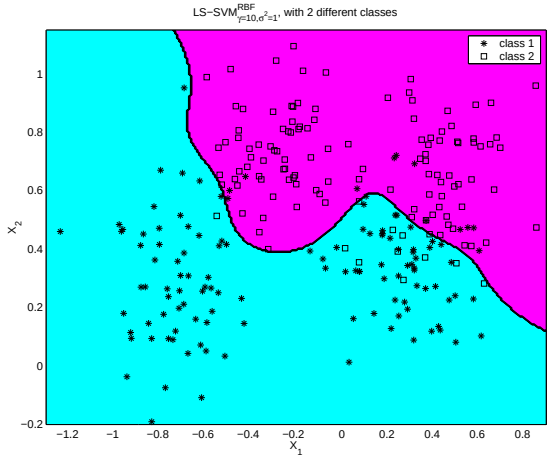
The Receiver Operating Characteristic (ROC) curve gives information about the quality of the classifier:

```
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel'});
>> Y_latent = latentlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,b},X);
>> [opp,se,deltab,oneMinusSpec,Sens]=roc(Y_latent,Y);
>> [deltab oneMinusSpec Sens]
ans =


   -2.1915    1.0000    1.0000
   -1.1915    0.9920    1.0000
   -1.1268    0.9840    1.0000
   -1.0823    0.9760    1.0000

    ...       ...       ...

   -0.2699    0.1840    0.9360
   -0.2554    0.1760    0.9360
   -0.2277    0.1760    0.9280
   -0.1811    0.1680    0.9280

    ...       ...       ...

    1.1184         0    0.0080
    1.1220         0         0
    2.1220         0         0
```

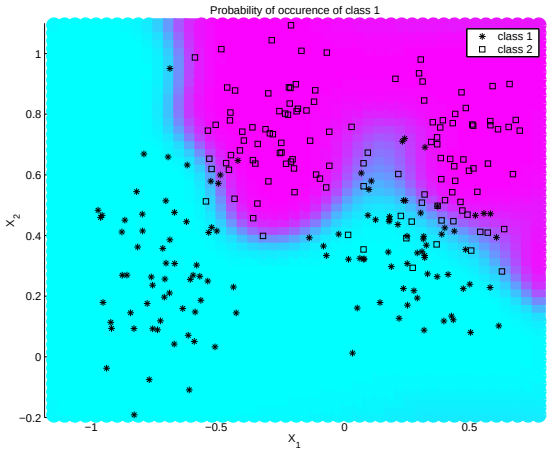
The corresponding ROC curve is shown on Figure 3.2c. This information can be used to further introduce prior knowledge in the classifier. A bias term correction can be found from the previous outcome:

```
>> plotlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,-0.2277});
```

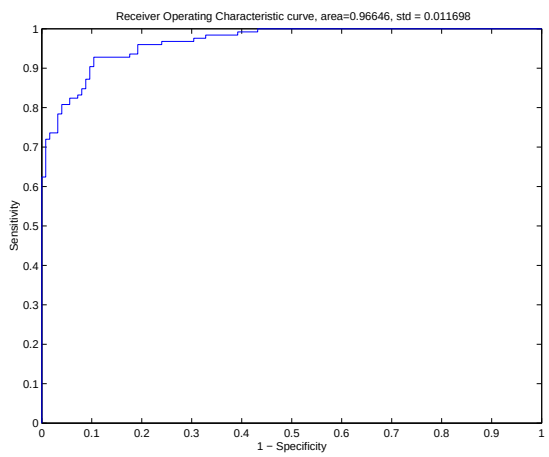
The result is shown in Figure 3.2d.



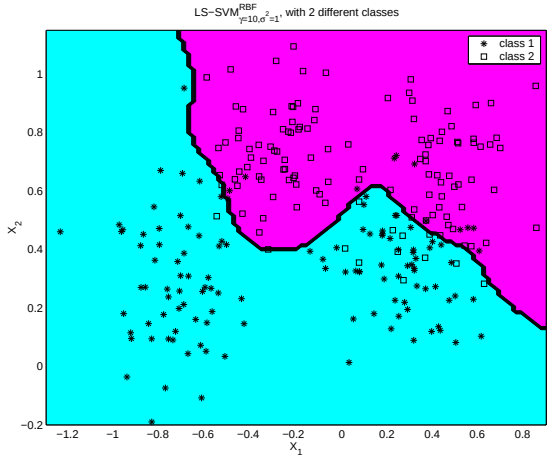
(a) Original Classifier



(b) Moderated Output



(c) ROC Curve



(d) Bias Term Correction

Figure 3.2: ROC curve and bias term correction on the Ripley classification task. **(a)** Original LS-SVM classifier. **(b)** Moderated output of the LS-SVM classifier on the Ripley data set. Shown are the probabilities to belong to the positive class (red: probability is 0, green: probability is 1). **(c)** Receiver Operating Characteristic curve. **(d)** The bias term correction can be used to avoid misclassifications for one of the two classes.

3.1.3 Bayesian Inference for Classification

This subsection further proceeds on the results of Subsection 3.1.2. A Bayesian framework is used to optimize the hyperparameters and to infer the moderated output. The optimal regularization parameter `gam` and kernel parameter `sig2` can be found by optimizing the cost on the second and the third level of inference, respectively. As the corresponding cost function is only smooth in the region of the optimum, it is recommended to initiate the model with appropriate starting values:

```
>> [gam, sig2] = bay_initlssvm({X,Y,type,gam,sig2,'RBF_kernel'});
```

Optimization on the second level leads to an optimal regularization parameter:

```
>> [model, gam_opt] = bay_optimize({X,Y,type,gam,sig2,'RBF_kernel'},2);
```

Optimization on the third level leads to an optimal kernel parameter:

```
>> [cost_L3,sig2_opt] = bay_optimize({X,Y,type,gam_opt,sig2,'RBF_kernel'},3);
```

The posterior class probabilies are found by incorporating the uncertainty of the model parameters:

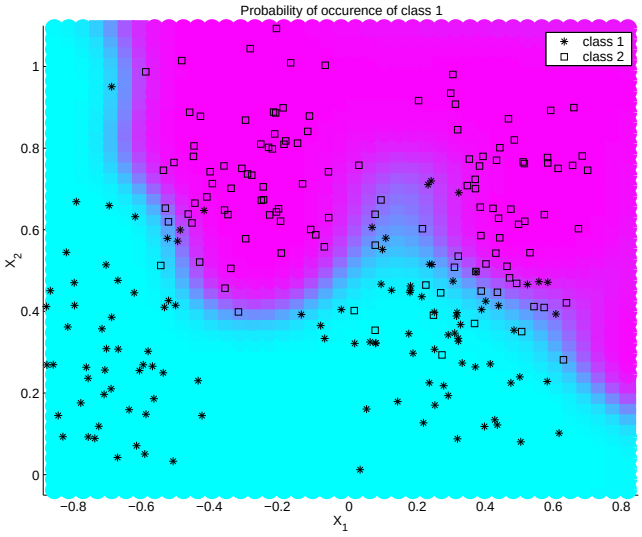
```
>> Ymodout = bay_modoutClass({X,Y,type,gam_opt,sig2_opt,'RBF_kernel'});
```

One can specify a prior class probability in the moderated output in order to compensate for an unbalanced number of training data points in the two classes. When the training set contains N^+ positive instances and N^- negative ones, the moderated output is calculated as:

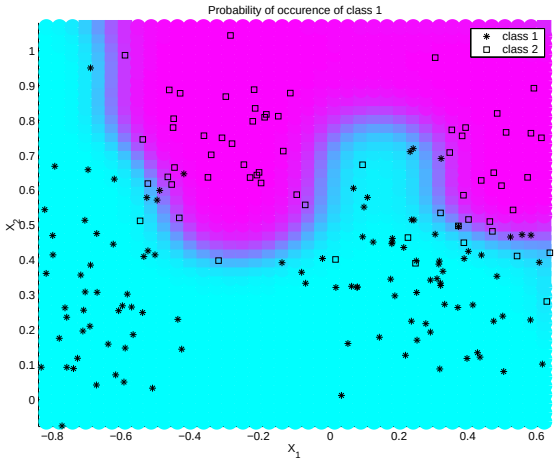
$$\text{prior} = \frac{N^+}{N^+ + N^-}$$

```
>> Np = 10;
>> Nn = 50;
>> prior = Np / (Nn + Np);
>> Posterior_class_P = bay_modoutClass({X,Y,type,gam_opt,sig2_opt,'RBF_kernel'},...
                                    'figure', prior);
```

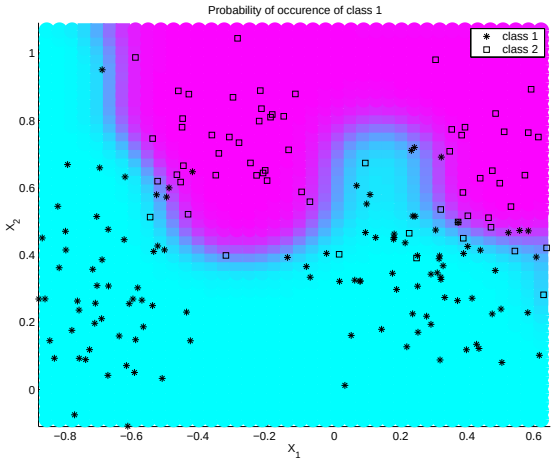
The results are shown in Figure 3.3.



(a) Moderated Output



(b) Unbalanced subset



(c) With correction for unbalancing

Figure 3.3: **(a)** Moderated output of the LS-SVM classifier on the Ripley data set. The colors indicate the probability to belong to a certain class; **(b)** This example shows the moderated output of an unbalanced subset of the Ripley data; **(c)** One can compensate for unbalanced data in the calculation of the moderated output. One can notice that the area of the green zone with the positive samples increases by the compensation. The red zone shrinks accordingly.

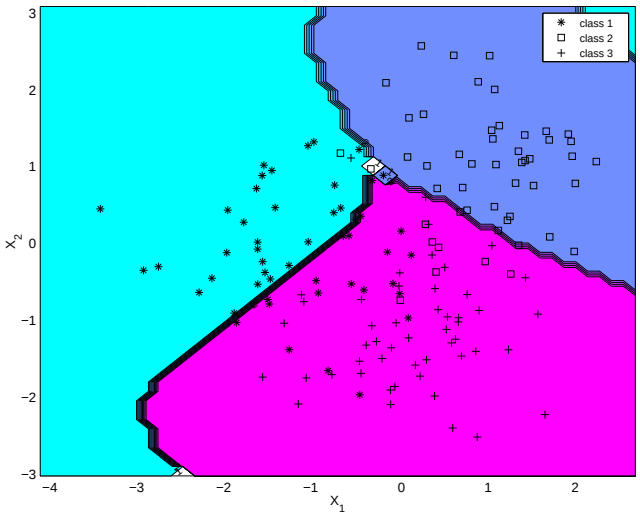


Figure 3.4: LS-SVM multi-class example with one versus one encoding.

3.1.4 Multi-class coding

The following example shows how to use encoding schemes for multi-class problems. The encoding and decoding are seen as a separate and independent preprocessing and postprocessing step respectively (Figure 3.4).

```
>> % load multiclass data ...
>> [Ycode, codebook, old_codebook] = code(Y,'code_MOC');
>>
>>   [alpha,b] = trainlssvm({X,Ycode,'class',gam,sig2});
>>   Yhc = simlssvm({X,Ycode,'class',gam,sig2},{alpha,b},Xtest);
>>
>> Yhc = code(Yh,old_codebook,[],codebook,'codedist_hamming');
```

3.2 Regression

3.2.1 A Simple Sinc Example

This is a simple demo, solving a simple regression task using LS-SVMlab. A dataset is constructed in the right formatting. The data are represented as matrices where each row contains one datapoint:

```
>> X = (-3:0.2:3)';
>> Y = sinc(X)+normrnd(0,0.1,length(X),1);
>> X

X =

   -3.0000
   -2.8000
   -2.6000
   -2.4000
   -2.2000
   -2.0000
    ...
    2.8000
    3.0000

>> Y =

Y =

   -0.0433
   -0.0997
    0.1290
    0.1549
   -0.0296
    0.1191
    ...
    0.1239
   -0.0400
```

In order to make an LS-SVM model, we need 2 parameters: γ (`gam`) is the regularization parameter, determining the trade-off between the fitting error minimization and smoothness of the estimated function. σ^2 (`sig2`) is the kernel function parameter.

```
>> gam = 10;
>> sig2 = 0.2;
>> type = 'function estimation';
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel'});
```

The parameters and the variables relevant for the LS-SVM are passed as one cell. This cell allows for consistent default handling of LS-SVM parameters and syntactical grouping of related arguments. This definition should be used consistently throughout the use of that LS-SVM in e.g. the following calls `simlssvm`, `plotlssvm`. The object oriented interface to LS-SVMlab leads to shorter function calls (see `demomodel`).

By default, the data are **preprocessed** by applying the function `prelssvm` to the raw data and the function `postlssvm` on the predictions of the model. This option can explicitly be stated in the call:

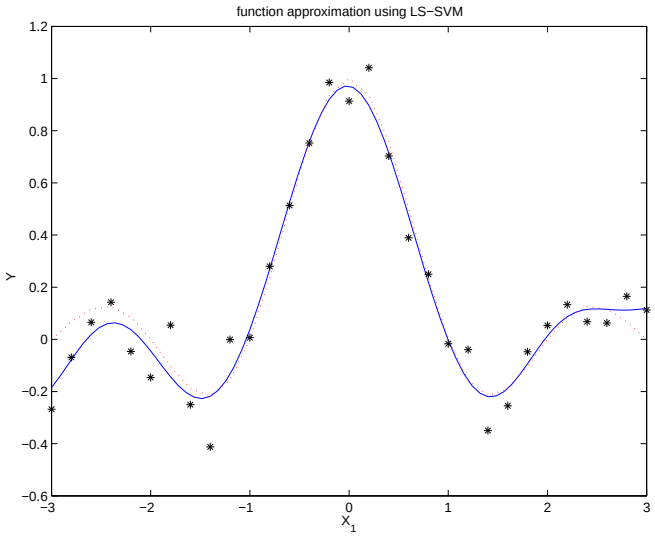


Figure 3.5: Simple regression problem. The solid line indicates the estimated outputs, the dotted line represents the true underlying function. The stars indicate the training data points.

```
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel','preprocess'});
```

or can be switched off:

```
>> [alpha,b] = trainlssvm({X,Y,type,gam,sig2,'RBF_kernel','original'});
```

Remember to consistently use the same option in all successive calls.

To evaluate new points for this model, the function `simlssvm` is used. At first, test data is generated:

```
>> Xt = normrnd(0,3,10,1);
```

Then, the obtained model is simulated on the test data:

```
>> Yt = simlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,b},Xt);

ans =

    0.9372
    0.0569
    0.8464
    0.1457
    0.1529
    0.6050
    0.5861
    0.0398
   -0.0865
    0.1517
```

```
>> figure;plotlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,b});
```

The LS-SVM result can be displayed if the dimension of the input data is 1 or 2.

```
>> plotlssvm({X,Y,type,gam,sig2,'RBF_kernel'},{alpha,b});
```

All plotting is done with this simple command. It looks for the best way of displaying the result (Figure 3.5).

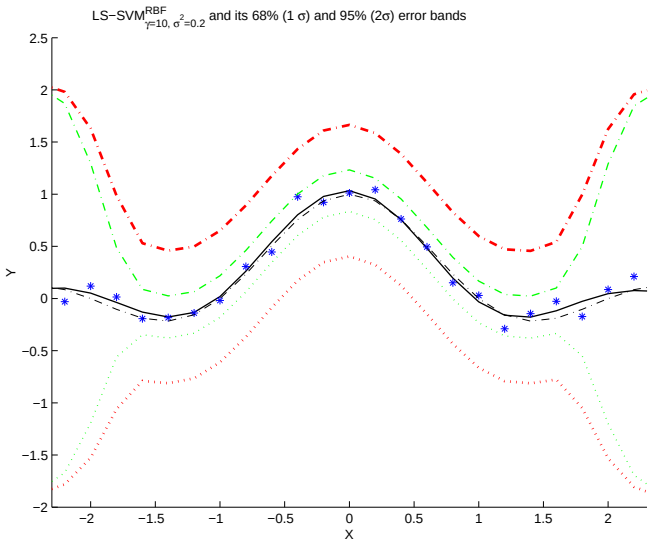


Figure 3.6: This figure gives the 68% errorbars (green dotted and dashed-dotted line) and the 95% errorbars (red dotted and dashed-dotted line) of the LS-SVM estimate (solid line) of a simple sinc function.

3.2.2 Bayesian Inference Applications for Regression

The following data set is constructed:

```
>> type = 'function';
>> X = normrnd(0,2,100,3);
>> Y = sinc(X(:,1)) + 0.05.*X(:,2) +normrnd(0,.1,size(X,1),1);
```

An LS-SVM for regression is trained on the data:

```
>> [alpha,b] = trainlssvm({X,Y,type, 10,3});
>> Yh = simlssvm({X,Y,type, 10,3},Xt);
```

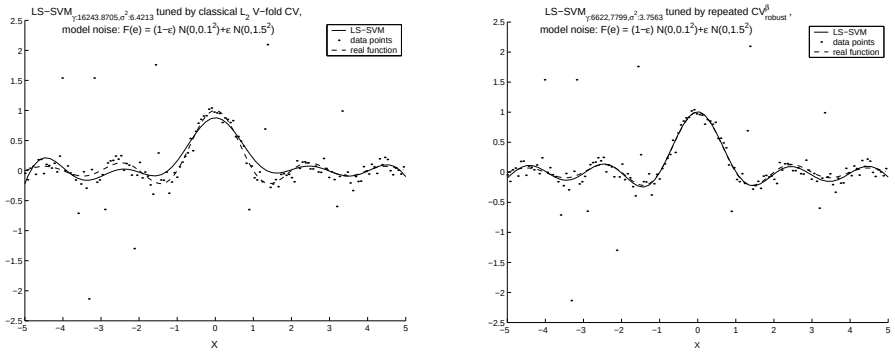
The errorbars on the training data are computed using Bayesian inference:

```
>> sig2e = bay_errorbar({X,Y,type, 10,3},'figure');
```

Automatic relevance determination is used to determine the subset of the most relevant inputs for the proposed model:

```
>> inputs = bay_lssvmARD({X,Y,type, 10,3});
>> [alpha,b] = trainlssvm({X(:,dimensions),Y,type, 10,1});
>> Yh = simlssvm({X(:,dimensions),Y,type, 10,1},Xt);
```

See Figure 3.6 for the resulting error band.



(a) (b)

Figure 3.7: *Experiments on a noisy sinc dataset with 15% outliers.* **(a)** *Application of the standard training and hyperparameter selection techniques;* **(b)** *Application of a weighted LS-SVM training together with a robust crossvalidation score function, which enhances the test set performance.*

3.2.3 Robust Regression

First, a dataset containing 15% outliers is constructed:

```
>> X = (-5:.07:5)';
>> epsilon = 0.15;
>> sel = rand(length(X),1)>epsilon;
>> Y = sinc(X)+sel.*normrnd(0,.1,length(X),1)+(1-sel).*normrnd(0,2,length(X),1);
```

Robust training is performed by `robustlssvm`:

```
>> gam = 10;
>> sig2 = 0.2;
>> [alpha,b] = robustlssvm({X,Y,'function',gam,sig2});
```

The tuning of the hyperparameters is performed by `rcrossvalidate`:

```
>> performance = rcrossvalidate({X,Y,'function',gam,sig2},X,Y,10)
>> costfun = 'rcrossvalidate';
>> costfun_args = {X,Y,10};
>> optfun = 'gridsearch';
>> [gam, sig2] = tunelssvm({X,Y,'f',gam,sig2}, optfun, {}, costfun, costfun_args)
```

3.2.4 Multiple Output Regression

In the case of multiple output data one can treat the different outputs separately. One can also let the toolbox do this by passing the right arguments. This case illustrates how to handle multiple outputs:

```
>> % load data in X and Y
>> % where dimension Y is N x 3
>>
>> sigs = [1 2 1.5];
>> [alpha,b] = trainlssvm({X,Y,'classification',gam,sigs});
>> Yhs = simlssvm({X,Y,'classification',gam,sigs},{alpha,b},Xt);
>>
>> % different kernels
>> kernels = {'lin_kernel','RBF_kernel','RBF_kernel'};
>> kpars = [0 2 2];
>> gams =  [1 2 3];
>> [alpha,b] = trainlssvm({X,Y,'classification',gams,kpars,kernels});
>> Yhs = simlssvm({X,Y,'classification',gam,kpars,kernels},{alpha,b},Xt);
>>
>> % tune the different parameters
>> [sigs,gam] = tunelssvm({X,Y,'classification',gam,kpars,kernels});
```

3.2.5 A Time-Series Example: Santa Fe Laser Data Prediction

Using the static regression technique, a nonlinear feedforward prediction model can be built. The NARX model takes the past measurements as input to the model

```
>> % load time-series in X
>> delays = 50;
>> Xu = windowize(X,1:delays+1);
```

The hyperparameters can be determined on a validation set. Here the data are split up in 2 distinct set of successive signals: one for training and one for validation:

```
>> Xtra = Xu(1:400,1:delays); Ytra = Xu(1:400,end);
>> Xval = Xu(401:950,1:delays); Yval = Xu(401:950,end);
```

Validation is based on feedforward simulation of the validation set using the feedforwardly trained model:

```
>> performance = ...
       validate({Xu(:,1:delays),Xu(:,end),'function',1,1,'RBF_kernel'},...
                Xtra, Ytra, Xval, Yval);
>> [gam,sig2] = tunelssvm({Xu(:,1:delays),Xu(:,end),'function',1,1,'RBF_kernel'},...
                          'gridsearch',{},'validate',{Xtra, Ytra, Xval, Yval});
```

The number of lags can be determined by Automatic Relevance Determination, although this technique is known to work suboptimal in the context of recurrent models

```
>> dimensions = bay_lssvmARD({Xu(:,1:delays),Xu(:,end),...
                               'function',10,50,'RBF_kernel'});
```

Prediction of the next 100 points is done in a recurrent way:

```
>> [alpha,b] = trainlssvm({Xu(:,dimensions),Xu(:,end),...
                          'function',gam,sig2,'RBF_kernel'});
>> prediction = predict({Xu(:,dimensions),Xu(:,end),...
                        'function',gam,sig2,'RBF_kernel'},X(dimensions),100);
>> plot([pred X(delays+1:100)']);
```

In Figure 3.8 results are shown for the Santa Fe laser data.

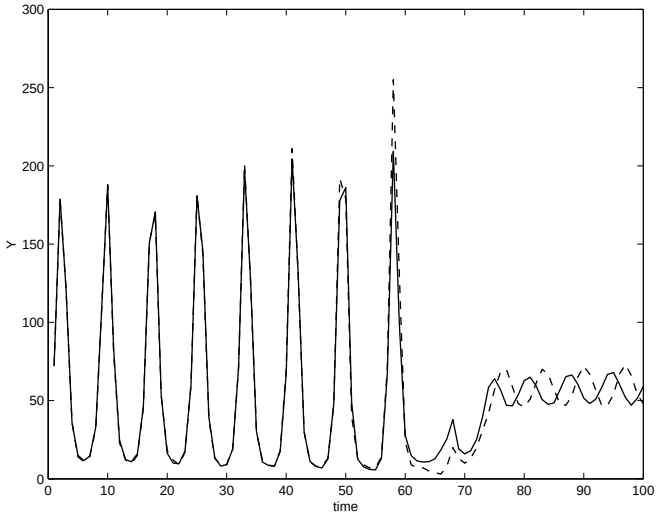


Figure 3.8: *The solid line denotes the Santa Fe chaotic laser data. The dashed line shows the iterative prediction using LS-SVM with the RBF kernel.*

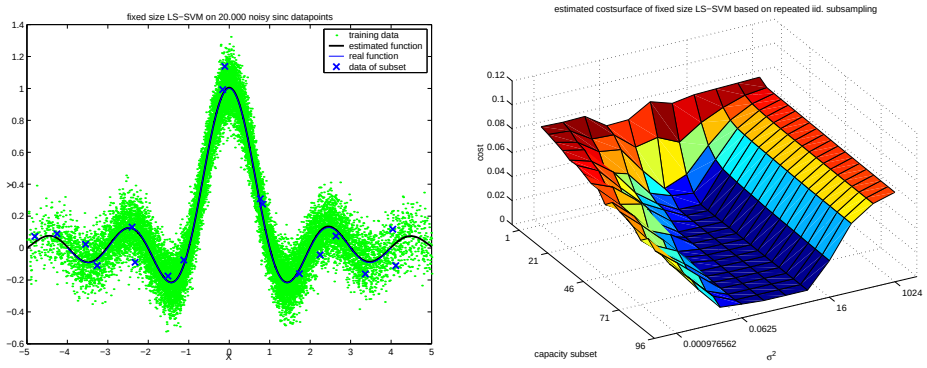
3.2.6 Fixed size LS-SVM

The fixed size LS-SVM is based on two ideas (see also Section 2.4): the first is to exploit the primal-dual formulations of the LS-SVM in view of a Nyström approximation, the second one is to do active support vector selection (here based on entropy criteria). The first step is implemented as follows:

```
>> % X,Y contains the dataset, svX is a subset of X
>> sig2 = 1;
>> features = AFE(svX,'RBF_kernel',sig2, X);
>> [Cl3, gam_optimal] = bay_rr(features,Y,1,3);
>> [W,b] = ridgeregress(features, Y, gam_optimal);
>> Yh = W*features+b;
```

Optimal values for the kernel parameters and the capacity of the fixed size LS-SVM can be obtained using a simple Monte Carlo experiment. For different kernel parameters and capacities (number of chosen support vectors), the performance on random subsets of support vectors are evaluated. The means of the performances are minimized by an exhaustive search (Figure 3.9b):

```
>> caps = [10 20 50 100 200]
>> sig2s = [.1 .2 .5 1 2 4 10]
>> nb = 10;
>> for i=1:length(caps),
     for j=1:length(sig2s),
       for t = 1:nb,
         sel = randperm(size(X,1));
         svX = X(sel(1:caps(i)));
         features = AFE(svX,'RBF_kernel',sig2s(j), X);
         [Cl3, gam_optimal] = bay_rr(features,Y,1,3);
         [W,b, Yh] = ridgeregress(features, Y, gam_opt);
         performances(t) = mse(Y - Yh);
       end
       minimal_performances(i,j) = mean(performances);
     end
```



(a) (b)

Figure 3.9: *Illustration of fixed size LS-SVM on a noisy sinc function with 20000 data points:* **(a)** *fixed size LS-SVM selects a subset of the data after Nyström approximation. The regularization parameter for the regression in the primal space is optimized here using the Bayesian framework;* **(b)** *Estimated cost surface of the fixed size LS-SVM based on random subsamples of the data, of different subset capacities and kernel parameters.*

```
   end
```

The kernel parameter and capacity corresponding to a good performance are searched:

```
>> [minp,ic] = min(minimal_performances,[],1);
>> [minminp,is] = min(minp);
>> capacity = caps(ic);
>> sig2 =     sig2s(is);
```

The following approach optimizes the selection of support vectors according to the quadratic Renyi entropy:

```
>> % load data X and Y, 'capacity' and the kernel parameter 'sig2'
>> sv = 1:capacity;
>> max_c = -inf;
>> for i=1:size(X,1),
     replace = ceil(rand.*capacity);
     subset = [sv([1:replace-1 replace+1:end]) i];
     crit = kentropy(X(subset,:),'RBF_kernel',sig2);
     if max_c <= crit, max_c = crit; sv = subset; end
   end
```

This selected subset of support vectors is used to construct the final model (Figure 3.9a):

```
>> features = AFE(svX,'RBF_kernel',sig2, X);
>> [Cl3, gam_optimal] = bay_rr(features,Y,1,3);
>> [W,b, Yh] = ridgeregress(features, Y, gam_opt);
```

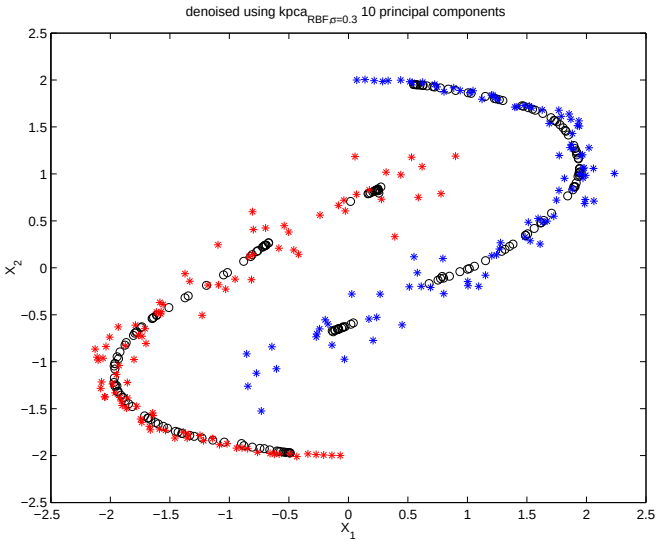


Figure 3.10: De-noised data ('o') obtained by reconstructing the data-points ('*') using the first principal components of kernel PCA.

3.3 Unsupervised Learning using kernel based Principal Component Analysis

A simple example shows the idea of denoising in input space by means of PCA in feature space. The model is optimized to have a minimal reconstruction error [12]. The eigenvectors corresponding to the two largest eigenvalues in this problem represent the two bows on Figure 3.10.

```
>> % load dataset in X...
>> sig2 = 1;
>> [eigval, scores] = kpca(X, 'RBF_kernel',sig2);
>> Xd = denoise_kpca(X,'RBF_kernel',sig2, nb);
```

Appendix A

MATLAB functions

A.1 General Notation

In the full syntax description of the function calls, a star (*) indicates that the argument is optional. In the description of the arguments, a (*) denotes the default value. In this extended help of the function calls of LS-SVMlab, a number of symbols and notations return in the explanation and the examples. These are defined as follows:

Variables	**Explanation**
`d`	Dimension of the input data
`m`	Dimension of the output data
`N`	Number of training data
`Nt`	Number of test data
`nb`	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation
`X`	`N`×`d` matrix with the inputs of the training data
`Xt`	`Nt`×`d` matrix with the inputs of the test data
`Y`	`N`×`m` matrix with the outputs of the training data
`Yt`	`Nt`×`m` matrix with the outputs of the test data
`Zt`	`Nt`×`m` matrix with the predicted latent variables of a classifier

This toolbox supports a classical functional interface as well as an object oriented interface. The latter has a few dedicated structures which will appear many times:

Structures	**Explanation**
`bay`	Object oriented representation of the results of the Bayesian inference
`model`	Object oriented representation of the LS-SVM model

A.2 Index of Function Calls

A.2.1 Training and Simulation

Function Call	Short Explanation	Reference
`latentlssvm`	Calculate the latent variables of the LS-SVM classifier	A.3.18
`plotlssvm`	Plot the LS-SVM results in the environment of the training data	A.3.23
`simlssvm`	Evaluate the LS-SVM at the given points	A.3.29
`trainlssvm`	Find the support values and the bias term of a Least Squares Support Vector Machine	A.3.31

A.2.2 Object Oriented Interface

This toolbox supports a classical functional interface as well as an object oriented interface. The latter has a few dedicated functions. This interface is recommended for the more experienced user.

Function Call	Short Explanation	Reference
`changelssvm`	Change properties of an LS-SVM object	A.3.14
`demomodel`	Demo introducing the use of the compact calls based on the model structure	
`initlssvm`	Initiate the LS-SVM object before training	A.3.14

A.2.3 Training and Simulating Functions

Function Call	Short Explanation	Reference
`lssvm.mex*`	MATLAB CMEX linked C-interface for training in MATLAB for UNIX/LINUX	-
`lssvm.dll`	MATLAB CMEX linked C-interface for training in MATLAB for windows	-
`lssvmFILE.m`	MATLAB code for file interfaced C-coded executable	-
`lssvmFILE.x/exe`	C-coded executable for training UNIX/ Windows	-
`lssvmMATLAB.m`	MATLAB implementation of training	-
`prelssvm`	Internally called preprocessor	A.3.25
`postlssvm`	Internally called postprocessor	A.3.25
`simclssvm.dll`	MATLAB CMEX linked C-interface for training in MATLAB for Windows	-
`simclssvm.mex*`	MATLAB CMEX linked C-interface for training in MATLAB for UNIX/LINUX	-
`simFILE.x/exe`	C-coded executable for training in MATLAB for UNIX/Windows	-

A.2.4 Kernel Functions

Function Call	Short Explanation	Reference
`lin_kernel`	Linear kernel for MATLAB implementation	A.3.21
`MLP_kernel`	Multilayer Perceptron kernel for MATLAB implementation	A.3.21
`poly_kernel`	Polynomial kernel for MATLAB implementation	A.3.21
`RBF_kernel`	Radial Basis Function kernel for MATLAB implementation	A.3.21

A.2.5 Tuning, Sparseness and Robustness

Function Call	Short Explanation	Reference
`crossvalidate`	Estimate the model performance with L-fold crossvalidation	A.3.10
`gridsearch`	A two-dimensional minimization procedure based on exhaustive search in a limited range	A.3.32
`leaveoneout`	Estimate the model performance with leave-one-out crossvalidation	A.3.19
`leaveoneout_lssvm`	Fast leave-one-out cross-validation for the LS-SVM based on one full matrix inversion	A.3.20
`mae, medae`	L_1 cost measures of the residuals	A.3.22
`minf, misclass`	L_∞ and L_0 cost measures of the residuals	A.3.22
`mse, trimmedmse`	L_2 cost measures of the residuals	A.3.22
`sparselssvm`	Remove iteratively the least relevant support vectors to obtain sparsity	A.3.30
`tunelssvm`	Tune the hyperparameters of the model with respect to the given performance measure	A.3.32
`robustlssvm`	Robust training in the case of non-Gaussian noise or outliers	A.3.27
`validate`	Validate a trained model on a fixed validation set	A.3.33

A.2.6 Classification Extensions

Function Call	Short Explanation	Reference
`code`	Encode and decode a multi-class classification task to multiple binary classifiers	A.3.9
`code_ECOC`	Error correcting output coding	A.3.9
`code_MOC`	Minimum Output Encoding	A.3.9
`code_OneVsAll`	One versus All encoding	A.3.9
`code_OneVsOne`	One versus One encoding	A.3.9
`codedist_hamming`	Hamming distance measure between two encoded class labels	A.3.9
`codelssvm`	Encoding the LS-SVM model	A.3.9
`deltablssvm`	Bias term correction for the LS-SVM classificatier	A.3.11
`roc`	Receiver Operating Characteristic curve of a binary classifier	A.3.28

A.2.7 Bayesian Framework

Function Call	Short Explanation	Reference
bay_errorbar	Compute the error bars for a one dimensional regression problem	A.3.2
bay_initlssvm	Initialize the hyperparameters for Bayesian inference	A.3.3
bay_lssvm	Compute the posterior cost for the different levels in Bayesian inference	A.3.4
bay_lssvmARD	Automatic Relevance Determination of the inputs of the LS-SVM	A.3.5
bay_modoutClass	Estimate the posterior class probabilities of a binary classifier using Bayesian inference	A.3.6
bay_optimize	Optimize model- or hyperparameters with respect to the different inference levels	A.3.7
bay_rr	Bayesian inference for linear ridge regression	A.3.8
eign	Find the principal eigenvalues and eigenvectors of a matrix with Nyström's low rank approximation method	A.3.13
kernel_matrix	Construct the positive (semi-) definite kernel matrix	A.3.16
kpca	Kernel Principal Component Analysis	A.3.17
ridgeregress	Linear ridge regression	A.3.26

A.2.8 NARX models and Prediction

Function Call	Short Explanation	Reference
`predict`	Iterative prediction of a trained LS-SVM NARX model (in recurrent mode)	A.3.24
`windowize`	Rearrange the data points into a Hankel matrix for (N)AR time-series modeling	A.3.34
`windowize_NARX`	Rearrange the input and output data into a (block) Hankel matrix for (N)AR(X) time-series modeling	A.3.34

A.2.9 Unsupervised learning

Function Call	Short Explanation	Reference
`AFE`	Automatic Feature Extraction from Nyström method	A.3.1
`denoise_kpca`	Reconstruct the data mapped on the principal components	A.3.12
`kentropy`	Quadratic Renyi Entropy for a kernel based estimator	A.3.15
`kpca`	Compute the nonlinear kernel principal components of the data	A.3.17

A.2.10 Fixed Size LS-SVM

The idea of fixed size LS-SVM is still under development. However, in order to enable the user to explore this technique a number of related functions are included in the toolbox. A demo illustrates how to combine these in order to build a fixed size LS-SVM.

Function Call	Short Explanation	Reference
`AFE`	Automatic Feature Extraction from Nyström method	A.3.1
`bay_rr`	Bayesian inference of the cost on the 3 levels of linear ridge regression	A.3.8
`demo_fixedsize`	Demo illustrating the use of LS-SVMlab for handling large datasets	-
`kentropy`	Quadratic Renyi Entropy for a kernel based estimator	A.3.15
`ridgeregress`	Linear ridge regression	A.3.26

A.2.11 Demos

name of the demo	Short Explanation
`demo_fixedsize`	Demo illustrating the use of LS-SVMlab for handling large datasets
`demo_yinyang`	Demo illustrating the possibilities of unsupervised learning by kernel PCA
`democlass`	Simple demo illustrating the use of LS-SVMlab for classification
`demofun`	Simple demo illustrating the use of LS-SVMlab for regression
`demomodel`	Simple demo illustrating the use of the object oriented interface of LS-SVMlab

A.3 Alphabetical List of Function Calls

A.3.1 AFE

Purpose

Automatic Feature Extraction by Nyström method

Basic syntax

```
>> features = AFE(X, kernel, sig2, Xt)
```

Description

Using the *Nyström* approximation method, the mapping of data to the feature space can be evaluated explicitly. This gives the features that one can use for a linear regression or classification. The decomposition of the mapping to the feature space relies on the eigenvalue decomposition of the kernel matrix. The Matlab (`'eigs'`) or Nyström's (`'eign'`) approximation using the `nb` most important eigenvectors/eigenvalues can be used. The eigenvalue decomposition is not re-calculated if it is passed as an extra argument. This routine internally calls a cmex file.

Full syntax

```
>> [features, U, lam] = AFE(X, kernel, sig2, Xt, type, nb)
>>  features          = AFE(X, kernel, sig2, Xt, [],[], U, lam)
```

Outputs

`features`	`Nt`×`nb` matrix with extracted features
`U`(*)	`N`×`nb` matrix with eigenvectors
`lam`(*)	`nb`×`1` vector with eigenvalues

Inputs

`X`	`N`×`d` matrix with input data
`kernel`	Name of the used kernel (e.g. `'RBF_kernel'`)
`sig2`	Parameter of the used kernel
`Xt`	`Nt`×`d` data from which the features are extracted
`type`(*)	`'eig'`(*), `'eigs'` or `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation
`U`(*)	`N`×`nb` matrix with eigenvectors
`lam`(*)	`nb*1` vector with eigenvalues

See also:

`kernel_matrix, RBF_kernel, demo_fixedsize`

A.3.2 bay_errorbar

Purpose

Compute the error bars for a one dimensional regression problem

Basic syntax

```
>> sig_e = bay_errorbar({X,Y,'function',gam,sig2}, Xt)
>> sig_e = bay_errorbar(model, Xt)
```

Description

The computation takes into account the estimated noise variance and the uncertainty of the model parameters, estimated by Bayesian inference. `sig_e` is the estimated standard deviation of the error bars of the points `Xt`. A plot is obtained by replacing `Xt` by the string `'figure'`.

Full syntax

- Using the functional interface:

```
>> sig_e = bay_errorbar({X,Y,'function',gam,sig2,kernel,preprocess}, ...
                        Xt, type, nb)
>> sig_e = bay_errorbar({X,Y,'function',gam,sig2,kernel,preprocess}, ...
                        'figure', type, nb)
```

Outputs	
`sig_e`	`Nt`×`1` vector with the σ^2 errorbands of the test data
Inputs	
`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the inputs of the training data
`type`	`'function estimation'` (`'f'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`Xt`	`Nt`×`d` matrix with the inputs of the test data
`type`(*)	`'svd'`(*), `'eig'`, `'eigs'` or `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

- Using the object oriented interface:

```
>> [sig_e, bay, model] = bay_errorbar(model, Xt,       type, nb)
>> [sig_e, bay, model] = bay_errorbar(model, 'figure', type, nb)
```

Outputs

`sig_e`	`Nt`×`1` vector with the σ^2 errorbands of the test data
`model`(*)	Object oriented representation of the LS-SVM model
`bay`(*)	Object oriented representation of the results of the Bayesian inference

Inputs

`model`	Object oriented representation of the LS-SVM model
`Xt`	`Nt`×`d` matrix with the inputs of the test data
`type`(*)	`'svd'`(*), `'eig'`, `'eigs'` or `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

See also:

`bay_lssvm, bay_optimize, bay_modoutClass, plotlssvm`

A.3.3 bay_initlssvm

Purpose

Initialize the hyperparameters γ and σ^2 before optimization with `bay_optimize`

Basic syntax

```
>> [gam, sig2] = bay_initlssvm({X,Y,type,[],[]})
>> model       = bay_initlssvm(model)
```

Description

A starting value for σ^2 is only given if the model has kernel type `'RBF_kernel'`.

Full syntax

- Using the functional interface:

  ```
  >> [gam, sig2] = bay_initlssvm({X,Y,type,[],[],kernel})
  ```

  Outputs

`gam`	Proposed initial regularization parameter
`sig2`	Proposed initial `'RBF_kernel'` parameter

  Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)

- Using the object oriented interface:

  ```
  >> model = bay_initlssvm(model)
  ```

  Outputs

`model`	Object oriented representation of the LS-SVM model with initial hyperparameters

  Inputs

`model`	Object oriented representation of the LS-SVM model

See also:

`bay_lssvm, bay_optimize`

A.3.4 bay_lssvm

Purpose

Compute the posterior cost for the 3 levels in Bayesian inference

Basic syntax

```
>> cost = bay_lssvm({X,Y,type,gam,sig2}, level, type)
>> cost = bay_lssvm(model              , level, type)
```

Description

Estimate the posterior probabilities of model (hyper-) parameters on the different inference levels. By taking the negative logarithm of the posterior and neglecting all constants, one obtains the corresponding cost.

Computation is only feasible for one dimensional output regression and binary classification problems. Each level has its different in- and output syntax:

- **First level:** The cost associated with the posterior of the model parameters (support values and bias term) is determined. The type can be:
  - `'train'`: do a training of the support values using `trainlssvm`. The total cost, the cost of the residuals (`Ed`) and the regularization parameter (`Ew`) are determined by the solution of the support values
  - `'retrain'`: do a retraining of the support values using `trainlssvm`
  - the cost terms can also be calculated from an (approximate) eigenvalue decomposition of the kernel matrix: `'svd'`, `'eig'`, `'eigs'` or Nyström's `'eign'`
- **Second level:** The cost associated with the posterior of the regularization parameter is computed. The type can be `'svd'`, `'eig'`, `'eigs'` or Nyström's `'eign'`.
- **Third level:** The cost associated with the posterior of the chosen kernel and kernel parameters is computed. The type can be: `'svd'`, `'eig'`, `'eigs'` or Nyström's `'eign'`.

Full syntax

- **Outputs on the first level**

  ```
  >> [costL1,Ed,Ew,bay] = bay_lssvm({X,Y,type,gam,sig2,kernel,preprocess}, 1, type, nb)
  >> [costL1,Ed,Ew,bay] = bay_lssvm(model, 1, type, nb)
  ```

  With

`costL1`	Cost proportional to the posterior
`Ed`(*)	Cost of the fitting error term
`Ew`(*)	Cost of the regularization parameter
`bay`(*)	Object oriented representation of the results of the Bayesian inference

- **Outputs on the second level**

  ```
  >> [costL2,DcostL2, optimal_cost, bay] = ...
          bay_lssvm({X,Y,type,gam,sig2,kernel,preprocess}, 2, type, nb)
  >> [costL2,DcostL2, optimal_cost, bay] = bay_lssvm(model, 2, type, nb)
  ```

With

`costL2`	Cost proportional to the posterior on the second level
`DcostL2(*)`	Derivative of the cost
`optimal_cost(*)`	Optimality of the regularization parameter (optimal = 0)
`bay(*)`	Object oriented representation of the results of the Bayesian inference

- **Outputs on the third level**

```
>> [costL3,bay] = bay_lssvm({X,Y,type,gam,sig2,kernel,preprocess}, 3, type, nb)
>> [costL3,bay] = bay_lssvm(model, 3, type, nb)
```

With

`costL3`	Cost proportional to the posterior on the third level
`bay(*)`	Object oriented representation of the results of the Bayesian inference

- **Inputs using the functional interface**

```
>> bay_lssvm({X,Y,type,gam,sig2,kernel,preprocess}, level, type, nb)
```

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel(*)`	Kernel type (by default `'RBF_kernel'`)
`preprocess(*)`	`'preprocess'`(*) or `'original'`
`level`	1, 2, 3
`type(*)`	`'svd'`(*), `'eig'`, `'eigs'`, `'eign'`
`nb(*)`	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

- **Inputs using the object oriented interface**

```
>> bay_lssvm(model, level, type, nb)
```

`model`	Object oriented representation of the LS-SVM model
`level`	1, 2, 3
`type(*)`	`'svd'`(*), `'eig'`, `'eigs'`, `'eign'`
`nb(*)`	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

See also:

`bay_lssvmARD, bay_optimize, bay_modoutClass, bay_errorbar`

A.3.5 bay_lssvmARD

Purpose

Bayesian Automatic Relevance Determination of the inputs of an LS-SVM

Basic syntax

```
>> dimensions = bay_lssvmARD({X,Y,type,gam,sig2})
>> dimensions = bay_lssvmARD(model)
```

Description

For a given problem, one can determine the most relevant inputs for the LS-SVM within the Bayesian evidence framework. To do so, one assigns a different weighting parameter to each dimension in the kernel and optimizes this using the third level of inference. According to the used kernel, one can remove inputs corresponding the larger or smaller kernel parameters. This routine *only* works with the `'RBF_kernel'` with a `sig2` per input. In each step, the input with the largest optimal `sig2` is removed (backward selection). For every step, the generalization performance is approximated by the cost associated with the third level of Bayesian inference.

The ARD is based on backward selection of the inputs based on the `sig2s` corresponding in each step with a minimal cost criterion. Minimizing this criterion can be done by `'continuous'` or by `'discrete'`. The former uses in each step continuous varying kernel parameter optimization, the latter decides which one to remove in each step by binary variables for each component (this can only applied for rather low dimensional inputs as the number of possible combinations grows exponentially with the number of inputs). If working with the `'RBF_kernel'`, the kernel parameter is rescaled appropriately after removing an input variable.

The computation of the Bayesian cost criterion can be based on the singular value decomposition `'svd'` of the full kernel matrix or by an approximation of these eigenvalues and vectors by the `'eigs'` or `'eign'` approximation based on `'nb'` data points.

Full syntax

- Using the functional interface:

```
>>  [dimensions, ordered, costs, sig2s] = ...
       bay_lssvmARD({X,Y,type,gam,sig2,kernel,preprocess}, method, type, nb)
```

Outputs

`dimensions`	`r`×`1` vector of the relevant inputs
`ordered`(*)	`d`×`1` vector with inputs in decreasing order of relevance
`costs`(*)	Costs associated with third level of inference in every selection step
`sig2s`(*)	Optimal kernel parameters in each selection step

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`method`(*)	`'discrete'`(*) or `'continuous'`
`type`(*)	`'svd'`(*), `'eig'`, `'eigs'`, `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

- Using the object oriented interface:

```
>>  [dimensions, ordered, costs, sig2s, model] = bay_lssvmARD(model, method, type, nb)
```

Outputs

`dimensions`	`r`×`1` vector of the relevant inputs
`ordered`(*)	`d`×`1` vector with inputs in decreasing order of relevance
`costs`(*)	Costs associated with third level of inference in every selection step
`sig2s`(*)	Optimal kernel parameters in each selection step
`model`(*)	Object oriented representation of the LS-SVM model trained only on the relevant inputs

Inputs

`model`	Object oriented representation of the LS-SVM model
`method`(*)	`'discrete'`(*) or `'continuous'`
`type`(*)	`'svd'`(*), `'eig'`, `'eigs'`, `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

See also:

`bay_lssvm, bay_optimize, bay_modoutClass, bay_errorbar`

A.3.6 bay_modoutClass

Purpose

Estimate the posterior class probabilities of a binary classifier using Bayesian inference

Basic syntax

```
>> [Ppos, Pneg] = bay_modoutClass({X,Y,'classifier',gam,sig2}, Xt)
>> [Ppos, Pneg] = bay_modoutClass(model, Xt)
```

Description

Calculate the probability that a point will belong to the positive or negative classes taking into account the uncertainty of the parameters. Optionally, one can express prior knowledge as a probability between 0 and 1, where prior equal to 2/3 means that the prior positive class probability is 2/3 (more likely to occur than the negative class).

For binary classification tasks with a 2 dimensional input space, one can make a surface plot by replacing `Xt` by the string `'figure'`.

Full syntax

- Using the functional interface:

```
>> [Ppos, Pneg] = bay_modoutClass({X,Y,'classifier',...
                        gam,sig2, kernel, preprocess}, Xt, prior)
>>  bay_modoutClass({X,Y,'classifier',...
                        gam,sig2, kernel, preprocess}, 'figure', prior)
```

Outputs

`Ppos`	`Nt`×`1` vector with probabilities that testdata `Xt` belong to the positive class
`Pneg`	`Nt`×`1` vector with probabilities that testdata `Xt` belong to the negative(zero) class

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`Xt`(*)	`Nt`×`d` matrix with the inputs of the test data
`prior`(*)	Prior knowledge of the balancing of the training data (or `[]`)
`type`(*)	`'svd'`(*), `'eig'`, `'eigs'` or `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

- Using the object oriented interface:

```
>> [Ppos, Pneg, bay, model] = bay_modoutClass(model, Xt, prior, type, nb)
>>                            bay_modoutClass(model, 'figure', prior, type, nb)
```

Outputs

`Ppos`	`Nt×1` vector with probabilities that testdata `Xt` belong to the positive class
`Pneg`	`Nt×1` vector with probabilities that testdata `Xt` belong to the negative(zero) class
`bay`(*)	Object oriented representation of the results of the Bayesian inference
`model`(*)	Object oriented representation of the LS-SVM model

Inputs

`model`	Object oriented representation of the LS-SVM model
`Xt`(*)	`Nt×d` matrix with the inputs of the test data
`prior`(*)	Prior knowledge of the balancing of the training data (or `[]`)
`type`(*)	`'svd'`(*), `'eig'`, `'eigs'` or `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

See also:

`bay_lssvm, bay_optimize, bay_errorbar, ROC`

A.3.7 bay_optimize

Purpose

Optimize the posterior probabilities of model (hyper-) parameters with respect to the different levels in Bayesian inference

Basic syntax

One can optimize on the three different inference levels as described in section 2.1.3.

- **First level:** In the first level one optimizes the support values α's and the bias `b`.
- **Second level:** In the second level one optimizes the regularization parameter `gam`.
- **Third level:** In the third level one optimizes the kernel parameter. In the case of the common `'RBF_kernel'` the kernel parameter is the bandwidth `sig2`.

This routine is only tested with Matlab version 6 using the corresponding optimization toolbox.

Full syntax

- **Outputs on the first level:**

```
>> [model, alpha, b] = bay_optimize({X,Y,type,gam,sig2,kernel,preprocess}, 1, type, nb)
>> [model, alpha, b] = bay_optimize(model, 1, type, nb)
```

With

`model`	Object oriented representation of the LS-SVM model optimized on the first level
`alpha`(*)	Support values optimized on the first level of inference
`b`(*)	Bias term optimized on the first level of inference

- **Outputs on the second level:**

```
>> [model,gam] = bay_optimize({X,Y,type,gam,sig2,kernel,preprocess}, 2, type, nb)
>> [model,gam] = bay_optimize(model, 2, type, nb)
```

With

`model`	Object oriented representation of the LS-SVM model optimized on the second level of inference
`gam`(*)	Regularization parameter optimized on the second level of inference

- **Outputs on the third level:**

```
>> [model, sig2] = bay_optimize({X,Y,type,gam,sig2,kernel,preprocess}, level, type, nb)
```

With

`model`	Object oriented representation of the LS-SVM model optimized on the third level of inference
`sig2`(*)	Kernel parameter optimized on the third level of inference

- **Inputs using the functional interface**

```
>> model = bay_optimize({X,Y,type,gam,sig2,kernel,preprocess}, level, type, nb)
```

X	N×d matrix with the inputs of the training data
Y	N×1 vector with the outputs of the training data
type	'function estimation' ('f') or 'classifier' ('c')
gam	Regularization parameter
sig2	Kernel parameter (bandwidth in the case of the 'RBF_kernel')
kernel(*)	Kernel type (by default 'RBF_kernel')
preprocess(*)	'preprocess'(*) or 'original'
level	1, 2, 3
type(*)	'eig', 'svd'(*), 'eigs', 'eign'
nb(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

- **Inputs using the object oriented interface**

```
>> model = bay_optimize(model, level, type, nb)
```

model	Object oriented representation of the LS-SVM model
level	1, 2, 3
type(*)	'eig', 'svd'(*), 'eigs', 'eign'
nb(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

See also:

bay_lssvm, bay_lssvmARD, bay_modoutClass, bay_errorbar

A.3.8 bay_rr

Purpose

Bayesian inference of the cost on the three levels of linear ridge regression

Basic syntax

```
>> cost = bay_rr(X, Y, gam, level)
```

Description

This function implements the cost functions related to the Bayesian framework of linear ridge Regression [29]. Optimizing this criteria results in optimal model parameters `W`,`b`, hyperparameters. The criterion can also be used for model comparison.

The obtained model parameters `w` and `b` are optimal on the first level w.r.t `J = 0.5*w'*w+gam*0.5*sum(Y-X*w-b).^2`.

Full syntax

- **Outputs on the first level:** Cost proportional to the posterior of the model parameters.

  ```
  >> [costL1, Ed, Ew] = bay_rr(X, Y, gam, 1)
  ```

  With

`costL1`	Cost proportional to the posterior
`Ed`(*)	Cost of the fitting error term
`Ew`(*)	Cost of the regularization parameter

- **Outputs on the second level:** Cost proportional to the posterior of `gam`.

  ```
  >> [costL2, DcostL2, Deff, mu, ksi, eigval, eigvec] = bay_rr(X, Y, gam, 2)
  ```

  With

`costL2`	Cost proportional to the posterior on the second level
`DcostL2`(*)	Derivative of the cost proportional to the posterior
`Deff`(*)	Effective number of parameters
`mu`(*)	Relative importance of the fitting error term
`ksi`(*)	Relative importance of the regularization parameter
`eigval`(*)	Eigenvalues of the covariance matrix
`eigvec`(*)	Eigenvectors of the covariance matrix

- **Outputs on the third level:** The following commands can be used to compute the level 3 cost function for different models (e.g. models with different selected sets of inputs). The best model can then be chosen as the model with best level 3 cost (`CostL3`).

  ```
  >> [costL3, gam_optimal] = bay_rr(X, Y, gam, 3)
  ```

  With

`costL3`	Cost proportional to the posterior on the third inference level
`gam_optimal`(*)	Optimal regularization parameter obtained from optimizing the second level

- **Inputs:**

```
>> cost = bay_rr(X, Y, gam, level)
```

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`gam`	Regularization parameter
`level`	`1`, `2`, `3`

See also:

`ridgeregress`,`bay_lssvm`

A.3.9 code, codelssvm

Purpose

Encode and decode a multi-class classification task into multiple binary classifiers

Basic syntax

```
>> Yc = code(Y, codebook)
```

Description

The coding is defined by the codebook. The codebook is represented by a matrix where the columns represent all different classes and the rows indicate the result of the binary classifiers. An example is given: the 3 classes with original labels `[1 2 3]` can be encoded in the following codebook (using Minimal Output Encoding):

```
 >> codebook
    = [-1  -1  1;
        1  -1  1]
```

For this codebook, a member of the first class is found if the first binary classifier is negative and the second classifier is positive. A *don't care* is represented by `eps`. By default it is assumed that the original classes are represented as different numerical labels. One can overrule this by passing the `old_codebook` which contains information about the old representation.

A codebook can be created by one of the functions (`codefct`) `code_MOC`, `code_OneVsOne`, `code_OneVsAll`, `code_ECOC`. Additional arguments to this function can be passed as a cell in `codefct_args`.

```
>>  Yc = code(Y,codefct,codefct_args)
>>  Yc = code(Y,codefct,codefct_args, old_codebook)
>> [Yc, codebook, oldcodebook] = code(Y,codefct,codefct_args)
```

To detect the classes of a disturbed encoded signal given the corresponding codebook, one needs a distance function (`fctdist`) with optional arguments given as a cell (`fctdist_args`). By default, the Hamming distance (of function `codedist_hamming`) is used.

```
>> Yc = code(Y, codefct, codefct_args, old_codebook, fctdist, fctdist_args)
```

A simple example is given here, a more elaborated example is given in section 3.1.4. Here, a short categorical signal `Y` is encoded in `Yec` using Minimal Output Encoding and decoded again to its original form:

```
>> Y = [1; 2; 3; 2; 1]
>> [Yc,codebook,old_codebook] = code(Y,'code_MOC')       % encode
>> Yc
   = [-1    -1
      -1     1
       1    -1
      -1     1
      -1    -1]

>> codebook
   = [ -1    -1     1
       -1     1    -1]

>> old_codebook
   = [1     2     3]
```

```
>> code(Yc, old_codebook, [], codebook, 'codedist_hamming') % decode
   ans
   = [1; 2; 3; 2; 1]
```

Different encoding schemes are available:

- **Minimum Output Coding** (`code_MOC`)
  Here the minimal number of bits n_b is used to encode the n_c classes:
  $$n_b = \lceil \log_2 n_c \rceil.$$
- **Error Correcting Output Code** (`code_ECOC`)
  This coding scheme uses redundant bits. Typically, one bounds the number of binary classifiers n_b by
  $$n_b \leq 15 \lceil \log_2 n_c \rceil.$$
  However, it is not guaranteed to have a valid n_b-representation of n_c classes for all combinations. This routine based on backtracking can take some memory and time.
- **One versus All Coding** (`code_OneVsAll`)
  Each binary classifier $k = 1, ..., n_c$ is trained to discriminate between class k and the union of the others.
- **One Versus One Coding** (`code_OneVsOns`)
  Each of the n_b binary classifiers is used to discriminate between a specific pair of n_c classes
  $$n_b = \frac{n_c(n_c - 1)}{2}.$$

Different decoding schemes are implemented:

- **Hamming Distance** (`codedist_hamming`)
  This measure equals the number of corresponding bits in the binary result and the codeword. Typically, it is used for the Error Correcting Code.
- **Bayesian Distance Measure** (`codedist_bay`)
  The Bayesian moderated output of the binary classifiers is used to estimate the posterior probability.

Encoding using the previous algorithms of the LS-SVM multi-class classifier can easily be done by `codelssvm`. It will be invoked by `trainlssvm` if an appropriate encoding scheme is defined in a model. An example shows how to use the Bayesian distance measure to extract the estimated class from the simulated encoded signal. Assumed are input and output data `X` and `Y` (size is respectively $N_{train} \times D_{in}$ and $N_{train} \times 1$), a kernel parameter `sig2` and a regularization parameter `gam`. `Yt` corresponding to a set of data points `Xt` (size is $N_{test} \times D_{in}$) is to be estimated:

```
% encode for training
>> model = initlssvm(X, Y, 'classifier', gam, sig2)
>> model = changelssvm(model, 'codetype', 'code_MOC')
>> model = changelssvm(model, 'codedist_fct', 'codedist_hamming')
>> model = codelssvm(model) % implicitly called by next command
>> model = trainlssvm(model)
>> plotlssvm(model);

% decode for simulating
>> model = changelssvm(model, 'codedist_fct', 'codedist_bay')
>> model = changelssvm(model, 'codedist_args',...
                              {bay_modoutClass(model,Xt)})
>>   Yt  = simlssvm(model, Xt)
```

Full syntax

We denote the number of used binary classifiers by `nbits` and the number of different represented classes by `nc`.

- For encoding:

```
>> [Yc, codebook, old_codebook] = code(Y, codefct, codefct_args)
>> Yc = code(Y, given_codebook)
```

Outputs

`Yc`	`N`×`nbits` encoded output classifier
`codebook`(*)	`nbits*nc` matrix representing the used encoding
`old_codebook`(*)	`d*nc` matrix representing the original encoding

Inputs

`Y`	`N`×`d` matrix representing the original classifier
`codefct`(*)	Function to generate a new codebook (e.g. `code_MOC`)
`codefct_args`(*)	Extra arguments for `codefct`
`given_codebook`(*)	`nbits*nc` matrix representing the encoding to use

- For decoding:

```
>> Yd = code(Yc, codebook,[], old_codebook, codedist_fct, codedist_args)
```

Outputs

`Yd`	`N`×`nc` decoded output classifier

Inputs

`Y`	`N`×`d` matrix representing the original classifier
`codebook`	`d*nc` matrix representing the original encoding
`old_codebook`	`bits*nc` matrix representing the encoding of the given classifier
`codedist_fct`	Function to calculate the distance between to encoded classifiers (e.g. `codedist_hamming`)
`codedist_args`(*)	Extra arguments of `codedist_fct`

See also:

`code_ECOC`, `code_MOC`, `code_OneVsAll`, `code_OneVsOne`, `codedist_hamming`

A.3.10 crossvalidate

Purpose

Estimate the model performance of a model with l-fold crossvalidation

Basic syntax

```
>> cost = crossvalidate({Xtrain,Ytrain,type,gam,sig2}, Xval, Yval)
>> cost = crossvalidate(model, Xval, Yval)
```

Description

The data is once permutated randomly, then it is divided into `L` (by default `10`) disjunct sets. In the i-th ($i = 1, ..., l$) iteration, the i-th set is used to estimate the performance ('validation set') of the model trained on the other $l - 1$ sets ('training set'). At last, the l (denoted by `L`) different estimates of the performance are combined (by default by the 'mean'). The assumption is made that the input data are distributed independent and identically over the input space. As additional output, the costs in the different folds ('costs') and all residuals ('ec') of the data are returned:

```
>> [cost, costs, ec] = crossvalidate(model, Xval, Yval)
```

By default, this function will call the training (*trainlssvm*) and simulation (*simlssvm*) algorithms for LS-SVMs. However, one can use the validation function more generically by specifying the appropriate training and simulation function. Some commonly used criteria are:

```
>> cost = crossvalidate(model, Xval, Yval, 10, 'misclass', 'mean', 'corrected')
>> cost = crossvalidate(model, Xval, Yval, 10, 'mse', 'mean', 'original')
>> cost = crossvalidate(model, Xval, Yval, 10, 'mae', 'median', 'corrected')
```

Full syntax

- Using LS-SVMlab with the functional interface:

```
>> [cost, costs, ec] = crossvalidate({X,Y,type,gam,sig2,kernel,preprocess},...
                                      Xval, Yval, L, estfct, combinefct, correction)
```

Outputs

`cost`	Cost estimation of the L-fold cross validation
`costs`(*)	`L`×`1` vector with costs estimated on the L different folds
`ec`(*)	`N`×`1` vector with residuals of all data

Inputs

`X`	Training input data used for defining the LS-SVM and the preprocessing
`Y`	Training output data used for defining the LS-SVM and the preprocessing
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`Xval`	`N`×`d` matrix with the inputs of the data used for cross-validation
`Yval`	`N`×`m` matrix with the outputs of the data used for cross-validation
`L`(*)	Number of folds (by default `10`)
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)
`combinefct`(*)	Function combining the estimated costs on the different folds (by default `mean`)
`correction`(*)	`'original'`(*) or `'corrected'`

- Using the object oriented interface:

```
>> [cost, costs, ec] = crossvalidate(model, Xval, Yval, L, ...
                                      estfct, combinefct, correction)
```

Outputs

`cost`	Cost estimation of the L-fold cross validation
`costs`(*)	`L`×`1` vector with costs estimated on the L different folds
`ec`(*)	`N`×`1` vector with residuals of all data

Inputs

`model`	Object oriented representation of the LS-SVM model
`Xval`	`Nt`×`d` matrix with the inputs of the validation points used in the procedure
`Yval`	`Nt`×`m` matrix with the outputs of the validation points used in the procedure
`L`(*)	Number of folds (by default `10`)
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)
`combinefct`(*)	Function combining the estimated costs on the different folds (by default `mean`)
`correction`(*)	`'original'`(*) or `'corrected'`

- Using other modeling techniques::

```
>> [cost, costs, ec] = crossvalidate(model, Xval, Yval, L,...
                           estfct, combinefct, correction, trainfct, simfct)
```

Outputs

`cost`	Cost estimation of the L-fold cross validation
`costs`(*)	`l`×`1` vector with costs estimated on the L different folds
`ec`(*)	`N`×`1` vector with residuals of all data

Inputs

`model`	Object oriented representation of the model
`Xval`	`Nt`×`d` matrix with the inputs of the validation points used
`Yval`	`Nt`×`m` matrix with the outputs of the validation points used in the procedure
`L`(*)	Number of folds (by default `10`)
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)
`combinefct`(*)	Function combining the estimated costs on the different folds (by default `mean`)
`correction`(*)	`'original'`(*) or `'corrected'`
`trainfct`	Function used to train the model
`simfct`	Function used to simulate test data with the model

See also:

`validate, leaveoneout, leaveoneout_lssvm, trainlssvm, simlssvm`

A.3.11 deltablssvm

Purpose

Bias term correction for the LS-SVM classifier

Basic syntax

```
>> model = deltablssvm(model, b_new)
```

Description

This function is only useful in the object oriented function interface. Set explicitly the bias term `b_new` of the LS-SVM model.

Full syntax

```
>> model = deltablssvm(model, b_new)
```

Outputs

`model`	Object oriented representation of the LS-SVM model with initial hyper-parameters

Inputs

`model`	Object oriented representation of the LS-SVM model
`b_new`	`m`×`1` vector with new bias term(s) for the model

See also:

`roc, trainlssvm, simlssvm, changelssvm`

A.3.12 denoise_kpca

Purpose

Reconstruct the data mapped on the first principal components

Basic syntax

```
>> Xd = denoise_kpca(X, kernel, kernel_par);
```

Description

Denoising can be done by moving the point in inputspace so that its corresponding map to feature space is optimized. This means that the data point in feature space is as close as possible with its corresponding reconstructed points using the principal components. If the principal components are to be calculated on the same data 'X' as the one one wants to denoise, use the command:

```
>> Xd         = denoise_kpca(X, kernel, kernel_par);
>> [Xd,lam,U] = denoise_kpca(X, kernel, kernel_par, [], type, nb);
```

When one wants to denoise data `'Xt'` other than the data used to obtain the principal components:

```
>>  Xd          = denoise_kpca(X, kernel, kernel_par, Xt);
>> [Xd, lam, U] = denoise_kpca(X, kernel, kernel_par, Xt, type, nb);
```

Full syntax

- `>> [Xd, lam, U] = denoise_kpca(X, kernel, kernel_par, Xt, type, nb);`

  Outputs

`Xd`	`N×d` (`Nt×d`) matrix with denoised data `X` (`Xt`)
`lam`(*)	`nb×1` vector with eigenvalues of principal components
`U`(*)	`N×nb` (`Nt×d`) matrix with principal eigenvectors

  Inputs

`X`	`N×d` matrix with data points used for finding the principal components
`kernel`	Kernel type (e.g. `'RBF_kernel'`)
`kernel_par`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`Xt`(*)	`Nt×d` matrix with the points to denoise (if not specified, `X` is denoised instead)
`type`(*)	`'eig'`(*), `'svd'`, `'eigs'`, `'eign'`
`nb`(*)	Number of principal components used in approximation

- `>> Xd  = denoise_kpca(X, U, lam, kernel, kernel_par, Xt);`

  Outputs

`Xd`	`N×d` (`Nt×d`) matrix with denoised data `X` (`Xt`)

  Inputs

`X`	`N×d` matrix with data points used for finding the principal components
`U`	`N×nb` (`Nt×d`) matrix with principal eigenvectors
`lam`	`nb×1` vector with eigenvalues of principal components
`kernel`	Kernel type (e.g. `'RBF_kernel'`)
`kernel_par`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`Xt`(*)	`Nt×d` matrix with the points to denoise (if not specified, `X` is denoised instead)

See also:

`kpca, kernel_matrix, RBF_kernel`

A.3.13 eign

Purpose

Find the principal eigenvalues and eigenvectors of a matrix with Nyström's low rank approximation method

Basic syntax

```
>>  D     = eign(A, nb)
>> [V, D] = eign(A, nb)
```

Description

In the case of using this method for low rank approximation and decomposing the kernel matrix, one can call the function without explicit construction of the matrix A.

```
>>  D     = eign(X, kernel, kernel_par, nb)
>> [V, D] = eign(X, kernel, kernel_par, nb)
```

Full syntax

We denote the size of positive definite matrix A with a*a.

- Given the full matrix:

  ```
  >>  D    = eign(A,nb)
  >> [V,D] = eign(A,nb)
  ```

  Outputs

V(*)	a×nb matrix with estimated principal eigenvectors of A
D	nb×1 vector with principal estimated eigenvalues of A

  Inputs

A	a*a positive definite symmetric matrix
nb(*)	Number of approximated principal eigenvalues/eigenvectors

- Given the function to calculate the matrix elements:

  ```
  >>  D = eign(X, kernel, kernel_par, n)
  >> [V,D] = eign(X, kernel, kernel_par, n)
  ```

  Outputs

V(*)	a×nb matrix with estimated principal eigenvectors of A
D	nb×1 vector with estimated principal eigenvalues of A

  Inputs

X	N×d matrix with the training data
kernel	Kernel type (e.g. 'RBF_kernel')
kernel_par	Kernel parameter (bandwidth in the case of the 'RBF_kernel')
nb(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

See also:

eig, eigs, kpca, bay_lssvm

A.3.14 `initlssvm, changelssvm`

Purpose

Only for use with the object oriented model interface

Description

The Matlab toolbox interface is organized in two equivalent ways. In the *functional way*, function calls need explicit input and output arguments. An advantage is their similarity with the mathematical equations.

An alternative syntax is based on the concept of a *model*, gathering all the relevant signals, parameters and algorithm choices. The model is initialized by `model=initlssvm(...)`, or will be initiated implicitly by passing the arguments of `initlssvm(...)` in one cell as the argument of the LS-SVM specific functions, e.g. for training:

```
>> model = trainlssvm({X,Y,type,gam,sig2})
...
>> model = changelssvm(model,'field','value')
```

After training, the model contains the solution of the training including the used default values. All contents of the **model** can be requested (`model.<contenttype>`) or changed (`changelssvm`) each moment. The user is advised not to change the fields of the model by `model.<field>=<value>` as the toolbox cannot guarantee consistency anymore in this way.

The different options are given in following table:

- General options representing the kind of model:

```
          type: 'classifier' ,'function estimation'
implementation: 'CMEX' ,'CFILE' ,'MATLAB'
        status: Status of this model ('trained'  or 'changed' )
         alpha: Support values of the trained LS-SVM model
             b: Bias term of the trained LS-SVM model
      duration: Number of seconds the training lasts
        latent: Returning latent variables ('no' ,'yes' )
      x_delays: Number of delays of eXogeneous variables (by default 0 )
      y_delays: Number of delays of responses (by default 0 )
         steps: Number of steps to predict (by default 1 )
           gam: Regularisation parameter
   kernel_type: Kernel function
   kernel_pars: Extra parameters of the kernel function
```

- Fields used to specify the used training data:

```
    x_dim: Dimension of input space
    y_dim: Dimension of responses
  nb_data: Number of training data
   xtrain: (preprocessed) inputs of training data
   ytrain: (preprocessed,coded) outputs of training data
 selector: Indexes of training data effectively used during training
```

- Options used in the Conjugate Gradient (CG) algorithm:

```
    cga_max_itr: Maximum number of iterations in CG
        cga_eps: Stopcriterium of CG, largest allowed error
   cga_fi_bound: Stopcriterium of CG, smallest allowed improvement
       cga_show: Show the results of the CG algorithm (1 or 0)
cga_startvalues: Starting values of the CG algorithm
```

- Fields with the information for pre- and post-processing (only given if appropriate):

```
 preprocess: 'preprocess'  or 'original'
    schemed: Status of the preprocessing
             ('coded' ,'original'  or 'schemed' )
pre_xscheme: Scheme used for preprocessing the input data
pre_yscheme: Scheme used for preprocessing the output data
  pre_xmean: Mean of the input data
   pre_xstd: Standard deviation of the input data
  pre_ymean: Mean of the responses
   pre_ystd: Standard deviation of the reponses
```

- The specifications of the used encoding (only given if appropriate):

```
         code: Status of the coding
               ('original' ,'changed'  or 'encoded')
     codetype: Used function for constructing the encoding
               for multiclass classification (by default 'none')
codetype_args: Arguments of the codetype function
 codedist_fct: Function used to calculate to which class a
               coded result belongs
codedist_args: Arguments of the codedist function
    codebook2: Codebook of the new coding
    codebook1: Codebook of the original coding
```

Full syntax

- `>> model = initlssvm(X, Y, type, gam, sig2, kernel, preprocess)`

Outputs	
`model`	Object oriented representation of the LS-SVM model
Inputs	
`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'` (by default)

- `>> model = changelssvm(model, field, value)`

Outputs

`model`(*)	Obtained object oriented representation of the LS-SVM model

Inputs

`model`	Original object oriented representation of the LS-SVM model
`field`	Field of the model one wants to change (e.g. `'preprocess'`)
`value`	New value of the field of the model one wants to change

See also:

`trainlssvm, initlssvm, simlssvm, plotlssvm`.

A.3.15 kentropy

Purpose

Quadratic Renyi Entropy for a kernel based estimator

Basic syntax

Given the eigenvectors and the eigenvalues of the kernel matrix, the entropy is computed by

```
>> H = kentropy(X, U, lam)
```

The eigenvalue decomposition can also be computed (or approximated) implicitly:

```
>> H = kentropy(X, kernel, sig2)
```

Full syntax

- `>> H = kentropy(X, kernel, kernel_par, type, nb)`

  Outputs

`H`	Quadratic Renyi entropy of the kernel matrix

  Inputs

`X`	`N`×`d` matrix with the training data
`kernel`	Kernel type (e.g. `'RBF_kernel'`)
`kernel_par`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`type`(*)	`'eig'`(*), `'eigs'`, `'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation

- `>> H = kentropy(X, U, lam)`

  Outputs

`H`	Quadratic Renyi entropy of the kernel matrix

  Inputs

`X`	`N`×`d` matrix with the training data
`U`	`N`×`nb` matrix with principal eigenvectors
`lam`	`nb`×`1` vector with eigenvalues of principal components

See also:

`kernel_matrix, demo_fixedsize, RBF_kernel`

A.3.16 kernel_matrix

Purpose

Construct the positive (semi-) definite and symmetric kernel matrix

Basic Syntax

```
>> Omega = kernel_matrix(X, kernel_fct, sig2)
```

Description

This matrix should be positive definite if the kernel function satisfies the Mercer condition. Construct the kernel values for all test data points in the rows of `Xt`, relative to the points of `X`.

```
>> Omega_Xt = kernel_matrix(X, kernel_fct, sig2, Xt)
```

Full syntax

```
>> Omega = kernel_matrix(X, kernel_fct, sig2, Xt)
```

Outputs

`Omega`	`N`×`N` (`N`×`Nt`) kernel matrix

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`kernel`	Kernel type (by default `'RBF_kernel'`)
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`Xt`(*)	`Nt`×`d` matrix with the inputs of the test data

See also:

`RBF_kernel, lin_kernel, kpca, trainlssvm`

A.3.17 kpca

Purpose

Kernel Principal Component Analysis (KPCA)

Basic syntax

```
>> [eigval, scores] = kpca(X, kernel_fct, sig2)
```

Description

Compute the `nb` largest eigenvalues and the corresponding rescaled eigenvectors corresponding with the principal components in the feature space of the centered kernel matrix. To calculate the eigenvalue decomposition of this $N \times N$ matrix, Matlab's `eig` is called by default. The decomposition can also be approximated by Matlab (`'eigs'`) or by Nyström's method (`'eign'`) using `nb` components. In some cases one wants to disable (`'original'`) the rescaling of the principal components in feature space to unit length.

Full syntax

```
>> [eigval, eigvec] = kpca(X, kernel_fct, sig2, nb, rescaling)
```

Outputs

`eigval`	`N` (`nb`)×`1` vector with eigenvalues values
`eigvec`	`N`×`N` (`N`×`nb`) matrix with the principal directions

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`kernel`	Kernel type (e.g. `'RBF_kernel'`)
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`type`(*)	`'svd'`, `'eig'`(*),`'eigs'`,`'eign'`
`nb`(*)	Number of eigenvalues/eigenvectors used in the eigenvalue decomposition approximation
`rescaling`(*)	`'original size'` (`'o'`) or `'rescaling'`(*) (`'r'`)

See also:

`bay_lssvm, bay_optimize, eign`

A.3.18 latentlssvm

Purpose

Calculate the latent variables of the LS-SVM classifier at the given test data

Basic syntax

```
>>  Zt = latentlssvm({X,Y,'classifier',gam,sig2,kernel}, {alpha,b}, Xt)
>>  Zt = latentlssvm({X,Y,'classifier',gam,sig2,kernel}, Xt)
>>  [Zt, model] = latentlssvm(model, Xt)
```

Description

The latent variables of a binary classifier are the continuous simulated values of the test data which are used to make the final classifications. The classification of a testpoint depends on whether the latent value exceeds the model's threshold (b). If appropriate, the model is trained by the standard procedure (trainlssvm) first.

As an application example: crossvalidation can be based on the latent variables:

```
>> cost = crossvalidate(model, X, Y, 10,...
                  'mse', 'mean', 'original', 'trainlssvm', 'latentlssvm')
```

Full syntax

- Using the functional interface:

  ```
  >> Zt = latentlssvm({X,Y,type,gam,sig2,kernel,preprocess}, Xt)
  ```

  Outputs

Zt	Nt×m matrix with predicted latent simulated outputs

  Inputs

X	N×d matrix with the inputs of the training data
Y	N×1 vector with the outputs of the training data
type	'classifier' ('c')
gam	Regularization parameter
sig2	Kernel parameter (bandwidth in the case of the 'RBF_kernel')
kernel(*)	Kernel type (by default 'RBF_kernel')
preprocess(*)	'preprocess'(*) or 'original'
Xt	Nt×d matrix with the inputs of the test data

- Using the object oriented interface:

  ```
  >> [Zt, model] = latentlssvm(model, Xt)
  ```

  Outputs

Zt	Nt×m matrix with continuous latent simulated outputs
model(*)	Trained object oriented representation of the LS-SVM model

  Inputs

model	Object oriented representation of the LS-SVM model
Xt	Nt×d matrix with the inputs of the test data

See also:

trainlssvm, simlssvm

A.3.19 leaveoneout

Purpose

Estimate the performance of a trained model with leave-one-out crossvalidation

Basic syntax

```
>> leaveoneout({X,Y,type,gam,sig2}, Xval, Yval)
>> leaveoneout(model, Xval, Yval)
```

Description

In each iteration, one leaves one point, and fits a model on the other data points. The performance of the model is estimated based on the point left out. This procedure is repeated for each data point. Finally, all the different estimates of the performance are combined (default by computing the mean). The assumption is made that the input data is distributed independent and identically over the input space. A statistical bias reduction technique can be applied.

By default, this function will call the training (*trainlssvm*) and simulation (*simlssvm*) algorithms for LS-SVMs. However, one can use the validation function more generically by specifying the appropriate training and simulation function.

Full syntax

- Using the functional interface for the LS-SVMs:

```
>> [cost, costs, el] = leaveoneout({X,Y,type,gam,sig2,kernel,preprocess},...
                                    Xval, Yval, estfct, combinefct, correction)
```

Outputs

`cost`	Cost estimated by leave-one-out crossvalidation
`costs`(*)	`N`×`1` vector with the costs of the `N` folds
`el`(*)	`N`×`1` vector with the leave-one-out residuals

Inputs

`X`	Training input data used for defining the LS-SVM and the preprocessing
`Y`	Training output data used for defining the LS-SVM and the preprocessing
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`Xval`	`N`×`d` matrix with the inputs of the data used for leave-one-out crossvalidation
`Yval`	`N`×`m` matrix with the outputs of the data used for leave-one-out crossvalidation
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)
`combinefct`(*)	Function combining the estimated costs on the different folds (by default `mean`)
`correction`(*)	`'original'`(*) or `'corrected'`

- Using the object oriented interface for the LS-SVMs:

```
>> [cost, costs, el] = leaveoneout(model, Xval, Yval, ...
                                    estfct, combinefct, correction)
```

Outputs

`cost`	Cost estimated by leave-one-out crossvalidation
`costs`(*)	`N`×`1` vector with costs estimated on the `N` different folds
`el`(*)	`N`×`1` vector with residuals of all data

Inputs

`model`	Object oriented representation of the model
`Xval`	`Nt`×`d` matrix with the inputs of the validation points used
`Yval`	`Nt`×`m` matrix with the outputs of the validation points used in the procedure
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)
`combinefct`(*)	Function combining the estimated costs on the different folds (by default `mean`)
`correction`(*)	`'original'`(*) or `'corrected'`

- Using other modeling techniques:

```
>> [cost, costs, el] = leaveoneout(model, Xval, Yval, ...
                              estfct, combinefct, correction, trainfct, simfct)
```

Outputs

`cost`	Cost estimated by leave-one-out crossvalidation
`costs`(*)	`N`×`1` vector with costs estimated on the `N` different folds
`el`(*)	`N`×`1` vector with residuals of all data

Inputs

`model`	Object oriented representation of the model
`Xval`	`Nt`×`d` matrix with the inputs of the validation points used
`Yval`	`Nt`×`m` matrix with the outputs of the validation points used in the procedure
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)
`combinefct`(*)	Function combining the estimated costs on the different folds (by default `mean`)
`correction`(*)	`'original'`(*) or `'corrected'`
`trainfct`	Function used to train the model
`simfct`	Function used to simulate test data with the model

See also:

`leaveoneout_lssvm, validate, crossvalidate, trainlssvm, simlssvm`

A.3.20 leaveoneout_lssvm

Purpose

Fast leave-one-out cross-validation for the LS-SVM based on one full matrix inversion

Basic syntax

```
>> cost = leaveoneout_lssvm({X,Y,type,gam,sig2})
>> cost = leaveoneout_lssvm(model)
```

Description

This implementation is based on the matrix inversion lemma. Based on one global kernel matrix inversion, one can compute simultaneously all folds. One can evaluate simultaneously the leave-one-out error for a number of regularization parameters by passing them as an vector.

```
>> costs = leaveoneout_lssvm(model, [gam1, gam2, ...])
```

A different estimation function can be used (the default is `mse`), e.g. the mean absolute error:

```
>> costs = leaveoneout_lssvm(model, [], 'mae')
```

Full syntax

The number of different regularization parameters is denoted by `g`.

- Using the functional interface:

```
>> [cost, el, Yl] = leaveoneout_lssvm({X,Y,type,[],sig2,kernel,preprocess}, gams, estfct)
```

Outputs

`cost`	`g`×`1` vector with leave-one-out cost estimations corresponding with the number of passed regularization parameters.
`el`(*)	`N`×`g` matrix with the residuals of `g` different regularization parameters
`Yl`(*)	`N`×`g` matrix with the estimated (latent) outputs of the training data corresponding with the `g` different regularization parameters

Inputs

`X`	Training input data used for defining the LS-SVM and the preprocessing
`Y`	Training output data used for defining the LS-SVM and the preprocessing
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`gams`	`g`×`1` vector with different regularization parameters one wants to evaluate
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)

- Using the object oriented interface:

```
>> [cost, el, Yl, model] = leaveoneout_lssvm(model, gams, estfct)
```

Outputs

`cost`	`g`×`1` vector with different regularization parmeters
`el`(*)	`N`×`g` matrix with the residuals corresponding with the `g` different regularization parmeters
`Yl`(*)	`N`×`g` matrix with the estimated (latent) outputs of the training data corresponding with the `g` different regularization parameters
`model`(*)	Trained object oriented representation of the model

Inputs

`model`	Object oriented representation of the model
`gams`(*)	Different regularization parameters one wants to evaluate
`estfct`(*)	Function estimating the cost based on the residuals (by default `mse`)

See also:

`leaveoneout, crossvalidate, trainlssvm`

A.3.21 lin_kernel, MLP_kernel, poly_kernel, RBF_kernel

Purpose

Kernel implementations used with the Matlab training and simulation procedure

Description

`lin_kernel`

Linear kernel:

$$K(x_i, x_j) = x_i^T x_j$$

`MLP_kernel`

Multilayer perceptron kernel:

$$K(x_i, x_j) = \tanh(s x_i^T x_j + t^2)$$

with t the bias and s the scale parameter.

`poly_kernel`

Polynomial kernel:

$$K(x_i, x_j) = (x_i^T x_j + t)^d$$

with t the intercept and d the degree of the polynomial.

`RBF_kernel`

Radial Basis Function kernel:

$$K(x_i, x_j) = \exp - \frac{||x_i - x_j||^2}{\sigma^2}$$

with σ^2 the variance of the Gaussian kernel.

Full syntax

```
>> v = RBF_kernel(x1, X2, sig2)
```

Outputs

`v`	`N`×`1` vector with kernel values

Calls

`RBF_kernel`	or `lin_kernel`, `MLP_kernel`, `poly_kernel`,...

Inputs

`x1`	`1`×`d` matrix with a data point
`X2`	`N`×`d` matrix with data points
`sig2`	Kernel parameters

See also:

`kernel_matrix, kpca, trainlssvm`

A.3.22 `linf, mae, medae, misclass, mse, trimmedmse`

Purpose

Cost measures of resiuals

Description

A variety of global distance measures can be defined:

- `mae`: L_1

$$C_{L_1}(e) = \frac{\sum_{i=1}^{N} |e_i|}{N}$$

- `medae`: L_1

$$C_{L_1}^{median}(e) = \text{median}_{i=1}^{N} |e_i|$$

- `minf`: L_∞

$$C_{L_\infty}(e) = \sup_i |e_i|$$

- `misclass`: L_0

$$C_{L_0}(e) = \frac{\sum_{i=1}^{N} |y_i == \hat{y}_i|}{N}$$

- `mse`: L_2

$$C_{L_2}(e) = \frac{\sum_{i=1}^{N} e_i^2}{N}$$

- `trimmedmse`: L_2

$$C_{L_2}(e) = \frac{\sum_{s=1}^{n} e_{(s)}^2}{n}$$

where the residuals e_i are sorted as $e_{(s)}$. The mean of the squared residuals is taken for $s = 1 \ldots n < N$ with $1 - \frac{n}{N}$ the trimming factor.

Full syntax

- `>> C = mse(e)`

  Outputs
  - `C` Estimated cost of the residuals

  Calls
  - `mse` `mae`, `medae`, `minf` or `mse`

  Inputs
  - `e` `N×d` matrix with residuals

- `>> [C, which] = trimmedmse(e, beta, norm)`

  Outputs
  - `C` Estimated cost of the residuals
  - `which`(*) `N×d` matrix with indexes of the used residuals

  Inputs
  - `e` `N×d` matrix with residuals
  - `beta`(*) Trimming factor (by default `0.15`)
  - `norm`(*) Function implementing norm (by default `squared norm`)

- `>> [rate, n, which] = misclass(Y, Yh)`

  Outputs

`rate`	Rate of misclassification (between 0 (none misclassified) and 1 (all misclassified))
`n`(*)	Number of misclassified data points
`which`(*)	Indexes of misclassified points

  Inputs

`Y`	`N`×`d` matrix with true class labels
`Yh`	`N`×`d` matrix with estimated class labels

See also:

`crossvalidate, leaveoneout, leaveoneout_lssvm`

A.3.23 plotlssvm

Purpose

Plot the LS-SVM results in the environment of the training data

Basic syntax

```
>> plotlssvm({X,Y,type,gam, sig2, kernel})
>> plotlssvm({X,Y,type,gam, sig2, kernel}, {alpha,b})
>> model = plotlssvm(model)
```

Description

The first argument specifies the LS-SVM. The latter specifies the results of the training if already known. Otherwise, the training algorithm is first called. One can specify the precision of the plot by specifying the `grain` of the grid. By default this value is 50. The dimensions (`seldims`) of the input data to display can be selected as an optional argument in case of higher dimensional inputs (> 2). A grid will be taken over this dimension, while the other inputs remain constant (`0`).

Full syntax

- Using the functional interface:

```
>> plotlssvm({X,Y,type,gam,sig2,kernel,preprocess}, {alpha,b}, grain, seldims)
>> plotlssvm({X,Y,type,gam,sig2,kernel,preprocess}, [],      , grain, seldims)
```

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`alpha`(*)	Support values obtained from training
`b`(*)	Bias term obtained from training
`grain`(*)	The grain of the grid evaluated to compose the surface (by default `50`)
`seldims`(*)	The principal inputs one wants to span a grid (by default `[1 2]`)

- Using the object oriented interface:

```
>> model = plotlssvm(model, [], grain, seldims)
```

Outputs

`model`(*)	Trained object oriented representation of the LS-SVM model

Inputs

`model`	Object oriented representation of the LS-SVM model
`grain`(*)	The grain of the grid evaluated to compose the surface (by default `50`)
`seldims`(*)	The principal inputs one wants to span a grid (by default `[1 2]`)

See also:

`trainlssvm, simlssvm`.

A.3.24 predict

Purpose

Iterative prediction of a trained LS-SVM NARX model (in recurrent mode)

Description

```
>> Yp = predict({Xw,Yw,type,gam,sig2}, Xt, nb)
>> Yp = predict(model,                 Xt, nb)
```

Description

The model needs to be trained using `Xw, Yw` which is the result of `windowize` or `windowizeNARX`. The number of time lags for the model is determined by the dimension of the input, or if not appropriate, by the number of given starting values.

By default, the model is evaluated on the previous points using `simlssvm`. However, if one wants to use this procedure for other models, this default can be overwritten by your favorite training function. This function (denoted by `simfct`) has to follow the following syntax:

```
>> simfct(model,inputs,arguments)
```

thus:

```
>> Yp = predict(model, Xt, nb, simfct)
>> Yp = predict(model, Xt, nb, simfct, arguments)
```

Full syntax

- Using the functional interface for the LS-SVMs:

  ```
  >> Yp = predict({Xw,Yw,type,gam,sig2,kernel,preprocess}, Xt, nb)
  ```

  Outputs

`Yp`	`nb`×`m` matrix with the predictions

  Inputs

`Xw`	`N`×`d` matrix with the inputs of the training data
`Yw`	`N`×`w` matrix with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'` or `'original'` (by default)
`Xt`	`nb*d` matrix of the starting points for the prediction
`nb`(*)	Number of outputs to predict

- Using the object oriented interface with LS-SVMs:

  ```
  >> Yp = predict(model, Xt, nb)
  ```

  Outputs

`Yp`	`nb`×`m` matrix with the predictions

  Inputs

`Xt`	`nb`×`d` matrix of the starting points for the prediction
`nb`(*)	Number of outputs to predict

- Using another model:

```
>> Yp = predict(model, Xt, nb, simfct, arguments)
```

Outputs

`Yp`	`nb`×`m` matrix with the predictions

Inputs

`Xt`	`nb`×`d` matrix of the starting points for the prediction
`nb`	Number of outputs to predict
`simfct`	Function used to evaluate a test point
`arguments`(*)	Cell with the extra arguments passed to `simfct`

See also:

`windowize`, `trainlssvm`, `simlssvm`.

A.3.25 prelssvm, postlssvm

Purpose

Pre- and postprocessing of the LS-SVM

Description

These functions should only be called by `trainlssvm` or by `simlssvm`. At first the preprocessing assigns a label to each in- and output component (`c` for continuous, `a` for categorical or `b` for binary variables). According to this label each dimension is rescaled:

- continuous: zero mean and unit variance
- categorical: no preprocessing
- binary: labels -1 and $+1$

Full syntax

Using the object oriented interface:

- Preprocessing:

```
>>  model   = prelssvm(model)
>> [Xp, Yp] = prelssvm(model, Xt, Yt)
```

Outputs

`model`	Preprocessed object oriented representation of the LS-SVM model
`Xp`	`Nt`×`d` matrix with the preprocessed inputs of the test data
`Yp`	`Nt`×`d` matrix with the preprocessed outputs of the test data

Inputs

`model`	Object oriented representation of the LS-SVM model
`Xt`	`Nt`×`d` matrix with the inputs of the test data to preprocess
`Yt`	`Nt`×`d` matrix with the outputs of the test data to preprocess

- Postprocessing:

```
>>  model = postlssvm(model)
>> [Xt, Yt] = postlssvm(model, Xp, Yp)
```

Outputs

`model`	Postprocessed object oriented representation of the LS-SVM model
`Xt`	`Nt`×`d` matrix with the postprocessed inputs of the test data
`Yt`	`Nt`×`d` matrix with the postprocessed outputs of the test data

Inputs

`model`	Object oriented representation of the LS-SVM model
`Xp`	`Nt`×`d` matrix with the inputs of the test data to postprocess
`Yp`	`Nt`×`d` matrix with the outputs of the test data to postprocess

A.3.26 ridgeregress

Purpose

Linear ridge regression

Basic syntax

```
>>   [W, b]     = ridgeregress(X, Y, gam)
>>   [W, b, Yt] = ridgeregress(X, Y, gam, Xt)
```

Description

Ordinary Least squares with a regularization parameter (gam).

Full syntax

```
>> [W, b, Yt] = ridgeregress(X, Y, gam, Xt)
```

Outputs

W	d×1 vector with the regression coefficients
b	bias term
Yt(*)	Nt×1 vector with predicted outputs of test data

Inputs

X	N×d matrix with the inputs of the training data
Y	N×1 vector with the outputs of the training data
gam	Regularization parameter
Xt(*)	Nt×d matrix with the inputs of the test data

See also:

bay_rr,bay_lssvm

A.3.27 robustlssvm

Purpose

Robust training in the case of non-Gaussian noise or outliers

Basic syntax

```
>> [alpha, b] = robustlssvm({X,Y,type,gam,sig2,kernel})
>>  model     = robustlssvm(model)
```

Robustness towards outliers can be achieved by reducing the influence of support values corresponding to large errors.

Full syntax

- Using the functional interface:

  ```
  >> [alpha, b] = robustlssvm({X,Y,type,gam,sig2,kernel}, {alpha,b})
  ```

  Outputs

alpha	N×m matrix with support values of the robust LS-SVM
b	1×m vector with bias term(s) of the robust LS-SVM

  Inputs

X	N×d matrix with the inputs of the training data
Y	N×1 vector with the outputs of the training data
type	'function estimation' ('f') or 'classifier' ('c')
gam	Regularization parameter
sig2	Kernel parameter (bandwidth in the case of the 'RBF_kernel')
kernel(*)	Kernel type (by default 'RBF_kernel')
preprocess(*)	'preprocess'(*) or 'original'
alpha(*)	Support values obtained from training
b(*)	Bias term obtained from training

- Using the object oriented interface:

  ```
  >> model = robustlssvm(model)
  ```

  Outputs

model	Robustly trained object oriented representation of the LS-SVM model

  Inputs

model	Object oriented representation of the LS-SVM model

See also:

trainlssvm, tunelssvm, rcrossvalidate

A.3.28 roc

Purpose

Receiver Operating Characteristic (ROC) curve of a binary classifier

Basic syntax

```
>> [area, se, deltab, oneMinusSpec, sens, TN, TP, FN, FP] = roc(Zt, Y)
```

Description

The ROC curve shows the separation abilities of a binary classifier: by iteratively setting the possible classifier thresholds, the dataset is tested on misclassifications [9]. As a result, a plot is shown where the various outcomes are described. If the plot has a surface of 1 on test data, a perfectly separating classifier is found (on that particular dataset), if the area equals 0.5, the classifier has no discriminative power at all. In general, this function can be called with the latent variables `Zt` and the corresponding class labels `Yclass`

```
>> Zt       = [-.7             Yclass = [-1
               .3                       -1
              1.5                        1
              ...                       ..
              -.2]                       1]
>> roc(Zt, Yclass)
```

For use in LS-SVMlab, a shorthand notation allows making the ROC curve on the training data. Implicit training and simulation of the latent values simplifies the call.

```
>> roc({X,Y,'classifier',gam,sig2,kernel})
>> roc(model)
```

Full syntax

- Standard call (LS-SVMlab independent):

  ```
  >> [area, se, deltab, oneMinusSpec, sens, TN, TP, FN, FP] = roc(Zt, Y, figure)
  ```

  Outputs

`area`(*)	Area under the ROC curve
`se`(*)	Standard deviation of the residuals
`deltab`(*)	`N`×`1` different thresholds value
`oneMinusSpec`(*)	1-Specificity of each threshold value
`sens`(*)	Sensibility for each threshold value
`TN`(*)	Number of true negative predictions
`TP`(*)	Number of true positive predictions
`FN`(*)	Number of false negative predictions
`FP`(*)	Number of false positive predictions

  Inputs

`Zt`	`N`×`1` latent values of the predicted outputs
`Y`	`N`×`1` of true class labels
`figure`(*)	`'figure'`(*) or `'nofigure'`

- Using the functional interface for the LS-SVMs:

  ```
  >> [area, se, deltab, oneMinusSpec, sens, TN, TP, FN, FP] = ...
                              roc({X,Y,'classifier',gam,sig2,kernel}, figure)
  ```

Outputs

area(*)	Area under the ROC curve
se(*)	Standard deviation of the residuals
deltab(*)	Different thresholds
oneMinusSpec(*)	1-Specificity of each threshold value
sens(*)	Sensibility for each threshold value
TN(*)	Number of true negative predictions
TP(*)	Number of true positive predictions
FN(*)	Number of false negative predictions
FP(*)	Number of false positive predictions

Inputs

X	N×d matrix with the inputs of the training data
Y	N×1 vector with the outputs of the training data
type	'function estimation' ('f') or 'classifier' ('c')
gam	Regularization parameter
sig2	Kernel parameter (bandwidth in the case of the 'RBF_kernel')
kernel(*)	Kernel type (by default 'RBF_kernel')
preprocess(*)	'preprocess'(*) or 'original'
figure(*)	'figure'(*) or 'nofigure'

- Using the object oriented interface for the LS-SVMs:

```
>> [area, se, deltab, oneMinusSpec, sens, TN, TP, FN, FP] = roc(model, figure)
```

Outputs

area(*)	Area under the ROC curve
se(*)	Standard deviation of the residuals
deltab(*)	N×1 vector with different thresholds
oneMinusSpec(*)	1-Specificity of each threshold value
sens(*)	Sensibility for each threshold value
TN(*)	Number of true negative predictions
TP(*)	Number of true positive predictions
FN(*)	Number of false negative predictions
FP(*)	Number of false positive predictions

Inputs

model	Object oriented representation of the LS-SVM model
figure(*)	'figure'(*) or 'nofigure'

See also:

deltablssvm, trainlssvm

A.3.29 simlssvm

Purpose

Evaluate the LS-SVM at given points

Basic syntax

```
>> Yt = simlssvm({X,Y,type,gam,sig2,kernel}, {alpha,b}, Xt)
>> Yt = simlssvm({X,Y,type,gam,sig2,kernel}, Xt)
>> Yt = simlssvm(model, Xt)
```

Description

The matrix `Xt` represents the points one wants to predict. The first cell contains all arguments needed for defining the LS-SVM (see also `trainlssvm, initlssvm`). The second cell contains the results of training this LS-SVM model. The cell syntax allows for flexible and consistent default handling.

As in training, three different implementations are included (`simclssvm.mex*, simFILE.x, simlssvm.m`). The **cmex** algorithm is called, except when specified otherwise. After a simulation call, a `.` is displayed.

Full syntax

- Using the functional interface:

```
>> [Yt, Zt] = simlssvm({X,Y,type,gam,sig2,kernel},            Xt)
>> [Yt, Zt] = simlssvm({X,Y,type,gam,sig2,kernel}, {alpha,b}, Xt)
```

Outputs

`Yt`	`Nt`×`m` matrix with predicted output of test data
`Zt`(*)	`Nt`×`m` matrix with predicted latent variables of a classifier

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`alpha`(*)	Support values obtained from training
`b`(*)	Bias term obtained from training
`Xt`	`Nt`×`d` inputs of the test data

- Using the object oriented interface:

```
>> [Yt, Zt, model] = simlssvm(model, Xt)
```

Outputs

`Yt`	`Nt`×`m` matrix with predicted output of test data
`Zt`(*)	`Nt`×`m` matrix with predicted latent variables of a classifier
`model`(*)	Object oriented representation of the LS-SVM model

Inputs

`model`	Object oriented representation of the LS-SVM model
`Xt`	`Nt`×`d` matrix with the inputs of the test data

See also:

`trainlssvm, initlssvm, plotlssvm, code, changelssvm`

A.3.30 sparselssvm

Purpose

Remove iteratively the least relevant support vectors in order to obtain sparsity

Basic syntax

```
>> selector = sparselssvm({X,Y,type,gam, sig2}, tradeoff, step)
>> model    = sparselssvm(model,                tradeoff, step)
```

Description

In each iteration `step` (by default 5%) of the support values are set to zero until the performance becomes less than `cutoff` procent (default 75%) of the original performance.

Full syntax

- Using the functional interface:

```
>> [selector, errest] = ...
         sparselssvm({X,Y,type,gam, sig2,kernel,preprocess}, cutoff, step)
```

Outputs	
`selector`	`N×1` vector of index of chosen support vectors
`errest`(*)	Estimated cost on all training data after pruning
Inputs	
`X`	`N×d` matrix with the inputs of the training data
`Y`	`N×1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`cutoff`(*)	Cutoff between the validation of the original to the pruned LS-SVM
`step`(*)	Number of the pruned support vectors in each iteration step

- Using the object oriented interface:

```
>> [model, errest] = sparselssvm(model, cutoff, step)
```

Outputs	
`model`	Pruned object oriented representation of the LS-SVM model
`errest`(*)	Estimated cost on all training data after pruning
Inputs	
`model`	Object oriented representation of the LS-SVM model
`cutoff`(*)	Cutoff between the validation of the original to the pruned LS-SVM
`step`(*)	Number of the pruned support vectors in each iteration step

See also:

`trainlssvm, tunelssvm, robustlssvm, crossvalidate`

A.3.31 trainlssvm

Purpose

Train the support values and the bias term of an LS-SVM for classification or function approximation

Basic syntax

```
>> [alpha, b] = trainlssvm({X,Y,type,gam,kernel_par,kernel,preprocess})
>> model      = trainlssvm(model)
```

Description

`type` can be `'classifier'` or `'function estimation'` (these strings can be abbreviated into `'c'` or `'f'`, respectively). `X` and `Y` are matrices holding the training input and output data. The i-th data point is represented by the i-th row `X(i,:)` and `Y(i,:)`. `gam` is the regularization parameter: for `gam` low minimizing of the complexity of the model is emphasized, for `gam` high, good fitting of the training data points is stressed. `kernel_par` is the parameter of the kernel; in the common case of an RBF kernel, a large `sig2` indicates a stronger smoothing. The `kernel_type` indicates the function that is called to compute the kernel value (by default `RBF_kernel`). Other kernels can be used for example:

```
>> [alpha, b] = trainlssvm({X,Y,type,gam,[d; p],'poly_kernel'})
>> [alpha, b] = trainlssvm({X,Y,type,gam,[]   ,'lin_kernel'})
```

The kernel parameter(s) are passed as a row vector, *in the case no kernel parameter is needed, pass the empty vector!*

The training can either be proceeded by the preprocessing function (`'preprocess'`) (by default) or not (`'original'`). The training calls the preprocessing (`prelssvm, postlssvm`) and the encoder (`codelssvm`) if appropiate.

In the remainder of the text, the content of the cell determining the LS-SVM is given by `{X,Y, type, gam, sig2}`. However, the additional arguments in this cell can always be added in the calls.

If one uses the object oriented interface (see also A.3.14), the training is done by

```
>> model = trainlssvm(model)
>> model = trainlssvm(model, X, Y)
```

The status of the model checks whether a retraining is needed. The extra arguments `X, Y` allow to re-initialize the model with this new training data as long as its dimensions are the same as the old initiation.

Three training implementations are included:

- **The C-implementation linked with CMEX:** this implementation is based on the iterative solver Conjugate Gradient algorithm (CG) (`lssvm.mex*`). After this training call, a `'-'` is displayed. This is recommended for use on larger data sets.
- **The C-implementation called via a buffer file:** this is based on CG; check if the executable 'lssvmFILE.x' is in the current directory; (`lssvmFILE.x`). After this training call, a `'-'` is displayed.
- **The Matlab implementation:** a straightforward implementation based on the matrix division '\' (`lssvmMATLAB.m`). After this training call, a `'~'` is displayed. This is recommended for a number of training data points smaller than 500 (depending on the computer memory).

By default, the **cmex** implementation is called. If this one fails, the Matlab implementation is chosen instead. One can specify explicitly which implementation to use using the object oriented interface.

This implementation allows to train a multidimensional output problem. If each output uses the same kernel type, kernel parameters and regularization parameter, this is straightforward. If not so, one can specify the different types and/or parameters as a row vector in the appropriate argument. Each dimension will be trained with the corresponding column in this vector.

```
>> [alpha, b] = trainlssvm({X, [Y_1 ... Y_d],type,...
                            [ gam_1 ... gam_d], ...
                            [sig2_1 ... sig2_d],...
                          {kernel_1,...,kernel_d}})
```

Full syntax

- Using the functional interface:

```
>> [alpha, b] = trainlssvm({X,Y,type,gam,sig2,kernel,preprocess})
```

Outputs

`alpha`	`N`×`m` matrix with support values of the LS-SVM
`b`	`1`×`m` vector with bias term(s) of the LS-SVM

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`

- Using the object oriented interface:

```
>> model = trainlssvm(model)
>> model = trainlssvm({X,Y,type,gam,sig2,kernel,preprocess})
```

Outputs

`model`(*)	Trained object oriented representation of the LS-SVM model

Inputs

`model`	Object oriented representation of the LS-SVM model
`X`(*)	`N`×`d` matrix with the inputs of the training data
`Y`(*)	`N`×`1` vector with the outputs of the training data
`type`(*)	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`(*)	Regularization parameter
`sig2`(*)	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`

See also:

`simlssvm, initlssvm, changelssvm, plotlssvm, prelssvm, codelssvm`

A.3.32 tunelssvm & gridsearch

Purpose

Tune the hyperparameters of the model with respect to the given performance measure

Basic syntax

```
[gam, sig2, cost] = tunelssvm({X,Y,type,init_gam, init_sig2})
[gam, sig2, cost] = ...
     tunelssvm({X,Y,type,igam,isig2}, optfun, optargs, costfun, costargs)
```

where `igam` and `isig2` are the initial values of the hyperparameters. Using the object oriented interface this becomes:

```
[model, cost] = tunelssvm(model)
[model, cost] = tunelssvm(model, optfun, optargs, costfun, costargs)
```

where `model` contains the initial values of the hyperparmeters.

Description

`optfun` is the optimization algorithm. By default `gridsearch` is used (this one is restricted to 2-dimensional hyperparameter optimization). One can also use `fminunc` of the Matlab optimization toolbox (although this is less robust).

The hyperparameters are the regularization parameter `gam` and the kernel parameters `sig2` (or `sig2` in the case of the `'RBF_kernel'`). `costfun` gives an estimate of the performance of the model. This function is called as

```
>> cost = costfun(model, costargs{:})
```

The default values are

```
>> model =  tunelssvm(model, 'gridsearch', {}, 'crossvalidate', {X,Y})
```

where `X,Y` are the original training data. Possible alternative function calls in the case of function approximation are

```
>> [gam, sig2, cost] = ...
              tunelssvm({X,Y,type,igam,isig2}, 'fminunc', [])
>> model = tunelssvm(model, 'gridsearch', {}, 'leaveoneout_lssvm', {X,Y})
>> model = tunelssvm(model, 'gridsearch', {}, 'crossvalidate', {X,Y,10,'misclass'})
```

In the case of classification

```
>> model = tunelssvm(model, 'gridsearch', {},...
             'crossvalidate',{model.xtrain, model.ytrain,10,'misclass'})
```

```
>> Xopt = gridsearch(fun, startvalues)
```

The most simple algorithm to determine the minimum of a cost function with possibly multiple optima is to evaluate a grid over the parameter space and to pick the minimum. This procedure iteratively zooms to the candiade optimum. The startvalues determine the limits of the grid over parameter space.

An example is given:

```
 >> fun = inline('1-exp(-norm([X(1) X(2)]))','X');
 >> gridsearch(fun,[-3 3; 3 -3])
```

Full syntax

- Using the functional interface:

```
>> [gam, sig2, cost] = tunelssvm({X,Y,type,igam,isig2,kernel,preprocess},...
                                                  optfun, optargs, costfun, costargs)
```

Outputs

`gam`	Optimal regularization parameter
`sig2`	Optimal kernel parameter(s)
`cost`(*)	Estimated cost of the optimal hyperparameters

Inputs

`X`	`N`×`d` matrix with the inputs of the training data
`Y`	`N`×`1` vector with the outputs of the training data
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`igam`	Starting value of the regularization parameter
`isig2`	Starting value of the kernel parameter(s) (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`optfun`(*)	Optimization function (by default `'gridsearch'`)
`optfun`(*)	Cell with extra optimization function arguments
`costfun`(*)	Function estimating the cost-criterion (by default `'crossvalidate'`)
`optfun`(*)	Cell with extra cost function arguments

- Using the object oriented interface:

```
>> [model, cost] = tunelssvm(model, optfun, optargs, costfun, costargs)
```

Outputs

`model`	Object oriented representation of the LS-SVM model with optimal hyperparameters
`cost`(*)	Estimated cost of the optimal hyperparameters

Inputs

`model`	Object oriented representation of the LS-SVM model with initial hyperparameters
`optfun`(*)	Optimization function (by default `'gridsearch'`)
`optfun`(*)	Cell with extra optimization function arguments
`costfun`(*)	Function estimating the cost-criterion (by default `'crossvalidate'`)
`optfun`(*)	Cell with extra cost function arguments

- Optimization by exhaustive search:

```
>> [Xopt, Yopt, iterations] = gridsearch(fun, startvalues, funargs, ...
                          maxFunEvals, convspeed, TolFun, TolX, grain, nofigure)
```

Outputs

`Xopt`	Optimal parameter set
`Yopt`	Criterion evaluated at `Xopt`
`iterations`	Used number of iterations

Inputs

`fun`	Function implementing the cost criterion
`startvalues`	`2*d` matrix with starting values of the optimization routine
`funargs`(*)	Cell with optional extra function arguments of `fun`
`MaxFunEvals`(*)	Maximum number of function evaluations
`convspeed`(*)	Zoom-factor after each iteration
`TolFun`(*)	Minimal toleration of improvement on function value
`TolX`(*)	Minimal toleration of improvement on X value
`grain`(*)	Square root number of function evaluations in one grid
`nofigure`(*)	`'figure'`(*) or `'nofigure'`

See also:

`trainlssvm, crossvalidate, validate`

A.3.33 validate

Purpose

Validate a trained model on a fixed validation set

Basic syntax

```
>> cost = validate({X,Y,type,gam,sig2}, Xtrain, Ytrain, Xtest, Ytest)
```

Description

In the case of regression, most common is to use the mean squared error (`'mse'`) as an estimate of the cost of the model. It is known that the trimmed mean of the squared errors (`'trimmedmse'`) is more robust estimate when non-Gaussian noise or outliers occur. For classification, a suitable cost criterion is the rate of misclassification (`'misclass'`).

By default, this function will call the training (`'trainlssvm'`) and simulation (`'simlssvm'`) algorithms for LS-SVMs. However, one can use the validation function more generically by specifying the appropriate training and simulation function:

```
>> cost = validate(model, Xtrain, Ytrain, Xtest, Ytest, costfunction)
>> cost = validate(model, Xtrain, Ytrain, Xtest, Ytest, ...
                           costfunction, trainfunction, simfunction)
```

Full syntax

- Using the functional interface for the LS-SVMs:

```
>> cost = validate({X,Y,type,gam,sig2,kernel,preprocess},...
                            Xtrain, Ytrain, Xtest, Ytest, estfct)
```

Outputs

`cost`	Cost estimated by validation on test set

Inputs

`X`	Training input data used for defining the LS-SVM and the preprocessing
`Y`	Training output data used for defining the LS-SVM and the preprocessing
`type`	`'function estimation'` (`'f'`) or `'classifier'` (`'c'`)
`gam`	Regularization parameter
`sig2`	Kernel parameter (bandwidth in the case of the `'RBF_kernel'`)
`kernel`(*)	Kernel type (by default `'RBF_kernel'`)
`preprocess`(*)	`'preprocess'`(*) or `'original'`
`Xtrain`	`N`×`d` matrix with the input data used for training
`Ytrain`	`N`×`m` matrix with the output data used for training
`Xtest`	`N`×`d` matrix with the input data used for testing
`Ytest`	`N`×`m` matrix with the output data used for testing
`estfct`(*)	Function estimating the cost based on the residuals (by default `'mse'`)

- Using the object oriented interface for the LS-SVMs:

```
>> cost = validate(model, Xtrain, Ytrain, Xtest, Ytest, estfct)
```

Outputs

`cost`	Cost estimated by validation on test set

Inputs

`model`	Object oriented representation of the model
`Xtrain`	`N`×`d` matrix with the input data used for training
`Ytrain`	`N`×`m` matrix with the output data used for training
`Xtest`	`N`×`d` matrix with the input data used for testing
`Ytest`	`N`×`m` matrix with the output data used for testing
`estfct`(*)	Function estimating the cost based on the residuals (by default `'mse'`)

- Using other modeling techniques::

```
>> cost = validate(model, Xtrain, Ytrain, Xtest, Ytest, estfct, trainfct, simfct)
```

Outputs

`cost`	Cost estimated by validation on test set

Inputs

`model`	Object oriented representation of the model
`Xtrain`	`N`×`d` matrix with the input data used for training
`Ytrain`	`N`×`m` matrix with the output data used for training
`Xtest`	`N`×`d` matrix with the input data used for testing
`Ytest`	`N`×`m` matrix with the output data used for testing
`estfct`(*)	Function estimating the cost based on the residuals (by default `'mse'`)
`trainfct`	Function used for training `model`
`simfct`	Function used for simulating `model`

See also:

`crossvalidate, leaveoneout`

A.3.34 windowize & windowizeNARX

Purpose

Re-arrange the data points into a (block) Hankel matrix for (N)AR(X) time-series modeling

Basic Syntax

```
>> w = windowize(A, window)
>> [Xw,Yw] = windowizeNARX(X,Y,xdelays, ydelays, steps)
```

Description

Use `windowize` function to make a nonlinear AR predictor with a nonlinear regressor. The last elements of the resulting matrix will contain the future values of the time-series, the others will contain the past inputs. `window` is the relative index of data points in matrix `A`, that are selected to make a window. Each window is put in a row of matrix `W`. The matrix `W` contains as many rows as there are different windows selected in `A`.

Schematically, this becomes

```
 >> A = [a1 a2 a3;
         b1 b2 b3;
         c1 c2 c3;
         d1 d2 d3;
         e1 e2 e3;
         f1 f2 f3;
         g1 g2 g3];

>> W = windowize(A, [1 2 3])

   W =
     a1 a2 a3  b1 b2 b3  c1 c2 c3
     b1 b2 b3  c1 c2 c3  d1 d2 d3
     c1 c2 c3  d1 d2 d3  e1 e2 e3
     d1 d2 d3  e1 e2 e3  f1 f2 f3
     e1 e2 e3  f1 f2 f3  g1 g2 g3
```

The function `windowizeNARX` converts the time-series and his exogeneous variables into a block hankel format useful for training a nonlinear function approximation as a nonlinear ARX model.

Full syntax

- `>> Xw = windowize(X, window)`

  The length of `window` is denoted by `w`.

  Outputs

  `Xw` `(N-w+1)`×—w— matrix of the sequences of windows over `X`

  Inputs

  `X` `N`×`1` vector with data points

  `w` `w`×`1` vector with the relative indices of one window

- `>> [Xw, Yw, xdim, ydim, n] = windowizeNARX(X, Y, xdelays, ydelays, steps)`

Outputs

`Xw`	Matrix of the data used for input including the delays
`Yw`	Matrix of the data used for output including the next steps
`xdim`(*)	Number of dimensions in new input
`ydim`(*)	Number of dimensions in new output
`n`(*)	Number of new data points

Inputs

`X`	`N`×`m` vector with input data points
`Y`	`N`×`d` vector with output data points
`xdelays`	Number of lags of `X` in new input
`ydelays`	Number of lags of `Y` in new input
`steps`(*)	Number of future steps of `Y` in new output (by default 1)

See also:

`windowizeNARX, predict, trainlssvm, simlssvm`

Bibliography

[1] Baudat G., Anouar F. (2001), "Kernel-based methods and function approximation", in *International Joint Conference on Neural Networks* (IJCNN 2001), Washington DC USA, 1244–1249.

[2] Cawley G.C., Talbot N.L.C. (2002), "efficient formation of a basis in a kernel induced feature space", in *Proc. European Symposium on Artificial Neural Networks* (ESANN 2002), Brugge Belgium, 1–6.

[3] Cristianini N., Shawe-Taylor J. (2000), *An Introduction to Support Vector Machines*, Cambridge University Press.

[4] De Brabanter J., Pelckmans K., Suykens J.A.K., Vandewalle J. (2002), "Robust cross-validation score function for LS-SVM non-linear function estimation", *International Conference on Artificial Neural Networks* (ICANN 2002), Madrid Spain, Madrid, Spain, Aug. 2002, 713–719.

[5] Evgeniou T., Pontil M., Poggio T. (2000), "Regularization networks and support vector machines," *Advances in Computational Mathematics*, **13**(1), 1–50.

[6] Girolami M. (2002), "Orthogonal series density estimation and the kernel eigenvalue problem", *Neural Computation*, **14**(3), 669–688.

[7] Golub G.H. and Van Loan C.F. (1989), *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD.

[8] Hamers B., Suykens J.A.K., De Moor B. (2001), "A comparison of iterative methods for least squares support vector machine classifiers", Internal Report 01-110, ESAT-SISTA, K.U.Leuven (Leuven, Belgium).

[9] Hanley J.A., McNeil B.J. (1982), "The meaning and use of the area under a receiver operating characteristic (ROC) curve" *Radiology* 1982; **143**, 29-36.

[10] Huber P.J. (1964), "Robust estimation of a location parameter", *Ann. Math. Statist.*, **35**, 73–101.

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

[12] Mika S., Schölkopf B., Smola A., Müller K.-R., Scholz M., Ratsch G. (1999), "Kernel PCA and de-noising in feature spaces", *Advances in Neural Information Processing Systems 11*, 536–542, MIT Press.

[13] Mika S., Rätsch G., Weston J., Schölkopf B., Müller K.-R. (1999), "Fisher discriminant analysis with kernels", In *Neural Networks for Signal Processing IX*, 41–48, IEEE.

[14] Nabney I.T. (2002), *Netlab: Algorithms for Pattern Recognition*, Springer.

[15] Poggio T., Girosi F. (1990), "Networks for approximation and learning", *Proc. of the IEEE*, **78**, 1481–1497.

[16] Schölkopf B., Burges C., Smola A. (Eds.) (1998), *Advances in Kernel Methods - Support Vector Learning*, MIT Press.

[17] Schölkopf B., Smola A. J., Müller K.-R. (1998), "Nonlinear component analysis as a kernel eigenvalue problem", *Neural Computation*, **10**, 1299–1319.

[18] Schölkopf B., Smola A. (2002), *Learning with Kernels*, MIT Press.

[19] Smola A.J., Schölkopf B. (2000), "Sparse greedy matrix approximation for machine learning", *Proc. 17th International Conference on Machine Learning*, 911–918, San Francisco, Morgan Kaufman.

[20] Stone M. (1974), "Cross-validatory choice and assessment of statistical predictions", *J. Royal Statist. Soc. Ser. B*, **36**, 111–147.

[21] Suykens J.A.K., Vandewalle J. (1999), "Least squares support vector machine classifiers", *Neural Processing Letters*, **9**(3), 293–300.

[22] Suykens J.A.K., Vandewalle J. (2000), "Recurrent least squares support vector machines", *IEEE Transactions on Circuits and Systems-I*, **47**(7), 1109–1114.

[23] Suykens J.A.K., De Brabanter J., Lukas L., Vandewalle J. (2002), "Weighted least squares support vector machines : robustness and sparse approximation", *Neurocomputing*, Special issue on fundamental and information processing aspects of neurocomputing, **48**(1-4), 85–105.

[24] Suykens J.A.K., Van Gestel T., Vandewalle J., De Moor B. (2002), "A support vector machine formulation to PCA analysis and its Kernel version", accepted for publication in *IEEE Transactions on Neural Networks*.

[25] Suykens J.A.K., Van Gestel T., De Brabanter J., De Moor B., Vandewalle J. (2002), *Least Squares Support Vector Machines*, World Scientific, in press.

[26] Van Gestel T., Suykens J.A.K., Baestaens D., Lambrechts A., Lanckriet G., Vandaele B., De Moor B., Vandewalle J. (2001) "Financial time series prediction using least squares support vector machines within the evidence framework", *IEEE Transactions on Neural Networks* (special issue on Neural Networks in Financial Engineering), **12**(4), 809–821.

[27] Van Gestel T., Suykens J.A.K., De Moor B., Vandewalle J. (2001), "Automatic relevance determination for least squares support vector machine classifiers", *Proc. of the European Symposium on Artificial Neural Networks* (ESANN 2001), Bruges, Belgium, 13–18.

[28] Van Gestel T., Suykens J., Baesens B., Viaene S., Vanthienen J., Dedene G., De Moor B., Vandewalle J. (2001), "Benchmarking least squares support vector machine classifiers ", *Machine Learning*, in press.

[29] Van Gestel T., Suykens J.A.K., Lanckriet G., Lambrechts A., De Moor B., Vandewalle J. (2002), "Bayesian framework for least squares support vector machine classifiers, gaussian processes and kernel fisher discriminant analysis", *Neural Computation*, **15**(5), 1115–1148.

[30] Van Gestel T., Suykens J.A.K., Lanckriet G., Lambrechts A., De Moor B., Vandewalle J. (2002), "Multiclass LS-SVMs : moderated outputs and coding-decoding schemes", *Neural Processing Letters*, **15**(1), 45–58.

[31] Van Gestel T., Suykens J.A.K., De Moor B., Vandewalle J. (2002), "Bayesian inference for LS-SVMs on large data sets using the Nyström method", *International Joint Conference on Neural Networks* (WCCI-IJCNN 2002), Honolulu, USA, May 2002, 2779–2784.

[32] Van Gestel T. (2002), *From linear to kernel based methods in classification, modelling and prediction*, PhD thesis, K.U. Leuven Department of Electrical Engineering.

[33] Vapnik V. (1995), *The Nature of Statistical Learning Theory,* Springer-Verlag, New York.

[34] Vapnik V. (1998), *Statistical Learning Theory*, John Wiley, New-York.

[35] Williams C.K.I., Seeger M. (2001), "Using the Nyström method to speed up kernel machines", *Advances in neural information processing systems*, **13**, 682–688, MIT Press.

[36] Wahba G.,Wold S. (1975), "A completely automatic french curve: fitting spline functions by cross-validation", *Comm. Statist.*, **4**, 1-17.

[37] Wahba G. (1990), *Spline Models for observational data*, SIAM, **39**.