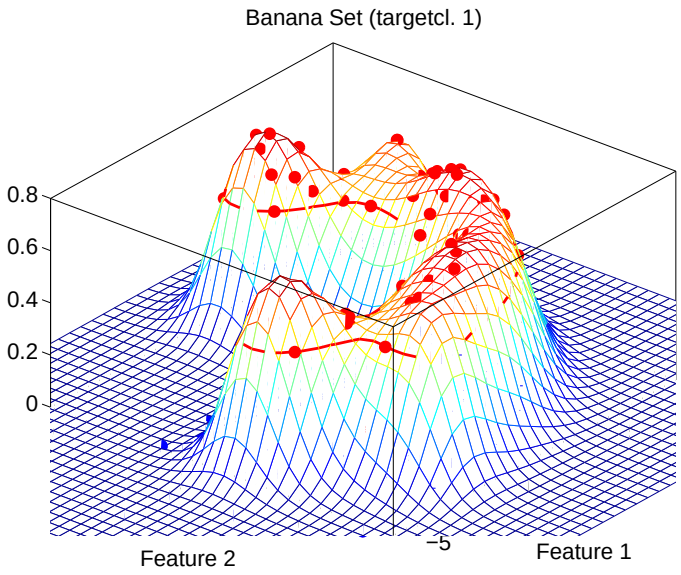
Data description toolbox

dd_tools 1.6.3

A Matlab toolbox for data description, outlier and novelty detection

September 24, 2008

D.M.J. Tax



Contents

Chapter 1

This manual

The dd_tools Matlab toolbox provides tools, classifiers and evaluation functions for the research of one-class classification (or data description). The dd_tools toolbox is an extension of the Prtools toolbox in which Matlab objects for mapping and dataset are defined. dd_tools uses these objects and their methods, but extends (and sometimes restricts) them to one-class classification. This means that before you can use dd_tools to its full potential, you need to know a bit about Prtools. When you are completely new to pattern recognition, Matlab or Prtools, please familiarize yourself a bit with them first (see http://www.prtools.org for more information on Prtools).

This short document should give the reader some idea what the data description toolbox (dd_tools) for Prtools offers. It provides some background information about one-class classification, about some implementation issues and it gives some practical examples. It does not try to be complete, though, because each new version of the dd_tools will probably include new commands and possibilities. The file Contents.m in the dd_tools-directory gives the up-to-date list of all functions and classifiers in the toolbox. The most up-to-date information can be found on the webpage on dd_tools, currently at: http://www-ict.ewi.tudelft.nl/~davidt/dd_tools.html

Note, that this is *not* a cookbook, solving all your problems. It should point out the basic philosophy of the dd_tools . You should always have a look at the help provided by each command (try help dd_tools). They should show all possible combinations of parameter arguments and output arguments. When a parameter is listed in the Matlab code, but not in the help, it often indicates an undocumented feature, which means: be careful!

Then I'm not 100% sure if it will work, how useful it is and if it will survive a next `dd_tools` version.
In chapter 2 a basic introduction about one-class classification/novelty detection/outlier detection is given. What is the goal, and how is the performance measured. You can skip that if you're familiar with one-class classification. In chapter 2.4 the basic idea of the `dd_tools` is given. Then in chapters 3 and 4 the specific use of datasets and classifiers is shown. In chapter 5 the computation of the error is explained, and finally in 6 some general remarks are given.

Chapter 2

Introduction

2.1 What is one-class classification?

The problem of one-class classification is a special type of classification problem. In one-class classification we are always dealing with a two-class classification problem, where each of the two classes have a special meaning. The two classes are called the *target* and the *outlier* class respectively:

target class : this class is assumed to be sampled well, in the sense that of this class many (training) example objects are available. It does not necessarily mean that the sampling of the training set is done completely according to the target distribution found in practice. It might be that the user sampled the target class according to his/her idea of how representative these objects are. It is assumed though, that the training data reflect the area that the target data covers in the feature space.

outlier class : this class can be sampled very sparsely, or can be totally absent. It might be that this class is very hard to measure, or it might be very expensive to do the measurements on these types of objects. In principle, a one-class classifier should be able to work, solely on the basis of target examples. Another extreme case is also possible, when the outliers are so abundant that a good sampling of the outliers is not possible.

An example of a one-class classification problem is the problem of machine diagnostics. Of a running machine it should be determined if the machine is

in a healthy operation condition, or that a faulty situation is occurring. It is relatively cheap and simple to obtain measurements from a normally working machine (although sampling from *all* possible possible normal situations might still be extensive). On the other hand, the sampling from the faulty situations will require that the machine have to damaged in several ways to obtain faulty measurement examples. The creating of a balanced training set for this example will therefore be very expensive, or completely impractical. Another example of one-class classification might be a detection problem. Here the task is to detect a specific target object (for instance faces) in some unspecified environment (for instance an image database, or surveillance camera recordings). In this problem the target class is relatively well defined, but the other class can be anything. Although in these types of problems it is often cheap to sample from the outlier class, the number of possibilities is so huge, that the chance of finding interesting objects, or objects near the target objects, is very small. To train a standard two-class classifier for this case will probably end up with a very high number of false positive detections.

2.2 Error minimization in one-class

In order to find a good one-class classifier, two types of errors have to be minimized, namely the fraction false positives and the fraction false negatives. In table 2.1 all possible classification situations for one-class classification are shown.

Table 2.1: Types of classification error in the one-class classification problem.

		true class label	
		target	outlier
assigned label	target	true positive target accepted	false positive outlier accepted
	outlier	false negative target rejected	true negative outlier rejected

The fraction false negative can be estimated using (for instance) cross-validation on the target training set. Unfortunately, the fraction false negative is much harder to estimate. When *no* example outlier objects are available, this fraction cannot be estimated. Minimizing just the fraction false negative, will result in a classifier which labels *all* object as target object. In order to avoid this degenerate solution, outlier examples have to be available, or artificial outliers have to be generated (see also section 5.5).

2.3 Receiver Operating Characteristic curve

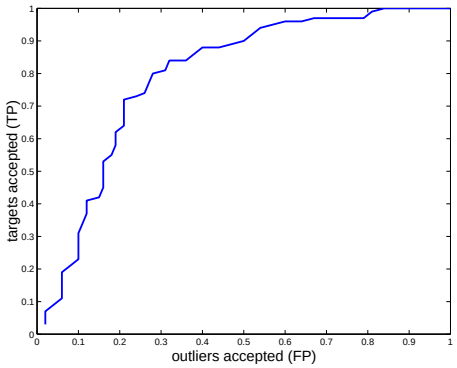


Figure 2.1: Example of a ROC curve.

A good one-class classifier will have both a small fraction false negative as a small fraction false positive. Because the error on the target class can be estimated (relatively) well, it is assumed that for all one-class classifiers a threshold can be set beforehand on the target error. By varying this threshold, and measuring the error on the (maybe artificial) outlier objects, an Receiver Operating Characteristics curve (ROC-curve) is obtained. This curve shows how the fraction false positive varies for varying fraction false negative. The smaller these fractions are, the more this one-class classifier is to be preferred. Traditionally the fraction true positive is plotted versus the fraction false positive, as shown in figure 2.1.
Although the ROC curve gives a very good summary of the performance of a one-class classifier, it is hard to compare two ROC curves. One way to summarize a ROC-curve in a single number, is the Area-under-the-ROC-

curve, AUC. This integrates the fraction true positive over varying thresholds (or equivalently, varying fraction false positive). Higher values indicate a better separation between target and outlier objects.
Note that for the actual application of a one-class classifier a specific threshold (or fraction false negative) has to be chosen. That means, that only a single point of the ROC-curve is used. It can therefore happen that for a specific threshold a one-class classifier with a lower AUC might be preferred over another classifier with a higher AUC. It just means that for that specific threshold, the fraction false positive is smaller for the first classifier than the second classifier.
In practical applications, the specific operation point on the ROC curve will not be known at the time of the training of the classifier. In many cases some range of reasonable false positives or false negatives can be given. It is therefore common to restrict the integration range for the AUC over this specific range. This will result in a more honest comparison between different classifiers for this application at hand. The toolbox therefore offers the possibility to compute the AUC over a limited integration range.

2.4 Introduction `dd_tools`

In `dd_tools` it is possible to define special one-class datasets and one-class classifiers. Furthermore, the toolbox provides methods for generating artificial outliers, estimating the different errors the classifiers make (false positive and false negative errors), estimating the ROC curve, the AUC (Area under the ROC curve) error, the AUC over a limited integration domain and many classifiers.
This is reflected in the setup of this manual. Each of the ingredients are discussed in a separate section:

1. One-class datasets,
2. One-class classifiers,
3. One-class error evaluation or model selection.

Before you can use the toolbox, you have to use `Prtools` and you have to put your data into a special `dataset`-format. Let us first start with the data.

Chapter 3

Datasets

3.1 Creating one-class datasets

The first and most important thing for the application of one-class classifiers, is the dataset and its preprocessing. All one-class classifiers require a Prtools `dataset` with objects labeled `target` or `outlier`. To create a one-class dataset, several functions are supplied: `gendatoc`, `oc_set` and `target_class`. What are the differences between the three?

- `gendatoc`: this function basically constructs from two Matlab arrays a one-class dataset. When you have two datasets `xt` and `xo` available, you can create a one-class dataset using:

  ```
  >> xt = randn(40,2);
  >> xo = gendatb([20,0]);
  >> x = gendatoc(xt,xo);
  ```

  In `gendatoc` `xt` or `xo` do not have to be defined, they can be empty (`xt = []` or `xo = []`). To label a Matlab array as outlier, is therefore every easily done:

  ```
  >> xo = 10*randn(25,2);
  >> x = gendatoc([],xo);
  ```

  If `xt` or `xo` is a Prtools dataset, this data is converted back to normal Matlab arrays. That means that the label information in these datasets

is *lost*. All data in `xt` will be labeled target and all data in `xo` outlier, without exception.

- `oc_set`: this function relabels an existing Prtools dataset such that one of the classes becomes target class, and all others become outlier. You have to supply the label of the class that you want to be target class. Assume you generate data from a banana-shaped distribution, and you want to have the class labeled `1` to be target class:

```
>> x = gendatb([20,20]);       % 40 objects in 2D
>> x = oc_set(x,'1')
```

  Now you still have 40 objects, half is labeled `target`, the other half `outlier`.

  This function `oc_set` also accepts several classes to be labeled as target class. When a 10-class problem is loaded, a subset of these classes can be assigned to be target class:

```
>> load nist16;   % this is an example of a 10-class dataset,
>>                % it might not be available everywhere
>> a
2000 by 256 dataset with 10 classes: [200  200  200
200  200  200  200  200  200  200]
>> x = oc_set(a,[1 5 6])  %select three classes
Class 0 is used as target class.
Class 4 is used as target class.
Class 5 is used as target class.
 (3 classes as target), 2000 by 256 dataset with 2
 classes: [600  1400]
```

  When you don't supply labels, it is assumed that all data is target data:

```
>> x = rand(20,2);
>> x = oc_set(x)
```

  This constructs a dataset, containing 20 target objects in 2D. All objects are now labeled target. When you want to label this data as outlier, you have to supply it as the second argument: `x = oc_set([],x)`.

- `target_class`: this functions labels one of the classes as target (identical to `oc_set`) but furthermore *removes* all other objects from the dataset.

```
>> x = gendatb([20,20]);       % 40 objects in 2D
>> x = target_class(x,'1')     % 20 objects in 2D
```

Now dataset `x` just contains 20 target objects. You can achieve the same in this way:

```
>> x = gendatb([20,20]);       % 40 objects in 2D
>> x = oc_set(x,'1');
>> x = target_class(x)         % 20 objects in 2D
```

but this is not so efficient.

In some cases you may need to extract the outlier data. This is obtained as the second output argument from `target_class`:

```
>> [xt,xo] = target_class(x)   % xo contains 20 outlier objects
```

3.2 Inspecting one-class datasets

You can always check if a dataset is a proper one-class dataset by

```
>> isocset(x)
```

This is often not very useful for normal users, but becomes important when you're constructing one-class classifiers for yourself.
When a one-class dataset is constructed, you can extract again which objects are target or outlier:

```
>> [It,Io] = find_target(x)
>> xt = x(It,:);
>> xo = x(Io,:);
```

Thus `find_target` returns the indices of the target (in `It`) and the outlier data (in `Io`). This is often used to split a one-class dataset into target and outlier objects. This is cheaper than running `target_class` twice:

```
>> xt = target_class(x);
>> xo = target_class(x,'outlier');
```

This last implementation has the added drawback that `xo` is now labeled target. The advantage of using `target_class` is, that in one comment you can extract the target class from the data into a new Prtools dataset. This avoids the lengthy construction `It=find_target(x); xt=x(It,:)`.
The only special thing about one-class datasets is actually, that they contain just 1 or 2 classes, with the labels `target` and/or `outlier`. When you define your own dataset with these two labels, it will be automatically recognized as a one-class dataset.[1] Other datasets with two classes are *not* one-class datasets, because there it is not clear which of the two is considered the target class.

[1]The toolbox also offers the function `relabel` for relabeling a dataset. It redefines the field `lablist` in the dataset. The user first has to find out in which order the classes are labeled in the dataset before he/she can relabel them. Therefore I will not recommend it, although it becomes very useful when you want to label *several* classes as target and the rest as outlier.

Chapter 4

Classifiers

4.1 Prtools classifiers

In Prtools the classifiers are also stored in an object, often called `w`. You can do three things with a classifier. First, you can define an empty mapping, by giving an empty matrix during training:

```
  >> w = parzenc([],0.6)
Parzen Classifier, untrained  mapping   --> parzenc
```

Secondly, you can train a mapping by supplying a dataset `a` during the construction:

```
  >> w = parzenc(a,0.6)
```

or by applying an untrained mapping to the dataset:

```
  >> w = parzenc([],0.6)
  >> w = a*w
```

Finally, you can apply a mapping to a new dataset. The result is another dataset in which the output of the classifier is stored. In principle the output is a posterior probability for each of the classes. This can be inspected by the `+`-operator:

```
  >> a = gendatb;
  >> w = parzenc(a,0.6)
  >> out = a*w
  >> +out
```

In some cases the classifier outputs a distance instead of a density estimate.

4.2 Creating one-class classifiers

The one-class classifiers should be trained on the datasets from the previous chapter. Many one-class classifiers do not know how to use example outliers in their training data. They may therefore complain, or just ignore the outlier objects completely if you supply them in your training data. For now, I call it the responsibility of the user...
All one-class classifiers share the same characteristics:

1. Their names end in `dd`,
2. Their second argument is always the error they may make on the target class (the fraction false negative),
3. Their third argument should characterize the complexity of the classifier. That means that for one extreme of the parameter values the error on the targets is low, but it therefore has a high error on the outlier class (the model is undertrained). For the other extreme of the parameter values, the error on the target class is high, but the error on the outliers is low (the model is overtrained). This complexity parameter can then be optimized using `consistent_occ`.
4. The mapping should output the labels `target` and `outlier`.
5. The mapping should contain a parameter `threshold` which defines the separation between the target and outlier class. In practice, this is only interesting for programmers who want to implement a classifier themselves. Please look at section 4.6.

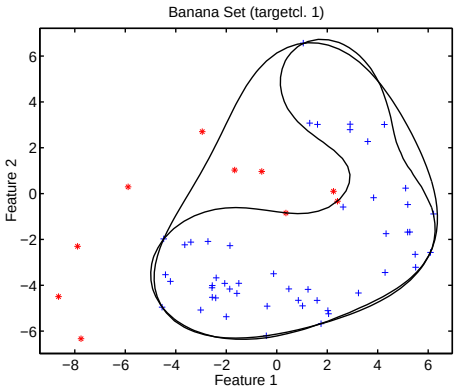
An example of a one-class classifier is for instance:

```
>> x = target_class(gendatb([20 0]),'1');
>> w = gauss_dd(x,0.1)
```

This trains a classifier `gauss_dd` on data `x` (this particular classifier just estimates a Gaussian density on the target class). A threshold is put such that 10% of the training target objects will be rejected and classified as outlier. So the fraction false negative will be 0.1. (Note that this is optimized on the training data. This means that the performance on an independent test set might deviate significantly!) After this rejection threshold, other

parameters can be given (for instance, for the k-means clustering method, it is the number of clusters `k`).
These one-class classifiers are normal `mappings` in the `Prtools` sense. So they can be plotted by `plotc`, can be combined with other mappings by `[]`, `*`, etc. To check if a classifier is a one-class classifier (i.e. it labels objects as `target` or `outlier`), use `isocc`.

```
>> x=oc_set(gendatb([50,10]),'1')
>> scatterd(x,'legend')
>> w = svdd(target_class(x),0.1,8);
>> plotc(w)
>> w = svdd(x,0.1,8);
>> plotc(w)
```



4.3 Inspecting one-class classifiers

In research, one often wants to see what the values of the optimized parameters are. For each type of classifier a data structure is saved. This can be retrieved by:

```
>> W = +w;          % possibility 1
>> W = w.data;      % possibility 2
```

This `W` now contains several sub-fields with the optimized parameters. Which parameters are stored, depends on the classifier. The format is free, except that one parameter, `threshold`, should always be there.
An example is the support vector data description. In a quadratic optimization procedure, the weights α are optimized. Assume, I want to have 10% of the data on the boundary, using a Gaussian kernel with a kernel parameter $\sigma = 5$ (forget the details, they are not important). Now I'm interested in what the optimal α's will be:

```
>> x = target_class(gendatb([50,0]),'1');
>> w = svdd(x,0.1,5);
```

```
>> W = +w;
>> W.a
```

Note The SVDD has been changed in this new version of the toolbox. Have a look at the remarks at the end of the file (chapter 6).
Another example is the Mixture of Gaussians. Let us see if we can plot the boundary and the centers of the clusters. First, we create some data and train the classifier (using 5 clusters):

```
>> x = target_class(gendatb([100,0]),'1');
>> w = mog_dd(x,0.1,5);
```

Now we inspect the trained classifier and give some visual feedback:

```
>> W = +w
>> scatterd(x);
>> plotc(w); hold on;
>> scatterd(W.m,'r*')
```

Apparently, the means of the clusters are stored in the `m` field of the structure in the classifier.

4.4 Available classifiers

Currently, a whole set of one-class classifiers is already implemented, ready to use. Please feel free to extend this list!

- `gauss_dd`: simple Gaussian target distribution, without any robustifying. This is the first classifier I would try. The target class is modeled as a Gaussian distribution. To avoid numerical instabilities the density estimate is avoided, and just the Mahalanobis distance is used:

$$f(\mathbf{x}) = (\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu) \tag{4.1}$$

  The classifier is defined as:

$$h(\mathbf{x}) = \begin{cases} \text{target} & \text{if} \quad f(\mathbf{x}) \leq \theta \\ \text{outlier} & \text{if} \quad f(\mathbf{x}) > \theta \end{cases} \tag{4.2}$$

  The mean μ and covariance matrix Σ are just sample estimates. The threshold θ is set according to the target error that the user has to supply.

- `rob_gauss_dd`: Gaussian target distribution, but robustified. Can be significantly better for data with long tails. The mathematical description of the method is identical to `gauss_dd`. The difference is in the computation of the μ and Σ. This procedure reweights the objects in the training set according to their proximity to the (previously estimated) mean. Remote objects (candidate outliers) will be down weighted such that a more robust estimate is obtained. It will not be strictly robust, because these outliers will always have some influence.

- `mcd_gauss_dd`: Minimum Covariance Determinant Gaussian classifier. The mathematical description is again the same as of the `gauss_dd`. For the estimation of the mean and covariance matrix just a fraction of the data is used. This part of the data is selected such that the determinant of the covariance matrix is minimal. The implementation is taken from [RVD99], but unfortunately it only works up to 50 dimensional data.

- `mog_dd`: Mixture of Gaussians. Here the target class is modeled using a mixture of K Gaussians, to create a more flexible description. The model looks like:

$$f(\mathbf{x}) = \sum_{i=1}^{K} P_i \exp\left(-(\mathbf{x}-\mu_i)^T \Sigma_i^{-1} (\mathbf{x}-\mu_i)\right) \tag{4.3}$$

  The classifier is defined as:

$$h(\mathbf{x}) = \begin{cases} \text{target} & \text{if} \quad f(\mathbf{x}) \geq \theta \\ \text{outlier} & \text{if} \quad f(\mathbf{x}) < \theta \end{cases} \tag{4.4}$$

  The parameters P_i, μ_i and Σ_i are optimized using the EM algorithm. Because in high dimensional data and larger number of clusters K, the number of free parameters can become huge (in particular in the covariance matrices), the covariance matrices can be constrained. This can improve the performance significantly.

  In this version of `mog_dd` it is also possible to use outlier objects in training. Individual mixtures of Gaussians are fitted for both the target and outlier data (having Kt and Ko Gaussians respectively). Objects are assigned to the class with the highest density. To avoid that the decision boundary around the target class will not be closed, one extra outlier cluster is introduced with a very wide covariance matrix.

This 'background' outlier cluster is fixed and will not be adapted in the EM algorithm (although it will be used in the computation of the probability density). This results in the following model:

$$\begin{aligned} f(\mathbf{x}) &= \sum_{i=1}^{Kt} P_i \exp\left(-(\mathbf{x}-\mu_i)^T \Sigma_i^{-1} (\mathbf{x}-\mu_i)\right) \\ &\quad -P_* \exp\left(-(\mathbf{x}-\mu)^T \Sigma_*^{-1} (\mathbf{x}-\mu)\right) \\ &\quad -\sum_{j=1}^{Ko} P_j \exp\left(-(\mathbf{x}-\mu_j)^T \Sigma_i^{-1} (\mathbf{x}-\mu_j)\right) \end{aligned} \tag{4.5}$$

Classifying is done according to (4.4). Here μ is the mean of the complete dataset, and Σ_* is taken as 10Σ, where Σ is covariance matrix of the complete dataset. The P_* is still optimized in the EM procedure, such that $P_* + \sum_j P_j = 1$.

Finally, it is also possible to *extend* a trained mixture by a new cluster (can be both a target or an outlier cluster). Have a look at the function `mog_extend`.

- `parzen_dd`: Parzen density estimator. An even more flexible density model with a Gaussian model around each of the training objects:

$$f(\mathbf{x}) = \sum_{i=1}^{N} \exp\left(-(\mathbf{x}-\mathbf{x}_i)^T h^{-2} (\mathbf{x}-\mathbf{x}_i)\right) \tag{4.6}$$

  The free parameter h is optimized by maximizing the likelihood on the training data using leave-one-out. The classifier becomes as in (4.4). This method often performs pretty well, but it requires a reasonable training set. My second choice!

- `autoenc_dd`: the auto-encoder neural network. A full explanation of neural networks is outside the scope of this manual, please have a look at [Bis95]. The idea is that a neural network is trained to reconstruct the input pattern $\mathbf{x}$ at the output $\mathrm{NeurN}(\mathbf{x})$ of the network. The difference between the input and output pattern is used as a characterization of the target class. This results in:

$$f(\mathbf{x}) = \left(\mathbf{x} - \mathrm{NeurN}(\mathbf{x})\right)^2 \tag{4.7}$$

  The classifier then becomes as in (4.2).

- `kmeans_dd`: the k-means data description, where the data is described by k clusters, placed such that the average distance to a cluster center is minimized. The cluster centers $\mathbf{c}_i$ are placed using the standard k-means clustering procedure ([Bis95]). The target class is then characterized by:

$$f(\mathbf{x}) = \min_i (\mathbf{x} - \mathbf{c}_i))^2 \tag{4.8}$$

  The classifier then becomes as in (4.2).

- `kcenter_dd`: the k-center data description, where the data is described by k clusters, placed such that the maximal distance to a cluster center is minimized [YD98]. When the clusters are placed, the mathematical description of the method is similar to `kmeans_dd`.

- `pca_dd`: Principal Component Analysis data description. This method describes the target data by a linear subspace. This subspace is defined by the eigenvectors of the data covariance matrix Σ. Only k eigenvectors are used. Assume they are stored in a $d \times k$ matrix $\mathbf{W}$ (where d is the dimensionality of the original feature space).

  To check if a new object fits the target subspace, the reconstruction error is computed. The reconstruction error is the difference between the original object $\mathbf{x}$and the projection of that object onto the subspace (in the original data). This projection is computed by:

$$\mathbf{x}_{proj} = \mathbf{W}(\mathbf{W}^T\mathbf{W})^{-1}\mathbf{W}^T\mathbf{x} \tag{4.9}$$

  For the reconstruction error I decided to use:

$$f(\mathbf{x}) = ||\mathbf{x} - \mathbf{x}_{proj}||^2 \tag{4.10}$$

  The default setting of the method is that the k eigenvectors with the largest eigenvalues are used. It appears that this does not always have the best performance [TM03]. It is also possible to choose the k eigenvectors with the *smallest* eigenvalues. This is an extra feature in the toolbox.

- `som_dd`: Self-Organizing Map data description. This method uses a Self-Organizing Map to describe the data. A SOM is an unsupervised

clustering method in which the cluster centers are constrained in their placing. The construction of the SOM is such that all objects in the feature space retain as much as possible their distance and neighborhood relations in the mapped space.

The mapping is performed by a specific type of neural network, equipped with a special learning rule. Assume that we want to map an d-dimensional measurement space to a k-dimensional feature space, where $k < d$. In the feature space, we define a finite orthogonal grid with $K \times K$ grid points $\mathbf{x}_{SOM}$. At each grid point we place a neuron. Each neuron stores an d-dimensional vector that serves as a cluster center. By defining a grid for the neurons, each neuron does not only have a neighboring neuron in the measurement space, it also has a neighboring neuron in the grid. During the learning phase, neighboring neurons in the grid are enforced to also be neighbors in the measurement space. By doing so, the local topology will be preserved. In this implementation the SOM only $k = 1$ or $k = 2$.

To evaluate if a new object fits this model, again a reconstruction error is defined. This reconstruction error is the difference between the object and its closest cluster center (neuron) in the SOM:

$$f(\mathbf{x}) = \min_i ||\mathbf{x} - \mathbf{x}_{SOM}||^2 \tag{4.11}$$

This distance is thresholded in order to get a classification result.

- `mst_dd`: The minimum spanning tree data description. On the training data a minimum spanning tree is fitted. The distance to the *edges* is used as the similarity to the target class. That means that a training target dataset of just two objects will define sausage-shaped target class.

- `nndd`: A simple nearest neighbor method [Tax01]. Here a new object is evaluated by computing the distance to its nearest neighbor $NN(\mathbf{x})$. This distance is normalized by the nearest neighbor distance of this object (that means the distance between object $NN(\mathbf{x})$ and $NN(NN(\mathbf{x}))$. Works not very well for low dimensional, well-sampled data, but surprisingly well for low-sample, high-dimensional data!

- `knndd`: A k-nearest neighbor data description, a much smarter approach to the `nndd` method. In its most simple version just the distance

to the k-th nearest neighbor is used. Slightly advanced methods use averaged distances, which works somewhat better. This simple method is often very good in high dimensional feature spaces.

- `svdd`: The support vector data description, the description inspired by the support vector classifier. For a full discussion of this method, please read [Tax01]. It basically fits a hypersphere around the target class. By introducing kernels, this inflexible model becomes much more powerful, and can give excellent results when a suitable kernel is used. It is possible to optimize the method to reject a pre-defined fraction of the target data. That means that for different rejection rates, the shape of the boundary changes. Furthermore it is possible to use example outliers to improve the classification results. The main drawback of the method is that it requires a difficult optimization. For this the `quadprog` algorithm from the optimization toolbox is required.

  This first implementation has the RBF kernel hard-coded. This is because it is the most useful kernel, and it makes the implementation and evaluation must simpler. When you want to use other kernels, you have to have a look at one of the next two classifiers:

- `ksvdd`: The support vector data description using a general kernel. The choice of kernel is free in this implementation, but this makes the evaluation in general much slower. The implementation is not the most convenient, so maybe I will change this again in the future.

- `incsvdd`: The incremental support vector machine which uses its own optimization routine. This makes it possible to optimize the SVDD without the use of an external quadratic programming optimizer, and to use any kernel. In future versions this will be adapted to cope with dynamically changing data (data distributions which change in time). Currently this is my preferred way to train a SVDD.

- `dlpdd`: The linear programming distance-data description [PTD03]. This data descriptor is specifically constructed to describe target objects which are represented in terms of distances to other objects. In some cases it might be much easier to define distances between objects than informative features (for instance when shapes have to be distinguished). To stress that the classifier is operating on distance data, the

name starts with a `d`. The classifier has basically the following form:

$$f(\mathbf{x}) \sum_i w_i d(\mathbf{x}, \mathbf{x}_i) \tag{4.12}$$

The weights w are optimized such that just a few weights stay non-zero, and the boundary is as tight as possible around the data.

- `lpdd`: The linear programming data description. The fact that `dlpdd` is using distance data instead of standard feature data makes it harder to simply use it on normal feature data. Therefore `lpdd` is created, which is basically a wrapper combining some high level algorithms together to make the application of `dlpdd` simpler.

- `mpm_dd`: The minimax probability machine by Lanckriet [LEGJ03]. It tries to find the linear classifier that separates the data from the origin, rejecting maximally a specific fraction of the target data. In the original version, an upper bound on this rejection error is used (applying a very general bound using only the mean and covariance matrix of the target data). Unfortunately, in practice this bound is so loose that it is not useful. Therefore the rejection threshold is re-derived from the target data.

To get more information for each individual classifier, have a look at their help (for instance `help gauss_dd`).

4.5 Combining one-class classifiers

Many real-world datasets have a much more complicated distribution than can be modeled by, say, a mixture of Gaussians. It appears that it might be very benificial to combine classifiers. Each of the classifiers can focus on a specific feature or characteristic in the data. By combining the classifiers, one hopes to combine all the strong points of the classifiers, and obtain a much more flexible model.
Like with normal classifiers, there is the problem that the outputs of the classifiers should be rescaled in such a way, that the outputs become comparable. For trainable combining this is not very essential, but when fixed combination rules like mean-rule, max-rule, median-rule are considered, the outputs of the classifiers should be rescaled.

In Prtools, the output of many classifiers are scaled by fitting a sigmoid function and then normalized such that the sum of the outputs becomes 1. In this way, the outputs of the classifier can be interpreted as (an approximation to) the class posterior probabilities. This scaling is done by the function `classc` (see also `prex_combining.m`).
For one-class classifiers there is a small complication. Here there are two types of classifiers: classifiers based on density estimations, and classifiers based on distances to a model. Normalization of the first type of classifiers is no problem, it directly follows the strategy of Prtools. The output of the second type of classifiers, on the other hand, causes problems. Imagine an object belonging to the target class. It will have a distance to the target model smaller than some threshold. According to Prtools, the objects are assigned to the class with the *highest* output. Therefore, in `dd_tools`, the distances of the distance-based classifiers are negated such that the output for the target class with be higher than the threshold.
To normalize the outputs of these distance-based classifiers, the output of these classifiers have to be negated again. This means that the standard `classc` cannot be applied. For this, a new function `dd_normc` is introduced. The standard approach is to multiply all classifiers per default with `dd_normc`. It will not change the classification by the classifier.

```
>> a = target_class(gendatb);
>> w1 = gauss_dd(a,0.1);    % define 4 arbitrary OC classifiers
>> w2 = pca_dd(a,0.1,1);
>> w3 = kmeans_dd(a,0.1,3);
>> w4 = mog_dd(a,0.1,2);
>>                          % combine them with the mean comb rule:
>> W = [w1*dd_normc w2*dd_normc w3*dd_normc w4*dd_normc] * meanc;
>> scatterd(x);
>> plotc(W);
```

4.6 Note for programmers

In Prtools the objects are assigned to the class with the largest output. In `dd_tools` it is a general convention (but not required) that the classifier outputs two values for each object (in a row vector). The output of a dataset with n objects is therefore a $n \times 2$ matrix. The first value (the first column of the

matrix) the 'probability'/'confidence'[1] for the target class is given, the second value (the second column in the matrix) gives the 'probability'/'confidence' for the outlier class.
In many one-class classifiers the target 'probability' value is modeled or estimated, while the outlier 'probability' value is determined afterwards by the user-defined fraction false negative. Therefore the first output value will vary for different input objects, while the second output value is constant. This is used in the computation of the ROC curve. For different thresholds the fraction acceptance and rejection is investigated using the first output column.
If you feel like implementing your own one-class classifier, please have a look at `random_dd.m`. This trivial classifier contains all essential parts for the definition of a one-class classifier, and it is very simple to adapt and extend this classifier.

```
%RANDOM_DD Random one-class classifier
%
%       W = RANDOM_DD(A,FRACREJ)
%
% This is the trivial one-class classifier, randomly assigning labels
% and rejecting FRACREJ of the data objects. This procedure is just to
% show the basic setup of a Prtools classifier, and what is required
% to define a one-class classifier.

% Copyright: D.M.J. Tax, R.P.W. Duin, D.M.J.Tax@prtools.org
% Faculty of Applied Physics, Delft University of Technology
% P.O. Box 5046, 2600 GA Delft, The Netherlands

function W = random_dd(a,fracrej)

% Take care of empty/not-defined arguments:
if nargin < 2 fracrej = 0.05; end
if nargin < 1 | isempty(a)
% When no inputs are given, we are expected to return an empty
% mapping:
```

[1] This 'probability' or 'confidence' is between brackets, because in many cases no strict probability is estimated, just a type of similarity to the target class.

```
W = mapping(mfilename,{fracrej});
W = setname(W,'Random one-class classifier');
return
end

if ~ismapping(fracrej)           %training

a = target_class(a);     % only use the target class
[m,k] = size(a);

% train it:
% this trivial classifier cannot be trained. for each object we will
% output a random value between 0 and 1, indicating the probability
% that an object belongs to class 'target'
% if we would like to train something, we should do it here.

%and save all useful data:
W.threshold = fracrej;  % a threshold should always be defined
W = mapping(mfilename,'trained',W,str2mat('target','outlier'),k,2);
W = setname(W,'Random one-class classifier');

else                                %testing

  W = getdata(fracrej); %unpack
  [m,k] = size(a);
  % This classifier only contains the threshold, nothing more.

  % Output should consist of two numbers: the first indicating the
  % probability that it belongs to the target, the second indicating
  % the probability that it belongs to the outlier class. The latter
  % is often the constant threshold:
  newout = [rand(m,1) repmat(W.threshold,m,1)];

  W = setdat(a,newout,fracrej);
end
return
```

Chapter 5

Error computation

5.1 Basic errors

In order to evaluate one-class classifiers, one has to find out what the error of the first and second kind are (or the false positive and false negative rate). When you are only given target objects, life becomes hard and you cannot estimate these errors. But first assume you have some target and outlier test objects available. The false positive and false negative rates can be computed by `dd_error`:

```
>> x = target_class(gendatb([50 0]),'1');
>> w = gauss_dd(x,0.1);
>> z = oc_set(gendatb(200),'1');
>> e = dd_error(z,w)
>> dd_error(z*w)   % other possibility
>> z*w*dd_error    % other possibility
```

The first entry `e(1)` gives the false negative rate (i.e. the error on the target class) while `e(2)` gives the false positive rate (the error on the outlier class). Question: can you imagine what would happen when you would replace `oc_set` in the third line by `target_class`?

5.2 Precision and recall

In the literature, two other measures are often used, namely

precision : defined as

$$\text{precision} = \frac{\#\text{ of correct target predictions}}{\#\text{ of target predictions}},$$

recall : is basically the true positive rate

$$\text{recall} = \frac{\#\text{ of correct target predictions}}{\#\text{ of target examples}}.$$

These errors are returned in the second output variable of `dd_error`:

```
>> [e,f] = dd_error(z,w)
```

Here `f(1)` contains the precision, and `f(2)` the recall.
Finally, a derived performance criterion using the precision and recall is the $F1$ measure, defined as:

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

This can be computed using `dd_f1`:

```
>> x = target_class(gendatb([50 0]),'1');
>> w = svdd(x,0.1);
>> z = oc_set(gendatb(200),'1');
>> dd_f1(x,w)
>> dd_f1(x*w)
>> x*w*dd_f1
```

5.3 Area under the ROC curve

In most cases we are not interested in just one single threshold (in the previous example we took an error of 10% on the target class), we want to estimate the whole ROC-curve. This can be estimated by `dd_roc`:

```
>> x = target_class(gendatb([50 0]),'1');
>> w = svdd(x,0.1,7);
>> z = oc_set(gendatb(200),'1');
>> e = dd_roc(z,w)
>> e = dd_roc(z*w)   % other possibility
>> e = z*w*dd_roc    % other possibility
```

First the classifier is trained on `x` for a specific threshold. Then for varying thresholds, the classifier is evaluated on dataset `z`. The results are returned in a ROC curve, given in a matrix `e` with two columns, the first indicating the false negatives, the second the false positives.

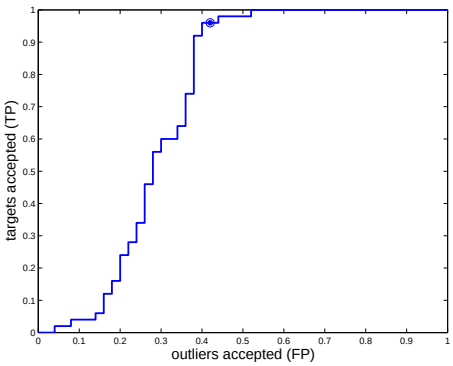


Figure 5.1: Receiver-Operating characteristic curve, wit the operating point indicated by the dot.

The ROC-curve can be plotted by:

```
>> plotroc(e);
```

An example of such a ROC curve is shown in figure 5.1.
In the newest version of the toolbox, the ROC is extended to show also the operating point of the classifier. When this feature is required, you have to supply the mapping and the dataset separately:

```
>> a = oc_set(gendatb,1);
>> w = gauss_dd(a,0.1);
>> h = plotroc(w,a)
```

By moving the mouse, and clicking, the user can change the position of the operating point. Inside the figure, a new mapping with this new operating point is stored. This mapping can be retrieved in the Matlab working space by:

```
>> w2 = getrocw(h)
```

To get a feeling for this, please try the demo `dd_ex8`.

Because it is very hard to compare ROC curves of different classifiers, often the AUC error (Area Under the AUC curve is taken). In my definition of the AUC error, the larger the value, the better the one-class classifier. It is computed from the ROC curve values using the function `dd_auc`:

```
>> x = target_class(gendatb([50 0]),'1');
>> w = svdd;
>> z = oc_set(gendatb(200),'1');
>> e = dd_roc(w,x,z);
>> err = dd_auc(e);
```

In many cases only a restricted range for the false negatives is of interest: for instance, we want to reject less than half of the target objects. In these cases one may want to set bounds on the range of the AUC error:

```
>> e = dd_roc(w,x,z);
>> err = dd_auc(e,[0.05 0.5]);
```

5.4 Cost curve

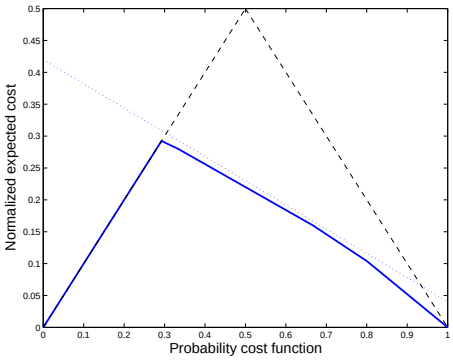


Figure 5.2: The cost curve derived from the same dataset and classifier as in Figure 5.1.

Another proposal for plotting the performance of a classifier is by using the cost-curve [DH00]. As you can see in figure 5.1, there are many thresholds that are suboptimal: there is often another operating point for which at least one of the errors is lower. For instance, the operating point $FP = 0.38, TP =$

0.72 is suboptimal, because operating point $FP = 0.38, TP = 0.92$ has much better true positive rate with equal false positive rate. For a relative large range of misclassification costs the operating point $FP = 0.38, TP = 0.92$ will be the optimal one.
This is indicated in a cost curve. For a varying cost-ratio between the two classes, the (normalized) expected cost is computed. Each operating point appears as a line in this plot. In Figure 5.2 the cost curve for the same dataset and classifier as of the ROC curve in Figure 5.1 is shown. The operating point of Figure 5.1 is indicated by the dotted line in Figure 5.2. The combination of operating points that form the lower hull is indicated by the thick line, and shows the best operating points over the range of costs. This cost curve is obtained like:

```
>> a = oc_set(gendatb,1);
>> w = gauss_dd(a,0.1);
>> c = a*w*dd_costc;
>> plotcostc(c)
```

For more information, please look at [DH00].

5.5 Generating artificial outliers

When you are not so fortunate to have example outliers available for testing, you can create them yourself. Say that `z` is a set of test target objects. Artificial outliers can be generated by:

```
>> z_o = make_outliers(z,100)
```

This creates a new dataset from `z`, containing both the target objects from `z` and 100 new artificial outliers. These are generated from a uniform spherical distribution around `z`.
This works well in practice for low dimensional dataset. For higher dimensions, it becomes very inefficient. Most of the data will be in the 'corners' of the box. In these cases it is better to generate data uniform in a sphere.

```
>> z_o = gendatout(z,100)
```

In this version, the most tight hypersphere around the data is fitted. Given the center and radius of this sphere, data can be uniformly generated by `randsph` (this is not trivial!).

5.6 Cross-validation

The toolbox has an extra procedure to facilitate cross-validation. In cross-validation a dataset is split into B batches. From these batches $B-1$ are used to train a classifier, and the left-out batch is used to evaluate it. This is repeated B times, and the performances are averaged. The advantage is that given a limited training set, it is still possible to obtain a relatively good classifier, and estimate its performance on an independent set.
In practice, this cross-validation procedure is applied over and over again. Not only to evaluate and compare the performance of classifiers, but also to optimize hyperparameters. To keep the procedure as flexible as possible, the cross-validation is kept as simple as possible. An index vector is generated that indicates to with batch each object in a training set belongs. By repeatedly applying the procedure, the different batches are combined in a training and evaluation set. The following piece of code shows how this is done in practice:

```
a = oc_set(gendatb,1);  % make or get some data
nrbags = 10;  % we are doing 10-fold crossvalidation
I = nrbags;   % initialization
              % now start the 10 folds:
for i=1:nrbags
    % extract the training (x) and validation (z) sets, and
 % update the index vector I:
    [x,z,I] = dd_crossval(a,I);

    % do something useful with the training and evaluation:
 w = gauss_dd(x,0.1);
 e(i) = dd_auc(z*w*dd_roc);
end
fprintf('AUC (10-fold) %5.3 (%5.3)',mean(e),std(e));
```

Note that the procedure takes class priors into account. It tries to retain the number of objects per class in each fold according to the total dataset.

Chapter 6

General remarks

In this chapter I collected some remarks which are important to see once, but did not fit in the line of the previous chapters.

1. If you want to know more about a classifier or function, always try the `help` command.

2. Also have a look at the file `Contents.m`. This contains the full list of functions and classifiers defined in the toolbox.

3. In older versions of the toolbox, the width parameter σ in the support vector data description was optimized automatically. This was done such that a prespecified fraction `fracrej` of the objects was *on* the boundary (so these are support vectors with $0 < \alpha_i < 1/(N\nu)$). Another parameter C was set such that another prespecified fraction `fracerr` of objects was *outside* the boundary (the support vectors with $\alpha_i = 1/(N\nu)$). The default of this fraction was `fracerr` $= 0.01$ and was often ignored in practical experiments. But this lead sometimes to poor results, and created a lot of confusion. If you really want to, and if you're lucky that I included it, it is still available under `newsvdd.m`.

   I decided to consider the parameter σ as a hyper-parameter. This parameter will not be optimized automatically, but has to be set by the user. To obtain the prespecified error `fracrej` on the target set, the parameter C will be set. The parameter `fracerr` is removed.

   Another complaint about the first implementation of the svdd was, that it was completely aimed at the RBF kernel. That was because the

optimization simplifies significantly with this assumption. Using `ksvdd` or `incsvdd` this restriction is now lifted. In particular `incsvdd` is recommended because it does not rely on external quadratic programming optimizers which always creates problems.

4. There is also a set of functions for visualizing the output of a classifier in 2D. One can define a grid of objects around a 2D dataset, and put that into a dataset. That dataset can be classified by the classifier, and mapped back into the feature space. The user can thus inspect the output of the classifier for the whole feature space around the target class.

   This is explicitly done in the following code:

   ```
   >> x = target_class(gendatb([50 0]),'1');
   >> w = svdd(x,0.1,5);
   >> scatterd(x);
   >> griddat = gendatgrid;
   >> out = w*griddat;
   >> plotg(out);
   >> hold on;
   >> scatterd(x);
   ```

5. There is also one function which is in essence not a one-class classifier, but a preprocessor: the kernel whitening `kwhiten`. This mapping does not classify data, only transforms it into a new dataset. It is hoped that it is transformed into a shaped which can be described better by one-class classifiers. The easiest way to work with this type of preprocessing, is to exploit some `Prtools` techniques:

   ```
   >> x = target_class(gendatb([50 0]),'1');
   >> w_kpca = kwhiten(x,0.99,'p',2);
   >> w = gauss_dd(w_kpca*x,0.1);
   >> W = w_kpca*w;
   ```

   This `W` can now be used as a normal classifier.

6. I'm not responsible for the correct functioning of the toolbox, but of course I do my best to make the toolbox as useful and bug-free as possible. Please email me when you have found a bug at `D.M.J.Tax@prtools.org`.

I'm also very interested when people have defined new one-class classifiers.

Chapter 7

Contents.m of the toolbox

```
% Data Description Toolbox
% Version 1.7.1 31-Jul-2008
%
%Dataset construction
%--------------------
%isocset        true if dataset is one-class dataset
%gendatoc       generate a one-class dataset from two data matrices
%oc_set         change normal classif. problem to one-class problem
%target_class   extracts the target class from an one-class dataset
%gendatgrid     create a grid dataset around a 2D dataset
%gendatout      create outlier data in a hypersphere around the
%               target data
%gendatblockout create outlier data in a box around the target class
%gendatoutg     create outlier data normally distributed around the
%               target data
%gendatouts     create outlier data in the data PCA subspace in a
%               hypersphere around the target data
%dd_crossval    cross-validation dataset creation
%dd_label       put the classification labels in the same dataset
%
%Data preprocessing
%------------------
%myproxm        replacement for proxm.m
%kwhiten        rescale data to unit variance in kernel space
%gower          compute the Gower similarities
```

```
%
%One-class classifiers
%---------------------
%random_dd      description which randomly assigns labels
%stump_dd       threshold the first feature
%gauss_dd       data description using normal density
%rob_gauss_dd   robustified gaussian distribution
%mcd_gauss_dd   Minimum Covariance Determinant gaussian
%mog_dd         mixture of Gaussians data description
%mog_extend     extend a Mixture of Gaussians data description
%parzen_dd      Parzen density data description
%nparzen_dd     Naive Parzen density data description
%
%autoenc_dd     auto-encoder neural network data description
%kcenter_dd     k-center data description
%kmeans_dd      k-means data description
%pca_dd         principal component data description
%som_dd         Self-Organizing Map data description
%mst_dd         minimum spanning tree data description
%
%nndd           nearest neighbor based data description
%knndd          K-nearest neighbor data description
%ball_dd        L_p-ball data description
%lpball_dd      extended L_p-ball data description
%svdd           Support vector data description
%incsvdd        Incremental Support vector data description
%ksvdd          SVDD on general kernel matrices
%lpdd           linear programming data description
%mpm_dd         minimax probability machine data description
%
%dkcenter_dd    distance k-center data description
%dnndd          distance nearest neighbor based data description
%dknndd         distance K-nearest neighbor data description
%dlpdd          distance-linear programming data description
%
%isocc          true if classifier is one-class classifier
%
%AUC optimizers
```

```
%--------------
%rankboostc     Rank-boosting algorithm
%auclpm         AUC linear programming mapping
%
%Classifier postprocessing/optimization/combining.
%-------------------------------------
%consistent_occ optimize the hyperparameter using consistency
%optim_auc      optimize the hyperparameter by maximizing AUC
%dd_normc       normalize oc-classifier output
%multic         construct a multi-class classifier from OCC's
%
%Error computation.
%-----------------
%dd_error       false positive and negative fraction of classifier
%dd_f1          F1 score computation
%dd_eer         equal error rate
%dd_roc         computation of the Receiver-Operating Characterisic curve
%dd_prc         computation of the Precision-Recall curve
%dd_auc         error under the ROC curve
%dd_meanprec    mean precision of the Precision-Recall curve
%dd_costc       cost curve
%dd_delta_aic   AIC error for density estimators
%dd_fp          compute false positives for given false negative
%               fraction
%simpleroc      basic ROC curve computation
%dd_setfn       set the threshold for a false negative rate
%
%Plot functions.
%--------------
%plotroc        plot an ROC curve or precision-recall curve
%plotcostc      plot the cost curve
%plotg          plot a 2D grid of function values
%plotw          plot a 2D real-valued output of classifier w
%askerplot      plot the FP and FN fraction wrt the thresholds
%plot_mst       plot the minimum spanning tree
%
%Support functions.
%-----------------
```

```
%istarget       true if an object is target
%find_target    gives the indices of target and outlier objs from a dataset
%getoclab       returns numeric labels (+1/-1)
%dist2dens      map distance to posterior probabilities
%dd_threshold   give percentiles for a sample
%randsph        create outlier data uniformly in a unit hypersphere
%makegriddat    auxiliary function for constructing grid data
%relabel        relabel a dataset
%dd_kernel      general kernel definitions
%center         center the kernel matrix in kernel space
%gausspdf       multi-variate Gaussian prob.dens.function
%mahaldist      Mahalanobis distance
%sqeucldistm    square Euclidean distance
%mog_init       initialize a Mixture of Gaussians
%mog_P          probability density of Mixture of Gaussians
%mog_update     update a MoG using EM
%mogEMupdate    EM procedure to optimize Mixture of Gaussians
%mogEMextend    smartly extend a MoG and apply EM
%mykmeans       own implementation of the k-means clustering algorithm
%getfeattype    find the nominal and continuous features
%knn_optk       optimization of k for the knndd using leave-one-out
%volsphere      compute the volume of a hypersphere
%scale_range    compute a reasonable range of scales for a dataset
%nndist_range   compute the average nearest neighbor distance
%inckernel      kernel definitions for the incsvdd
%Wstartup       startup function incsvdd
%Wadd/Wremove   add/remove one object to an incsvdd
%Wstore         store the structure in an incsvdd
%plotroc_update support function for plotroc
%roc_hull       convex hull over a ROC curve
%lpball_dist    lp-distance to a center
%lpball_vol     volume of a lpball
%lpdist         fast lp-distance between two datasets
%dd_message     printf with colors
%
%Examples
%--------
%dd_ex1         show performance of nndd and svdd
```

```
%dd_ex2         show the performances of a list of classifiers
%dd_ex3         shows the use of the svdd and ksvdd
%dd_ex4         optimizes a hyperparameter using consistent_occ
%dd_ex5         shows the construction of lpdd from dlpdd
%dd_ex6         shows the different Mixture of Gaussians classifiers
%dd_ex7         shows the combination of one-class classifiers
%dd_ex8         shows the interactive adjustment of the operating point
%dd_ex9         shows the use of dd_crossval
%dd_ex10        shows the use of the incremental SVDD
%dd_ex11        the construction of a multi-class classifier using OCCs
%
% Copyright: D.M.J. Tax, D.M.J.Tax@prtools.org
% Faculty EWI, Delft University of Technology
% P.O. Box 5031, 2600 GA Delft, The Netherlands
```

Index

Bibliography

[Bis95] C.M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Walton Street, Oxford OX2 6DP, 1995.

[DH00] C. Drummond and R.C. Holte. Explicitly representing expected cost: an alternative to ROC representation. In *Knowledge Discovery and Data Mining*, pages 198–207, 2000.

[LEGJ03] G.R.G. Lanckriet, L. El Ghaoui, and M.I. Jordan. Robust novelty detection with single-class mpm. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press: Cambridge, MA, 2003. E,.

[PTD03] E. Pekalska, D.M.J. Tax, and R.P.W. Duin. One-class LP classifier for dissimilarity representations. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press: Cambridge, MA, 2003.

[RVD99] P.J. Rousseeuw and K. Van Driessen. A fast algorithm for the minimum covariance determinant estimator. *Technometrics*, 41:212–223, 1999.

[Tax01] D.M.J. Tax. *One-class classification*. PhD thesis, Delft University of Technology, http://ict.ewi.tudelft.nl/~davidt/thesis.pdf, June 2001.

[TM03] D.M.J. Tax and K.R. Müller. Feature extraction for one-class classification. In *Proceedings of the ICANN/ICONIP 2003*, pages 342–349, 2003.

[YD98] A. Ypma and R.P.W. Duin. Support objects for domain approximation. In *ICANN'98*, Skovde (Sweden), September 1998.