PRTools

Version 3.0

A Matlab Toolbox for Pattern Recognition

R.P.W. Duin

January 2000

An introduction into the setup, definitions and use of PRTools is given. Readers are assumed to be familiar with Matlab and should have a basic understanding of field of statistical pattern recognition.

Pattern Recognition Group
Delft University of Technolgy
P.O. Box 5046, 2600 GA Delft
The Netherlands

tel : +31 15 2786143
fax: +31 15 2786740
email: duin@tn.tudelft.nl
http://www.ph.tn.tudelft.nl/prtools

1. Introduction

In statistical pattern recognition one studies techniques for the generalisation of decision rules to be used for the recognition of patterns in experimental data sets. This area of research has a strong computational character, demanding a flexible use of numerical programs for studying the data as well as for evaluating the data analysis techniques themselves. As still new techniques are being proposed in the literature a programming platform is needed that enables a fast and flexible implementation. Pattern recognition is studied in almost all areas of applied science. Thereby the use of a widely available numerical toolset like Matlab may be profitable for both, the use of existing techniques, as well as for the study of new algorithms. Moreover, because of its general nature in comparison with more specialised statistical environments, it offers an easy integration with the preprocessing of data of any nature. This may certainly be facilitated by the large set of toolboxes available in Matlab.

The more than 100 routines offered by PRTools in its present state represent a basic set covering largely the area of statistical pattern recognition. In order to make the evaluation and comparison of algorithms more easy a set of data generation routines is included, as well as a small set of 'standard' real world datasets. Of course, many methods and proposals are not yet implemented. Anybody who likes to contribute is cordially invited to do so. The very important field of neural networks has been skipped partially as Matlab already includes a very good toolbox in that area. At the moment just some basic routines based on that toolbox are included in order to facilitate a comparison with traditional techniques.

PRTools has a few limitations. Due to the heavy memory demands of Matlab very large problems with learning sets of tens of thousands of objects cannot be handled on moderate machines. Moreover, some algorithms are slow as it appeared to be difficult to avoid nested loops. A fundamental drawback with respect to some applications is that PRTools as yet does not offer the possibility of handling missing data problems, nor the use of fuzzy or symbolic data. These areas demand their own sets of routines and are waiting for manpower.

In the next sections, first the area of statistical pattern recognition covered by PRTools is described. Following the toolbox is summarized and details are given on some specific implementations. Finally some examples are presented.

2. The area of statistical pattern recognition

PRTools deals with sets of *labeled objects* and offers routines for generalising such sets into functions for *data mapping* and *classification*. An *object* is a k-dimensional vector of *feature values*. It is assumed that for all objects in a problem all values of the same set of features are given. The space defined by the actual set of features is called the *feature space*. Objects are represented as points or vectors in this space. A *classification function* assigns labels to new objects in the feature space. Usually, this is not done directly, but in a number of stages in which the initial feature space is successively mapped into intermediate stages, finally followed by a classification. The concept of *mapping* spaces and dataset is thereby important and constitutes the basis of many routines in the toolbox.

Sets of *objects* may be given externally or may be generated by one of the data generation routines of PRTools. Their *labels* may also be given externally or may be the result of a cluster analysis. By this technique similar objects within a larger set are grouped (clustered). The similarity measure is defined by the cluster technique in combination with the object representation in the feature space.

A fundamental problem is to find a good *distance measure* that agrees with the dissimilarity of the objects represented by the feature vectors. Throughout PRTools the Euclidean distance is used as default. However, scaling the features and transforming the feature spaces by different types of maps effectively changes the distance measure.

The *dimensionality of the feature space* may be reduced by the selection of subsets of good features. Several strategies and criterion functions are possible for searching good subsets. *Feature selection* is important because it decreases the amount of features that have to be measured and processed. In addition to the improved computational speed in lower dimensional feature spaces there might also be an increase in the accuracy of the classification algorithms. This is caused by the fact that for less features less parameters have to be estimated.

Another way to *reduce the dimensionality* is to *map* the data on a linear or nonlinear subspace. This is called linear or nonlinear feature extraction. It does not necessarily reduce the number of features to be measured, but the advantage of an increased accuracy might still be gained. Moreover, as lower dimensional representations yield less complex classifiers better generalisations can be obtained.

Using a *learning set* (or training set) a classifier can be trained such that it generalizes this set of examples of labeled objects into a *classification rule*. Such a classifier can be linear or nonlinear and can be based on two different kinds of strategies. The first one minimizes the expected classification error by using estimates of the *probability density functions*. In the second strategy this error is minimised directly by *optimizing the classification function* over its performance over the learning set. In this approach it has to be avoided that the classifier becomes entirely adapted to the learning set, including its noise. This decreases its generalisation capability. This ‘*overtraining*’ can be circumvented by several types over *regularisation* (often used in neural network training). Another technique is to simplify the classifiaction function afterwards (e.g. the pruning of decision trees).

If the class probability density functions are known like in simulations, the optimal classification function directly follows from the *Bayes rule*. In simulations this rule is often used as a reference.

Constructed classification functions may be evaluated by *independent test sets* of labeled objects. These objects have to be excluded from the learning set, otherwise the evaluation becomes biased. If they are added to the learning set, however, better classification function may be expected. A solution to this dilemma is the use of *cross validation* and *rotation* methods by which a small fraction of objects is excluded from learning and used for testing. This fraction is rotated over the available set of objects and results are avaraged. The extreme case is the *leave-one-out* method for which the excluded fraction is as large as one object.

The performance of classification functions can be improved by the following methods:
1. A *reject* option in which the objects close to the decision boundary are not classified. They are rejected and might be classified by hand or by another classifier.
2. The selection or averaging of classifiers.
3. A multi-stage classifier for *combining* classification results of several other classifiers.

For all these methods it is profitable or necessary that a classifier yields some distance measure or aposteriori probability in addition to the hard, unambiguous assignment of labels.

3. References

Yoh-Han Pao, *Adaptive pattern recognition and neural networks*, Addison-Wesley, Reading, Massachusetts, 1989.

K. Fukunaga, *Introduction to statistical pattern recognition*, second edition, Academic Press, New York, 1990.

S.M. Weiss and C.A. Kulikowski, *Computer systems that learn*, Morgan Kaufman Publishers, California, 1991.

C.M. Bishop, *Neural Networks for Pattern Recognition*, Clarendon Press, Oxford, 1995.

B.D. Ripley, *Pattern Recognition and Neural Networks*, Cambridge University Press, 1996.

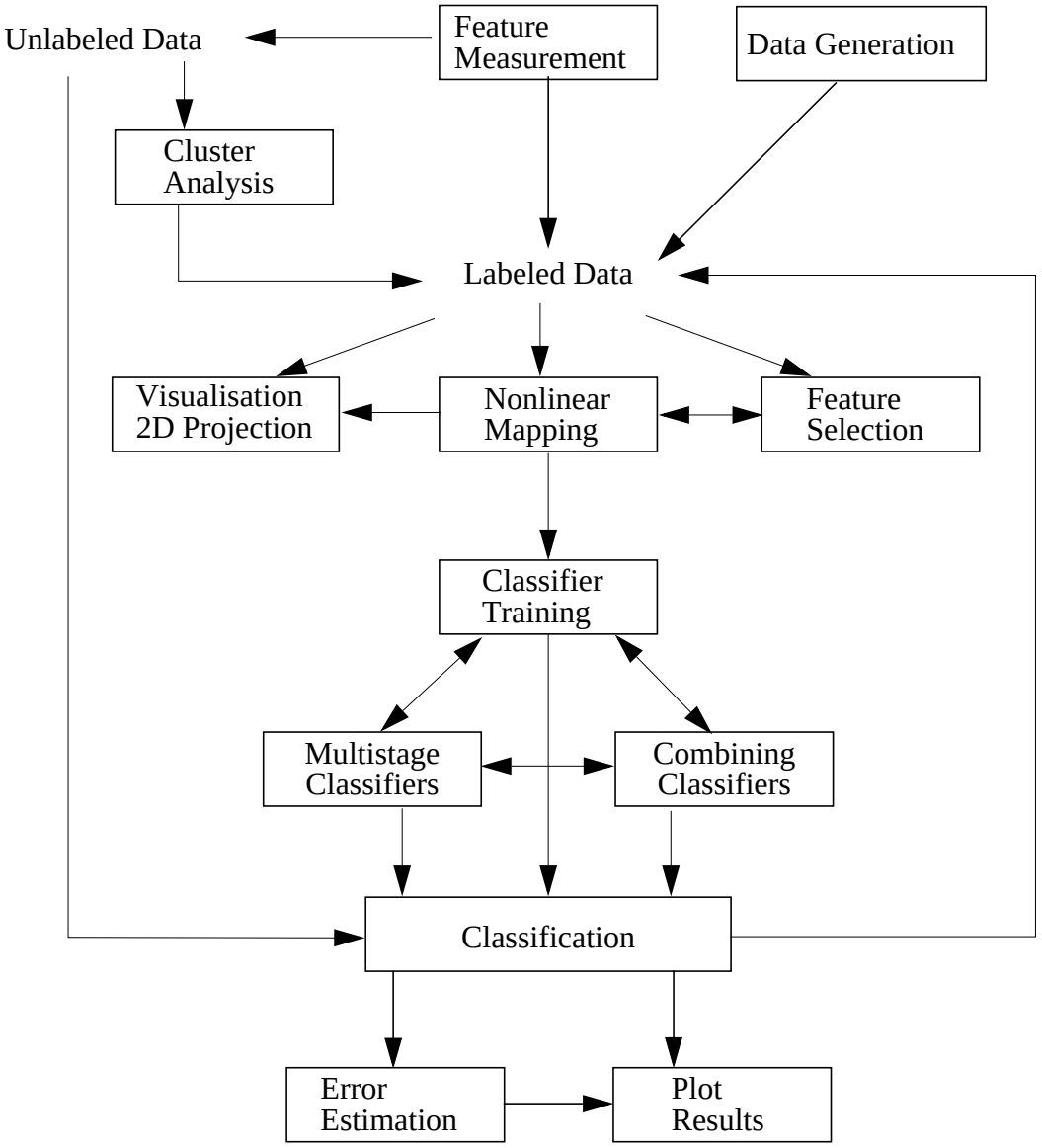
J. Schurmann, *Pattern classification, a unified view of statistical and neural approaches*, John Wiley & Sons, New York, 1996.

E. Gose, R. Johnsonbaugh and S. Jost, *Pattern recognition and image analysis*, Prentice-Hall, Englewood Cliffs, 1996

S. Haykin, *Neural Networks, a Comprehensive Foundation*, second edition, Prentice-Hall, Englewood Cliffs, 1999.

S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, Academic Press, New York, 1999.

4. A review of the toolbox



PRTools makes use of the possibility offered by Matlab 5 to define 'Classes' and 'Objects'. These programmatic concepts should not be confused with the *classes* and *objects* as defined in Pattern Recognition. Two 'Classes' have been defined: `dataset` and `mapping`. A large number of operators (like `* []`) and Matlab commands have been overloaded and have thereby a special meaning when applied to a `dataset` and/or a `mapping`.

The central data structure of PRTools is the `dataset`. It primarily consists of a set of objects represented by a matrix of feature vectors. Attached to this matrix is a set of labels, one for each object and a set of feature names. Moreover, a set of apriori probabilities, one for each class, is stored. In most help files of PRTools, a `dataset` is denoted by `A`. In almost any routine this is one of the inputs. Almost all routines can handle multiclass object sets.

In the above scheme the relations between the various sets of routines are given. At the moment

there are no commands for measuring features, so they have to be supplied externally. There are various ways to regroup the data, scale and transform the featurespace, find good features, build classifiers, estimate the classification performances and compute (new) object labels.

Data structures of the 'Class' `mapping` store trained classifiers, feature extracting results, data scaling definitions, nonlinear projections, etcetera. They are usually denoted by `W`. The result of the operation `A*W` is again a dataset. It is the classified, rescaled or mapped result of applying the mapping definition stored in `W` to `A`.

A typical example is given below:

```
A = gendath(100);  % Generate Highleyman's classes, 100 objects/class
                   % Training set C (20 objects / class)
                   % Test set D (80 objects / class)
[C,D] = gendat(A,20);
                   % Compute classifiers
W1 = ldc(C);       % linear
W2 = qdc(C);       % quadratic
W3 = parzenc(C);   % Parzen
W4 = bpxnc(C,3);   % Neural net with 3 hidden units
                   % Compute and display errors
disp([testd(D*W1),testd(D*W2),testd(D*W3),testd(D*W4)]);
                   % Plot data and classifiers
scatterd(A);       % scatter plot
plotd(W1,'-');     % plot the 4 discriminant functions
plotd(W2,'-.');
plotd(W3,'--');
plotd(W4,':');
```

This commandfile first generates by `gendath` two sets of labeled objects, both containing 100 two-dimensional object vectors, and stores them and their labels and apriori probabilities in the `dataset A`. The distribution follows the so-called 'Highleyman classes'. The next call to `gendat` takes this `dataset` and splits it at random into a `dataset C`, further on used for training, and a `dataset D`, used for testing. This training set `C` contains 20 objects from both classes. The remaining 2 x 80 objects are collected in `D`.

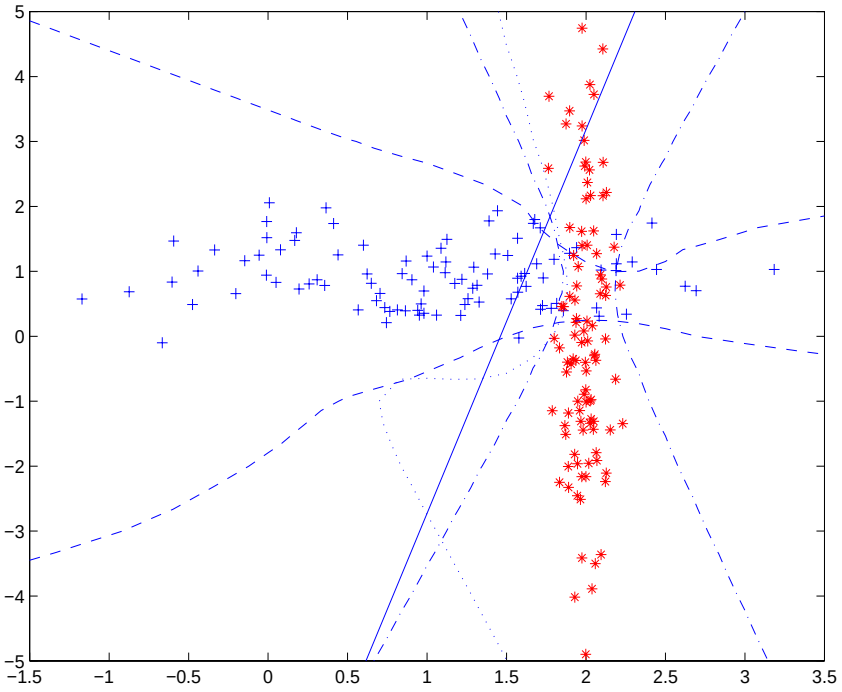
In the next lines four classification functions (discriminants) are computed, called `W1, W2, W3` and `W4`. The linear and quadratic classifier are both based on the assumption of normally distributed classes. The Parzen classifier estimates the class densities by the Parzen density estimation and has a built-in optimization for the smoothing parameter. The fourth classifier uses a feedforward neural network with three hidden units. It is trained by the backpropagation rule using a varying stepsize.

Hereafter the results are displayed and plotted. The test `dataset D` is used in a routine `testd` (test discriminant) on each of the four discriminants. The estimated probabilities of error are displayed in the Matlab command window and look like:

```
 0.1750 0.1062 0.1000 0.1562
```

Finally the classes are plotted in a scatter diagram together with the discriminants, see below. The plot routine `plotd` draws a vectorized straight line for the linear classifiers and computes the discriminant function values in all points of the plot grid (default 30 x 30) for the nonlinear discriminants. After that, the zero discriminant values are computed by interpolation and plotted.

:



We will now shortly discuss the PRTools commands group by group. The two basic structures of the toolbox can be defined by the constructors `dataset` and `mapping`. These commands can also be used to retrieve or redefine the data. It is thereby not necessary to use the general Matlab converter `struct()` for decomposing the structures. By `getlab` and `getfeat` the labels assigned to the objects and features can be found. The generation and handling of data is further facilitated by `genlab` and `renumlab`.

Datasets and Mappings	
`dataset`	Define dataset from datamatrix and labels and retrieve
`getlab`	Retrieve object labels from dataset
`getfeat`	Retrieve feature labels from dataset
`genlab`	Generate dataset labels
`renumlab`	Convert labels to numbers
`mapping`	Define mapping and classifier from data and retrieve
`getlab`	Retrieve labels assigned by a classifier

Data Generation	
gauss	Generation of multivariate Gaussian distributed data
gendat	Generation of subsets of a given data set
gendatb	Generation of banana shaped classes
gendatc	Generation of circular classes
gendatd	Generation of two difficult classes
gendath	Generation of Higleyman classes
gendatk	Nearest neighbour data generation
gendatl	Generation of Lithuanian classes
gendatm	Generation of many Gaussian distributed classes
gendatp	Parzen density data generation
gendats	Generation of two Gaussian distributed classes
gendatt	Generation of testset from given dataset
prdata	Read data from file and convert into a dataset

There is a large set of routines for the generation of arbitrary normally distributed classes (gauss), and for various specific problems (gendatc, gendatd, gendath, gendatm and gendats). There are two commands for enriching classes by noise injection (gendatk and gendatp). These are used for the general testset generator gendatt. A given dataset can be spit into a training set and a testset gendat. The routine gendat splits the dataset at random into two sets.

Linear and Higher Degree Polynnomial Classifiers	
klclc	Linear classifier by KL expansion of common cov matrix
kljlc	Linear classifier by KL expansion on the joint data
loglc	Logistic linear classifier
fisherc	Fisher's discriminant (minimum least square linear classifier)
ldc	Normal densities based linear classifier (Bayes' rule)
nmc	Nearest mean classifier
nmsc	Scaled nearest mean classifier
perlc	Linear classifier by linear perceptron
persc	Linear classifier by nonlinear perceptron
pfsvc	Pseudo-Fisher support vector classifier
qdc	Normal densities based quadratic (multi-class) classifier
udc	Uncorrelated normal densities based quadratic classifier
polyc	Add polynomial features and run arbitrary classifier
classc	Converts a mapping into a classifier
classd	General classification routine for trained classifiers
testd	General error estimation routine for trained classifiers

All routines operate in multi-class problems. classd and testd are the general classification and testing routines. They can handle any classifier from any routine, including the ones to

follow.

Nonlinear Classification	
knnc	k-nearest neighbour classifier (find k, build classifier)
mapk	k-nearest neighbour mapping routine
testk	Error estimation for k-nearest neighbour rule
parzenc	Parzen density based classifier
parzenml	Optimization of smoothing parameter in Parzen density estimation.
mapp	Parzen mapping routine
testp	Error estimation for Parzen classifier
edicon	Edit and condense training sets
treec	Construct binary decision tree classifier
classt	Classification with binary decision tree
bpxnc	Train feed forward neural network classifier by backpropagation
lmnc	Train feed forward neural network by Levenberg-Marquardt rule
rbnc	Train radial basis neural network classifier
neurc	Automatic neural network classifier
rnnc	Random neural network classifier
svc	Support vector classifier

knnc and parzenc are similar in the sense that the classifiers they build still include all training objects and that their parameter (the number of neighbours or the smoothing parameter) can be user supplied or can be optimized over the training set using a leave-one-out error estimation. For the Parzen classifier the smoothing parameter can also be estimated by parzenml using an optimization of the density estimation. The special purpose classification routines mapk, and mapp are called automatically when needed. In general, there is no need for the user to call them directly. The special purpose testing routines testk and testp are useful for obtaining leave-one-out error estimations.

Decision trees can be constructed by treec, using various criterion functions, stopping rules or pruning techniques. The resulting classifier can be used in classd, testd and plotd. They make use of classt.

PRTools offers three neural network classifiers(bpxnc, lmnnc and rbnnc) based on an old version of Matlab's Neural Network Toolbox. Adaptations of Mathwork's routines are made in order to prevent unnecessary display of intermediate results. They are stored in the private subdirectory. The resulting classifiers are ready to use by classd, testd and plotd. The automatic neural network classifier neurc builds a network without any parameter setting by the user. Random neural network classifiers can be generated by rnnc. The first one is totally random, the second optimizes the output layer by a linear classifier.

The Support Vector Classifier (svc) can be called for various kernels as defined by proxm (see

below). The classifier is optimized by a quadratic programming procedure. For some architectures a C-version is built-in for speeding up processing.

Normal Density Based Classification	
`distmaha`	Mahalanobis distance
`mapn`	Multiclass classification on normal densities
`meancov`	Estimation of means and covariance matrices from multiclass data
`nbayesc`	Bayes classifier for given normal densities
`ldc`	Normal densities based linear classifier (Bayes' rule)
`qdc`	Normal densities based quadratic classifier (Bayes' rule)
`udc`	Normal densities based quadratic classifier (independen features)
`testn`	Error estimate of discriminant on normal distributions

Classifiers for normal distributed classes can be trained by `ldc`, `qdc` and `udc`, while `nbayesc` assumes known densities. The special purpose testroutine `testn` can be used if the parameters of the normal distribution (means and covariances) are known or estimated by `meancov`.

Feature Selection	
`feateval`	Evaluation of a feature set
`featrank`	Ranking of individual feature permormances
`featselb`	Backward feature selection
`featself`	Forward feature selection
`featseli`	Individual feature selection
`featselo`	Branch and bound feature selection
`featselp`	Pudil's floating forward feature selection
`featselm`	Feature selection map, general routine for feature selection

The feature selection routines `featselb`, `featself`, `featseli`, `featselo` and `featselp` generate subsets of features, calling `feateval` for evaluating the feature set. `featselm` offers a general entry for feature selection, calling one of the other metohds. All routines produce a mapping `W` (e.g. `W = featself(A,[],k)`). So the reduction of a dataset `A` to `B` is done by `B = A*W`.

Classifiers and Tests (general)	
`classc`	Convert mapping to classifier
`classd`	General classification routine for trained classifiers
`cleval`	Classifier evaluation (learning curve)
`clevalb`	Classifier evaluation (learning curve), bootstrap version
`confmat`	Computation of confusion matrix
`crossval`	Error estimation by crossvalidation
`normc`	Normalisation of classifiers
`reject`	Compute error-reject curve
`roc`	Compute receiver-operator curve
`testd`	General error estimation routine for trained classifiers

A classifier maps, after training, objects from the feature space into its output space. The dimensionality of this space equals the number of classes (an exception is possible for two-class classifiers, that may have a one-dimensional output space). This output space is mapped on posterior probabilities by `classc`. Normalization of these probabilities on a given dataset can be controlled by `normc`. This is standard built-in for all training algorithms. Classification (determining the class with maximum output) is done by `classd`, error estimates for test data are made by `testd` and `confmat`. More advanced techniques like rotating datasets over test sets and training sets, are offered by `crossval`, `cleval` and `clevalb`.

Mappings	
`cmapm`	Compute some special maps
`featselm`	Feature selection map, general routine for feature selection
`fisherm`	Fisher mapping
`klm`	Decorrelation and Karhunen Loeve mapping (PCA)
`klms`	Scaled version of `klm`, useful for prewhitening
`lmnm`	Levenberg-Marquardt neural net diabolo mapping
`nlklm`	Nonlinear Karhunen Loeve mapping (NL-PCA)
`normm`	Object normalization map
`proxm`	Proximity mapping and kernel construction
`reducm`	Reduce to minimal space mapping
`scalem`	Compute scaling data
`sigm`	Simoid mapping
`subsm`	Subspace mapping
`svm`	Support vector mapping, useful for kernel PCA

Classifiers are a special type of mapping, as their output spaces are related to class membership. In general a mapping converts data from one space to another. This may be done by a fixed procedure, not depending on a dataset, but controlled by at most some parameters. Most of these mappings that don't need training are collected by `cmapm` (e.g. shifting, rotation, deletion of particular features), another example is the sigmoidal mapping `sigm`. Some of the mappings that need training don't depend on the object labels, e.g. the principal component analysis (PCA) by `klm` and `klms`, object normalization by `normm` and scaling by `scalem`, subspace mapping (maps defined by normalized objects and that include the origin) by `subsm` and nonlinear PCA or kernel PCA by support vector mapping, `svm`. The other routines depend on object labels as they define the mapping such that the class separability is maximized in one way or another. The Fisher criterion is optimized by `fisherm`, the scatter by `klm` (if called by labelled data), density overlap for normal distributions by `mlm` and general class separability by `lmnm`.

Combining classification rules	
baggingc	Boortstrapping and aggregation of classifiers
majorc	Majority voting combining classifier
maxc	Maximum combining classifier
minc	Minimum combining classifier
meanc	Averaging combining classifier
medianc	Median combining classifier
prodc	Product combining classifier
traincc	Train combining classifier
parsc	Parse classifier or map

Classifiers can be combined by horizontal and vertical concatenation, see section 5.2, e.g. `W = [W1, W2, W3]`. Such a set of classifiers can be combined by several rules, like majority voting (`majorc`), combining the posterior probabilities in several ways (`maxc`, `minc`, `meanc`, `medianc` and `prodc`), or by training an output classifier (`traincc`). The way classifiers are combined can be inspected by `parsc`.

dataim	Image operation on dataset images.
data2im	Convert dataset to image
datfilt	Filter dataset image
datgauss	Filter dataset image by Gaussian filter
im2obj	Convert image to object in dataset
im2feat	Convert image to feature in dataset
image	Display images stored in dataset
imagesc	Display images stored in dataset, automatic scaling

Images can be stored, either as features (`im2feat`), or as objects (`im2obj`) in a dataset. The first possibility is useful for segmenting images using a vector of values for each pixels (e.g. in case of multi-color images, or as a result of a filterbank). The second possibility enables the classification of entire images using their pixels as features. Such datasets can be displayed by the overloaded commands `image` and `imagesc`. The relation with image processing is established by `dataim`, enabling arbitrary image operations, Simple filtering can be sped up by the use of `datfilt` and `datgauss`.

Clustering and Distances	
distm	Distance matrix between two data sets.
proxm	Proximity mapping and kernel construction
hclust	Hierarchical clustering
kcentres	k-centers clustering
kmeans	k-means clustering
modeseek	Clustering by modeseeking

Plotting	
`plotd`	Plot discriminant function in scatterplot
`plotf`	Plot feature distribution
`plotm`	Plot mapping in scatterplot
`plot2`	Plot 2d function
`plotdg`	Plot dendrgram (see hclust)
`scatterd`	Scatterplot
`scatter3d`	3D Scatterplot

Examples	
`prex1`	Classifiers and scatter plot
`prex2`	Plot learning curves of classifiers
`prex3`	Multi-class classifier plot
`prex4`	Classifier combining
`prex5`	Use of images and eigenfaces

5. Some Details

The command help files and the examples given below should give sufficient information to use the toolbox with a few exeptions. These are discussed in the following sections. They deal with the ways classifiers and mappings are represented. As these are the constituting elements of a pattern recognition analysis, it is important that the user understands these issues.

5.1 Datasets

A `dataset` consists of a set of `m` objects, each given by `k` features. In PRTools such a dataset is represented by a `m` by `k` matrix: `m` rows, each containing an object vector of `k` elements. Usually a dataset is labeled. An example of a definition is:

```
> A = dataset([1 2 3; 2 3 4; 3 4 5; 4 5 6],[3 3 5 5]')
> 4 by 3 dataset with 2 classes
```

The 4 by 3 data matrix (4 objects given by 3 features) is accompanied by a labellist of 4 labels, connecting each of the objects to one of the two classes, 3 and 5. Class labels can be numbers or strings and should always be given as rows in the labellist. If the labellist is not given all objects are given the default label 255. In addition it is possible to assign labels to the columns (features) of a dataset:

```
> A = dataset(rand(100,3),genlab([50 50],[3 5]'),['r1';'r2';'r3'])
> 100 by 3 dataset with 2 classes
```

The routine genlab generates 50 labels with value 3, followed by 50 labels with value 5. In the last term the labels (r1, r2, r3) for the three features are set. The complete definition of a dataset is:

```
> A = dataset(datamatrix,labels,featlist,prob,lablist)
```

given the possibilitiy to set apriori probabilities for each of the classes as defined by the labels given in `lablist`. The values in `prob` should sum to one. If prob is empty or if it is not supplied the apriori probabilities are computed from the dataset label frequencies. If prob = 0 then equal class probabilities are assumed.

Various items stored in a dataset can be retrieved by

```
> [nlab,lablist,m,k,c,prob,featlist] = dataset(A)
```

in which nlab are numeric labels for the objects (1, 2, 3, ...) referring to the true labels stored in the rows of lablist. The size of the dataset is m by k, c is the number of classes (equal to max(nlab)). Datasets can be combined by [A; B] if A and B have equal numbers of features and by [A B] if they have equal numbers of objects. Creating subsets of datasets can be done by A(I,J) in which I is a set of indices defining the desired objects and J is a set of indices defining the desired features. In all these examples the apriori probabilities set for A remain unchanged.

The original datamatrix can be retireved by double(A) or by +A. The labels in the objects of A can be retrieved labels = getlab(A), which is equivalent to labels = lablist(nlab,:). The feature labels can be retrieved by featlist = getfeat(A). Conversion by struct(A) makes all fields in a dataset A accessible to the user.

5.2 Classifiers and mappings

There are many commands to train and use mappings between spaces of different (or equal) dimensionalities. For example:

if `A` is a `m` by `k` dataset (`m` objects in a `k`-dimensional space)
and `W` is a `k` by `n` mapping (map from `k` to `n` dimensions)
then `A*W` is a `m` by `n` dataset (`m` objects in a `n`-dimensional space)

Mappings can be linear (e.g. a rotation) as well as nonlinear (e.g. a neural network). Typically they can be used for classifiers. In that case a `k` by `n` mapping maps a `k`-feature data vector on the output space of a `n`-class classifier (exception: 2-class classifiers like discriminant functions may be implemented by a mapping to a 1-dimensional space like the distance to the discriminant, `n = 1`).

Mappings are of the datatype '`mapping`' (`class(W)` is '`mapping`'), have a size of `[k,n]` if they map from `k` to `n` dimensions. Mappings can be instructed to assign labels to the output columns, e.g. the class names. These labels can be retrieved by

`labels = getlab(W);` before the mapping, or
`labels = getlab(A*W);` after the dataset `A` is mapped by `W`.

Mappings can be learned from examples, (labeled) objects stored in a dataset `A`, for instance by training a classifier:

`W3 = ldc(A);`	the normal densities based linear classifier
`W2 = knnc(A,3);`	the 3-nearest neighbor rule
`W1 = svc(A,'p',2);`	the support vector classifier based on a 2-nd order polynomial kernel

Untrained or empty mappings are supported. They may be very useful. In this case the dataset is replaced by an empty set or entirely skipped:

```
V1 = ldc; V2 = knnc([],a); V3 = svc([],'p',2);
```

Such mappings can be trained later by

```
W1 = A*V1; W2 = A*V2; W3 = A*V3;
```

The mapping of a testset `B` by `B*W1` is now equivalent to `B*(A*V1)` or even, irregulary but very handy to `A*V1*B` (or even `A*ldc*B`). Note that expressions are evaluated from left to right, so `B*A*V1` may result in an error as the multiplication of the two datasets (`B*A`) is executed first.

Users can add new mappings or classifiers by a single routine that should support the following type of calls:

`W = newmapm([], par1, par2, ...);` Defines the untrained, empty mapping.
`W = newmapm(A, par1, par2, ...);` Defines the map based on the training dataset `A`.
`B = newmapm(A, W);` Defines the mapping of dataset `A` on `W`, resulting in a dataset `B`.

For an example list the routine `subsc.m`.

Some trainable mappings do not depend on class labels and can be interpreted as finding a space that approximates as good as possible the original dataset given some conditions and measures. Examples are the Karhunen-Loeve Mapping (`klm`) which may be used for PCA and the Support Vector Mapping (`svm`) by which nonlinear, kernel PCA mappings can be computed.

In addition to trainable mappings, there are fixed mappings, which operation is not computed from a trainingset but defined by just a few parameters. Most of them can be set by `cmapm`.

The result `D` of a mapping of a testset on a trained classifier, `D = B*W1` is again a dataset, storing for each object in `B` the output values of the classifier. These values, usually between `-inf` and `inf` can be interpreted as similarities: the larger, the more similar with the corresponding class. These number can be mapped on the `[0,1]` interval by the fixed mapping `sigm`: `D = B*W1*sigm`. The values in a single row (object) don't necessarily sum to one. This can be achieved by the fixed mapping normm: `D = B*W1*sigm*normm` which is equivalent to `B*W1*classc`. Effectively a mapping `W` is converted into a classifier by `W*classc`, which maps objects on the normalized `[0,1]` output space. Usually a mapping that can be converted into a classifier in this way, is scaled such by a multiplicative constant that these numbers optimally represent (in the maximum likelihood sense) the posterior probabilities for the training data. The resulting output dataset D has column labels for the classes and row labels for the objects. The class labels of the maximum values for each object can be retrieved by `labels = D*classd;` or `labels = classd(D);` A global classification error follows from `e = D*testd;` or `e = testd(D);`

Mappings can be combined in the following ways:

sequential: `W = W1 * W2 * W3` (equal inner dimensions)
stacked : `W = [W1, W2, W3]` (equal numbers of 'rows' (input dimensions))
parallel : `W = [W1; W2 ;W3]` (unrestricted)

The output size of the parallel mapping is irregulary equal to `(k1+k2+k3)` by`(n1+n2+n3)` as the output combining of columns is undefined. In a stacked or parallel mapping columns having the same label can be combined by various combiners like `maxc`, `meanc` and `prodc`. If the classifiers `W1, W2` and `W3` are trained for the same `n` classes, their output labels are the same and are combined by `W = prodc([W1;W2;W3])` into a `(k1+k2+k3)` by `n` classifier.

`W` for itself, or `display(W)` lists the size and type of a classifier as well as the routine or section in `@mapping/mtimes` used for computing a mapping `A*W`. The construction of a combined mapping may be inspected by `parsc(W)`.

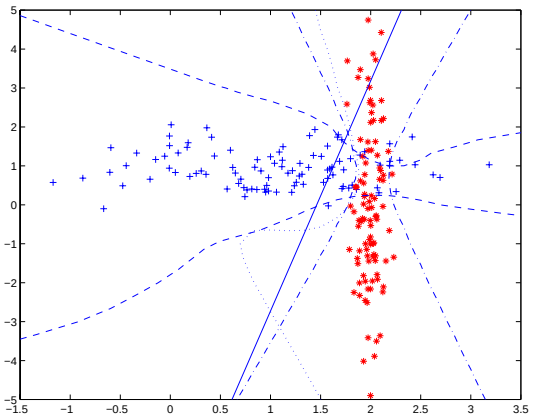
A mapping may be given an output selection by `W = W(:,J)`, in which `J` is a set of indices pointing to the desired classes. `B = A*W(:,J);` is equivalent to `B = A*W; B = B(:,J);` Input selection is not possible for a mapping.

6. Examples

The following examples are available under PRTools. We present here the source codes and the output they generate.

6.1 Classifiers and scatter plot

A 2-d Highleyman dataset `A` is generated, 100 objects for each class. Out of each class 20 objects are generated for training, `C` and 80 for testing, `D`. Four classifiers are computed: a linear one and a quadratic one, both assuming normal densities (which is correct in this case), a Parzen classifier and a neural network with 3 hidden units. Note that the data generation as well as the neural network initialisation use random generators. As a result they only reproduce if they use the original seed. After computing and displaying classification results for the test set a scatterplot is made in which all classifiers are drawn.



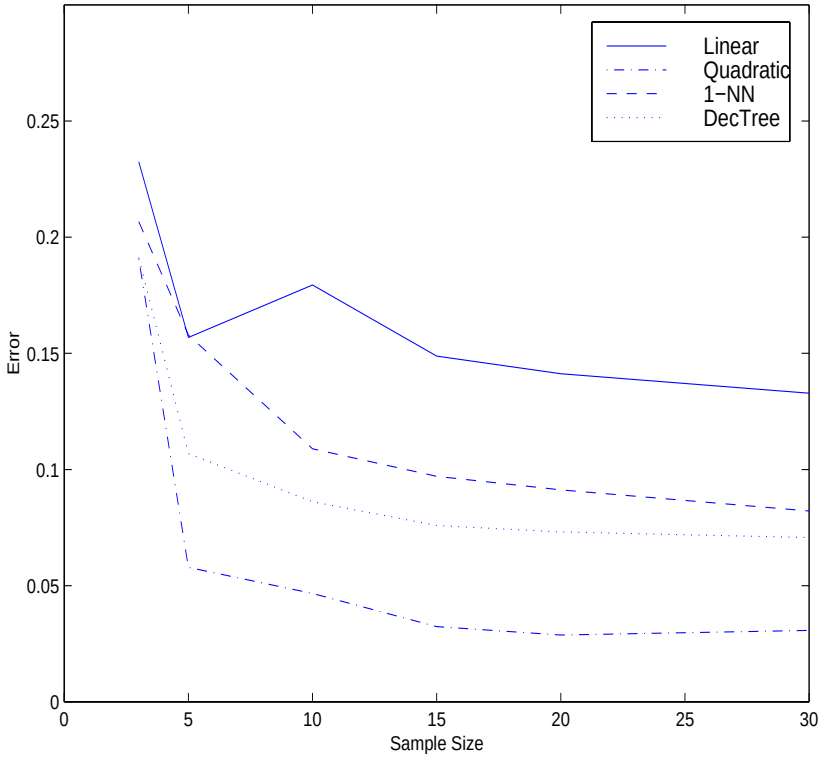
```
%PREX1 PRTools example of classifiers and scatter plot
help prex1
pause(1)
A = gendath(100,100); % Generate Highleyman's classes
                      % Training set c (20 objects / class)
                      % Test set d (80 objects / class)
[C,D] = gendat(A,20);
                      % Compute classifiers
w1 = ldc(C);          % linear
w2 = qdc(C);          % quadratic
w3 = parzenc(C);      % Parzen
w4 = lmnc(C,3);       % Neural Net
                      % Compute and display errors
disp([testd(D*w1),testd(D*w2),testd(D*w3),testd(D*w4)]);
                      % Plot data and classifiers
figure(1);
hold off;
scatterd(A); drawnow;
plotd(w1,'-'); drawnow;
plotd(w2,'-.'); drawnow;
plotd(w3,'--'); drawnow;
plotd(w4,':'); drawnow;
echo off
0.1875    0.0500    0.1437    0.0938
```

6.2 Learning curves

In this example the learning curves for four classifiers are computed using the Highleyman

dataset. The errors are computed using the `cleval` routine.

```
%PREX2 PRTools example, plot learning curves of classifiers
help prex2
pause(1)
                    % set desired learning sizes
learnsize = [3 5 10 15 20 30];
                    % Generate Highleyman's classes
A = gendath(100,100);
                    % avarage error over 10 repetitions
                    % testset is complement of training set
e1 = cleval(ldc,A,learnsize,10);
figure(1); hold off;
plot(learnsize,e1(1,:),'-');
axis([0 30 0 0.3]); hold on; drawnow;
e2 = cleval(qdc,A,learnsize,10);
plot(learnsize,e2(1,:),'-.'); drawnow;
e3 = cleval(knnc([],1),A,learnsize,10);
plot(learnsize,e3(1,:),'--'); drawnow;
e4 = cleval(treec,A,learnsize,10);
plot(learnsize,e4(1,:),':');  drawnow;
legend('Linear','Quadratic','1-NN','DecTree');
xlabel('Sample Size')
ylabel('Error');
```

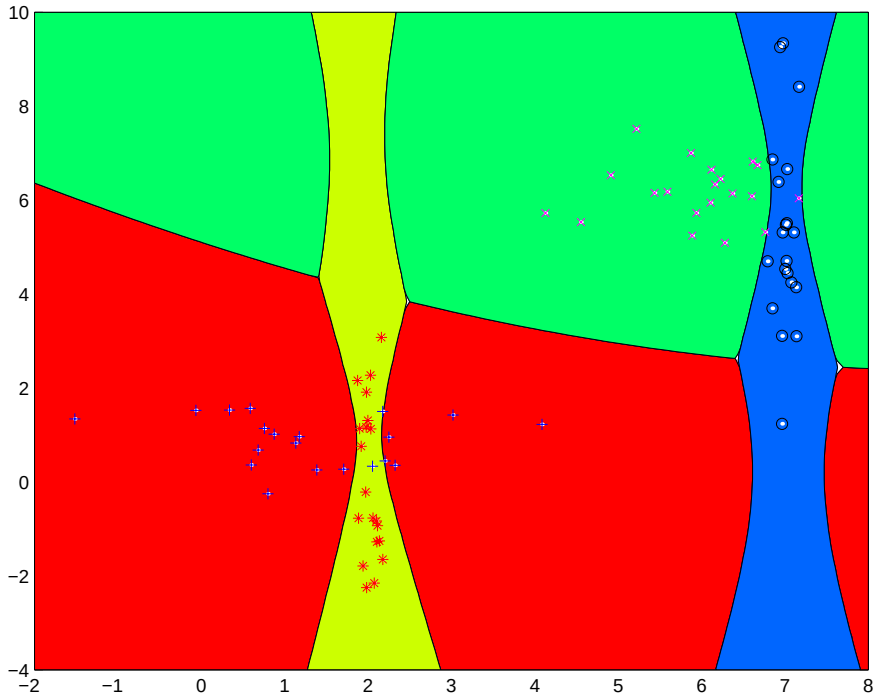


6.3 Multi-class classifier plot

This file shows how to construct a colored scatter diagram defining the areas assigned to the various classes. First the global variable GRIDSIZE is set to 100 in order to avoid empty areas.

Then the Highleyman dataset is used to construct a 4-class problem. This is done by using the data only and then generating the labels separately. Note that the scatter plot itself is called twice in order to have the scatter on top of the color plot generated by `plotd`.

```
%PREX3 PRTools example of multi-class classifier plot
help prex3
echo on
global GRIDSIZE
gs = GRIDSIZE;
GRIDSIZE = 100;
                    % generate 2 x 2 normal distributed classes
a = +gendath(20);   % data only
b = +gendath(20);   % data only
A = [a; b + 5];     % shift 2 over [5,5]
lab = genlab([20 20 20 20],[1 2 3 4]');% generate 4-class labels
A = dataset(A,lab);% construct dataset
hold off;           % clear figure
scatterd(A,'.'); drawnow;% make scatter plot for right size
w = qdc(A);         % compute normal densities based classifier
plotd(w,'col'); drawnow;% plot classification regions
hold on;
scatterd(A);        % redraw scatter plot
echo off
GRIDSIZE = gs;
.
```



6.4 Classifier combining

This example is just an illustration on the use of mapping and classifier combining. The method itself does not make much sense. There are sequential maps (like `w1 = wkl*vkl`) and a stacked

map (`wall = [w1,w2,w3,w4,w5]`), using various combining rules. Note how in the feature selection routine `featself` a classifier (`ldc`) is used for the criterion.

```
%PREX4 PRTools example of classifier combining
help prex4
echo on

A = gendatd(100,100,10);
[B,C] = gendat(A,20);

wkl = klm(B,0.95);        % find KL mapping input space
bkl = B*wkl;              % map training set
vkl = ldc(bkl);           % find classifier in mapped space
w1 = wkl*vkl;             % combine map and classifier
                          % (operates in original space)
testd(C*w1)               % test

wfn = featself(B,'NN',3); % find feature selection mapping
bfn = B*wfn;              % map training set
vfn = ldc(bfn);           % find classifier in mapped space
w2 = wfn*vfn;             % combine
testd(C*w2)               % test

wfm = featself(B,ldc,3); % find second feature set
bfm = B*wfm;              % map training set
vfm = ldc(bfm);           % find classifier in mapped space
w3 = wfm*vfm;             % combine
testd(C*w3)               % test

w4 = ldc(B);              % find classifier in input space
testd(C*w4)               % test
w5 = knnc(B,1);           % another classifier in input space
testd(C*w5)               % test

wall = [w1,w2,w3,w4,w5];  % parallel classifier set
testd(C*prodc(wall))      % test product rule
testd(C*meanc(wall))      % test mean rule
testd(C*medianc(wall))    % test median rule
testd(C*maxc(wall))       % test maximum rule again
testd(C*minc(wall))       % test minimum rule
testd(C*majorc(wall))     % test majority voting

echo off
```

6.5 Image segmentation by vector quantization

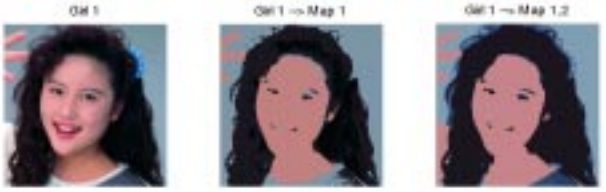
In this example an images is segmented using modeseeking clustering techniques based on an randomly selected subset of pixels. The resulting classifier is applied on all pixels, and also on the pixels of a second image (may apply not so good). Finally a common map is computed and applied for both images.

```
%PREX5 PRTOOLS example of image vector quantization
help prex5
echo on
     % standard Matlab TIFF read
girl = imread('girl.tif','tiff');
     % display
figure
subplot(2,3,1); subimage(girl); axis off;
title('Girl 1'); drawnow
     % construct 3-feature dataset from entire image
%[X,Y] = meshgrid(1:256,1:256);
%X = X/10000;
%Y = Y/10000;
%girl(:,:,4) = X;
%girl(:,:,5) = Y;
g1 = im2dfeat(girl);
imheight = size(girl,1);
     % generate testset
t = gendat(g1,250);
     % run modeseek, find labels, and construct labeled dataset
labt = modeseek(t*proxm(t),25);
t= dataset(t,labt);
     % train NMC classifier
w = t*qdc([],1e-6,1e-6);
     % classify all pixels
pack
lab = g1*w*classd;
     % show result
     % substitute class means for colors
cmap = +meancov(t(:,1:3));
subplot(2,3,2); subimage(reshape(lab,imheight,length(lab)/im-
height),cmap);
axis off;
title('Girl 1 --> Map 1')
drawnow

     % Now, read second image

girl2 = imread('girl2.tif','tiff');
     % display
subplot(2,3,4); subimage(girl2);
```

```
%girl2(:,:,4) = X;
%girl2(:,:,5) = Y;
axis off;
title('Girl 2'); drawnow
     % construct 3-feature dataset from entire image
g2 = im2feat(girl2);
clear girl girl2
pack
lab2 = g2*w*classd;
     % show result
     % substitute class means for colors
cmap = +meancov(t(:,1:3));
subplot(2,3,5);
subimage(reshape(lab2,imheight,length(lab)/imheight),cmap);
axis off;
title('Girl 2 --> Map 1')
drawnow
```



```
     % Compute combined map

g = [g1; g2];
t = gendat(g,250);
labt = modeseek(t*proxm(t),25);
t= dataset(t,labt);
w = t*qdc([],1e-6,1e-6);
cmap = +meancov(t(:,1:3));
clear g
pack
lab = g1*w*classd;
```

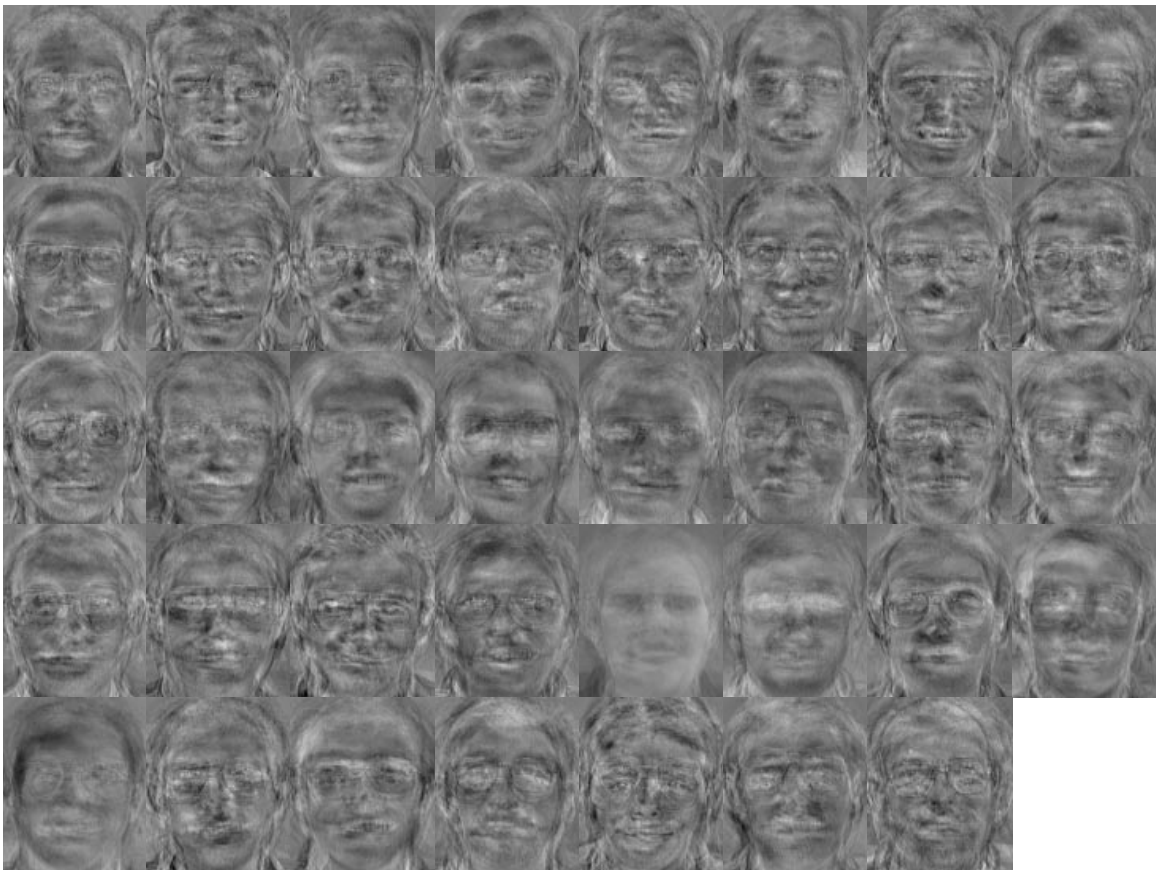
```
subplot(2,3,3);
subimage(reshape(lab,imheight,length(lab)/imheight),cmap);
axis off;
title('Girl 1 --> Map 1,2')
drawnow
pack
lab = g2*w*classd;
subplot(2,3,6); subimage(reshape(lab,imheight,length(lab)/im-
height),cmap);
axis off;
title('Girl 2 --> Map 1,2')
drawnow
set(gcf,'DefaultAxesVisible','remove')
```

6.6 Use of images and eigenfaces

This example illustrates the use of images by the face image dataset. The eigenfaces based on the first image of each subject are displayed. Next all images are mapped on this eigenspace, the scatterplot for the first two eigenfaces are displayed and the leave-one-out error is computed as a function of the number of eigenfaces used.

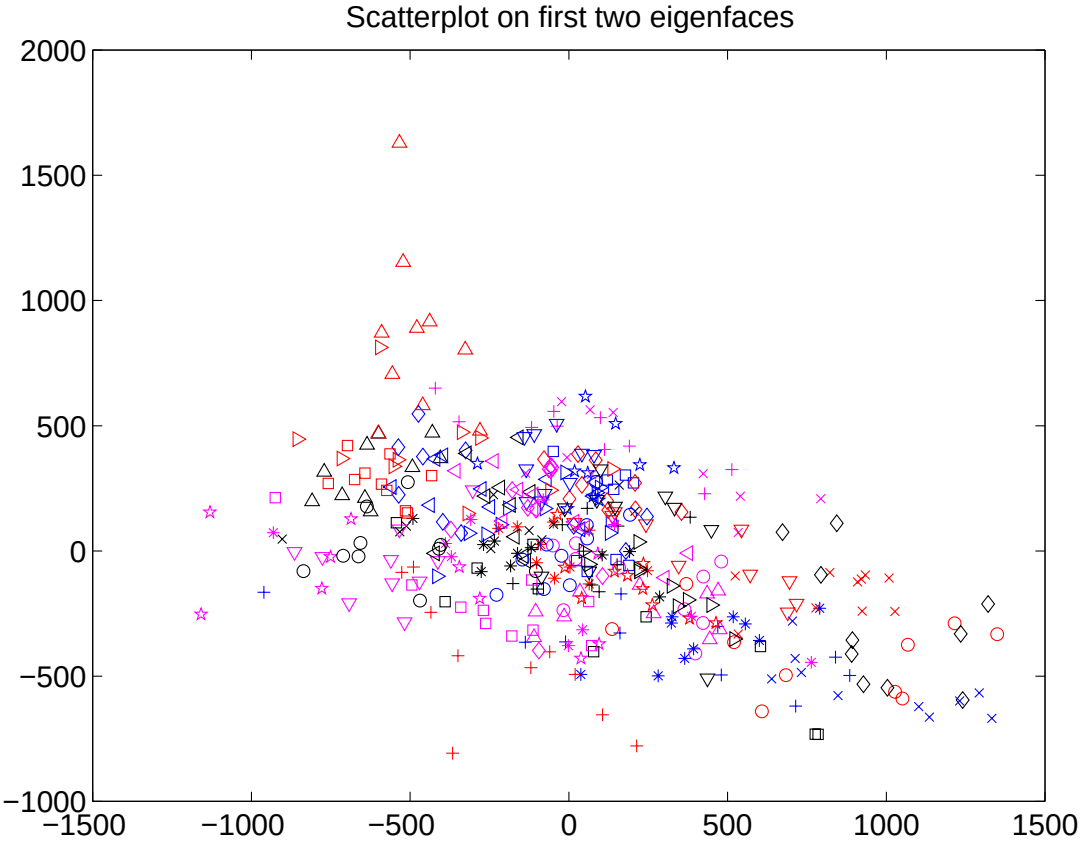
```
%PREX6 Use of images and eigenfaces
help prex6
echo on

if exist('face1.mat') ~= 2
 error('Face database not in search path')
end
a = readface([1:40],1);
w = klm(a);
imagesc(dataset(eye(39)*w',[],[],[],[],112)); drawnow
```



```
b = [];
for j = 1:40
     a = readface(j,[1:10]);
     b = [b;a*w];
end
figure
scatterd(b)
```

```
title('Scatterplot on first two eigenfaces')
fontsize(14)
```




```
featsizes = [1 2 3 5 7 10 15 20 30 39];
e = zeros(1,length(featsizes));
for j = 1:length(featsizes)
     k = featsizes(j);
     e(j) = testk(b(:,1:k),1);
end
figure
plot(featsizes,e)
xlabel('Number of eigenfaces')
ylabel('Error')
fontsize(14)
```