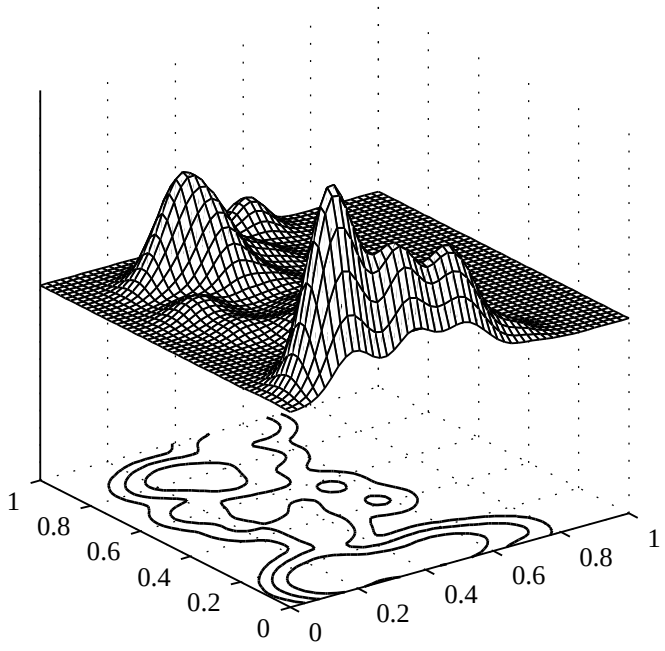
2021 BioSB course

on

Machine Learning for Bioinformatics & Systems Biology

Laboratory course manual



Marcel J.T. Reinders, Perry D. Moerland, Lodewyk F.A. Wessels,
Robert P.W. Duin, Marco Loog, David M.J. Tax, and Dick de Ridder

Contents

Day 1

Machine learning and density estimation

This first day, you will get acquainted with PRTools, a pattern recognition and machine learning toolbox for Matlab. If you are not familiar with Matlab yet, please read Appendix A first. Next, you will perform some exercises on density estimation and construct a first, simple classifier.

1.1 PRTools

PRTools is built around the concept of a *dataset*, a set of objects represented by vectors in a feature space. The central data structure is hence the `dataset` object. It consists of a matrix of size `n` $\times$ `p`; `n` row vectors corresponding to the objects, represented by `p` features each. Attached to this matrix is a set of `n` labels (strings or numbers), one for each object, and a set of `p` feature names (also strings or numbers), one for each feature. Moreover, a set of prior probabilities, one for each class, is stored. Objects with the same label belong to the same class.

In most help files in PRTools, a dataset is denoted by `A`. Almost all routines can handle multi-class objects. Some useful routines to handle datasets are:

`dataset` routines	
`dataset`	Define a dataset from data matrix and labels
`gendat`	Generate a random subset of a dataset
`genlab`	Generate dataset labels
`seldat`	Select a specific subset of a dataset
`setdat`	Define a new dataset from an old one by replacing its data
`getdata`	Retrieve data from dataset
`getlab`	Retrieve object labels
`getfeat`	Retrieve feature labels
`renumlab`	Convert labels to numbers

Sets of objects may be supplied externally or may be generated by one of the data generation routines in PRTools (see Appendix B). Their labels may also be supplied externally or may be the result of a classification or a cluster analysis. A dataset containing 10 objects with 5 random measurements can be generated by:

```
>> data = rand(10,5);
>> a = dataset(data)
10 by 5 dataset with 0 classes: [ ]
```

In this example no labels are supplied, therefore no classes are detected. Labels can be added to the dataset by:

```
>> labs = [1 1 1 1 1 2 2 2 2 2]'; % labs should be a column vector
>> a = dataset(a,labs)
10 by 5 dataset with 2 classes: [5  5]
```

Note that the labels have to be supplied as a column vector. A simple way to assign labels to a dataset is offered by the routine `genlab` in combination with the Matlab `char` command:

```
>> labs = genlab([4 2 4],char('apple','pear','banana'))
>> a = dataset(a,labs)
10 by 5 dataset with 3 classes: [4  4  2]
```

Note that the order of the classes has changed. Use the routines `getlab` and `getfeat` to retrieve the object labels and the feature labels of `a`. The fields of a dataset can be inspcted by the converting it into a structure, e.g.:

```
>> struct(a)
        data: [10x5 double]
     lablist: [2x4 cell]
        nlab: [10x1 double]
     labtype: 'crisp'
     targets: []
     featlab: []
     featdom:
       prior: []
        cost: []
     objsize: 10
    featsize: 5
       ident: 10x1 cell
     version: [1x1 struct]  '10-Mar-2015 09:36:22'
        name: []
        user: []
```

In the online information on datasets (`help datasets`), the meaning of these fields is explained. Each field may be changed by a `set*`-command, e.g.

```
>> b = setdata(a,rand(10,5))
```

Field values can be retrieved by a similar `get*`-command, e.g.

```
>> classnames = getlablist(a)
```

In `nlab` an index into the list of class names, `lablist`, is stored for each object. Note that this

list is ordered alphabetically. The size of a dataset can be found by both `size` and `getsize`:

```
>> [n,p] = size(a);
>> [n,p,c] = getsize(a);
```

The number of objects is returned in `n`, the number of features in `p` and the number of classes in `c`. The class prior probabilities are stored in `prior`. By default, these are set to the class frequencies if the field is empty. Data in a dataset can also be retrieved by `double(a)` or simply by `+a`.

Exercise 1.1 Have a look at the help of `seldat`. Note that it has many input parameters. In most cases, you can ignore input parameters of functions that are of no interest to you; the default values are often good enough. Use the routine to extract the `banana` class from `a`, store it again in `a` and check this by inspecting `+a`.

Datasets can be manipulated in many ways, comparable to MATLAB matrices. So `[a1; a2]` combines two datasets, provided that they have the same number of features. The feature set may be extended by `[a1 a2]` if `a1` and `a2` have the same number of objects.

Exercise 1.2 Generate 3 new objects for each of the classes `'apple'` and `'pear'` and add them to the dataset `a`. Check if the class sizes change accordingly.

Exercise 1.3 Add a new, 6^{th} feature to the dataset `a`.

Another way to inspect a dataset is to make a scatterplot of the objects in the dataset. For this the function `scatterd` is supplied. This plots each object in a dataset in a 2D graph, using a coloured marker when class labels are supplied. When more than two features are present in the dataset, only the first two are used. For obtaining a scatterplot of two other features they have to be explicitly extracted first, e.g. `a1 = a(:,[2 5]);`. With an extra option `'legend'` one can add a legend to the figure, showing which markers indicate which classes.

Exercise 1.4 Use `scatterd` to make a scatterplot of the features 2 and 5 of dataset `a`. Try using the `'legend'` option.

Exercise 1.5 Next, use `scatterdui` to make a scatterplot of `a` and use its buttons to select features (note that `'legend'` is not a valid option here).

Exercise 1.6 It is also possible to create 3D scatterplots. Make a 3-dimensional scatterplot by `scatterd(a,3)` and try to rotate it by the mouse after pressing the Rotate 3D toolbar button.

Exercise 1.7 Use one of the procedures described in the PRTOOLS summary (section B) to create an artificial dataset of 100 objects. Make a scatterplot. Repeat this a few times.

Exercise 1.8 Load the 4-dimensional Iris dataset by `a = iris` and make scatterplots of all feature combinations using the `gridded` option of `scatterd`. Try also all feature combinations using `scatterdui`.

Plot, in a separate figure, the one-dimensional feature densities by `plotf`. Identify visually the best combination of two features to classify this data. Create a new dataset `b` that contains just these two features. Create a new figure using the `figure` command and plot a scatterplot of `b`.

Exercise 1.9 Generate a dataset that consists of two 2D uniformly distributed classes of objects using the `rand` command (see `help rand`). Transform the sets such that for the `[xmin xmax; ymin ymax]` intervals the following holds: `[0 2; -1 1]` for class `1` and `[1 3; 1.5 3.5]` for class `2`. Generate 50 objects for each class. An easy way is to do this for the x and y coordinates separately and combine them afterwards. Label the features `'area'` and `'perimeter'`.

Check the result using `scatterd` and by retrieving object labels and feature labels.

Exercise 1.10 Generate a dataset using `gendatb` containing 10 objects per class. Enlarge this dataset to 100 objects per class by generating more data using the `gendatk` and `gendatp` commands. Compare the scatterplots with a scatterplot of 100 objects per class directly generated by `gendatb`. Explain the difference.

You should now be familiar enough with PRTools to work through most exercises in the remainder of this manual. For more information on PRTools, please see Appendix B.

1.2 The Gaussian distribution

1.2.1 Estimation

PRTools offers a function `gauss` allowing you to create datasets with samples drawn from a Gaussian distribution with specified mean μ and variance σ^2 (or, for multivariate Gaussian distributions, mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$).

Exercise 1.11 Generate 1000 objects from a 1D Gaussian distribution with zero mean and unit variance. Make a scatterplot. Do the same for data from a Gaussian distribution with mean 5 and variance 2.

The Gaussian distribution is very widely used, as was discussed in the lectures. There is a good reason for this: the *central limit theorem* says that the sum of a large number of independently distributed samples will approximately be distributed according to a Gaussian. In real world experiments, it is assumed that the things we measure are often such sums of independently distributed values. Furthermore, *noise* on physical measurements is often assumed to be Gaussian.[1]

[1] Which may often be justified, but coincidentally also makes life very easy for the experimenter.

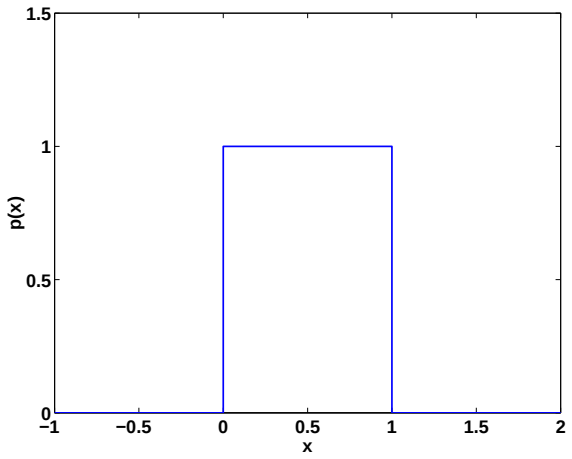


Figure 1.1: The pdf of a uniform distribution on $[0, 1]$.

Exercise 1.12 (a) Generate 100 random values between 0 and 1 using `rand`. This function generates samples drawn from a *uniform distribution*, which is constant in some range, and 0 outside. Figure 1.1 shows the probability density function of a uniform distribution on $[0, 1]$.

First, if we would average these 100 values, what value would you predict for the average? Compute this.

(b) Now repeat this 1000 times and store the averages in a vector `x`:

```
n = 100;
for i = 1:1000
  x(i) = mean(rand(n,1));
end;
```

Make a histogram of this vector, using `hist(+x)`. What does the histogram remind you of?

(c) Compute the mean and the variance (`var`) of the distribution. .

(d) When we would have generated 1000 objects at a time instead of 100, what would you have expected for the mean and variance of the average? Try it.

(e) When we would have repeated the experiment 10,000 times instead of 1000, what would you have expected for the mean and variance of the average?

The previous exercise showed you that if you draw a large number n of independent samples from any distribution (in this case, a uniform one), their average will tend to have a Gaussian distribution. In fact, if the original distribution has mean μ and non-zero variance σ^2, then:

$$\frac{s - n\mu}{\sqrt{n\sigma^2}} \quad \stackrel{n\to\infty}{\to} \quad N(0, 1). \tag{1.1}$$

where s is the *sum* of the samples. This is the so-called *central limit theorem*. It shows that the Gaussian distribution is likely to pop up on many occasions.

1.2.2 Visualisation

It is always informative to visualise a density after you have estimated its parameters. If such a visualisation is combined with a scatter plot of the data, it is often immediately obvious whether the density estimate is a good one. Unfortunately, we can only visualise densities in at most 3 dimensions.

Exercise 1.13 PRTools provides a routine `plotm` to plot a density estimate mapping.
(a) Generate a 1D Gaussian dataset `a` and make a scatter plot. Estimate the parameters of the density using `w = gaussm(a,1);` and plot the density estimate on top of the data, using `plotm(w,1);`.
(b) Do the same for a 2D Gaussian dataset. You can now use `plotm(w,2);` to plot a 2D contour plot of the density on top of the scatterplot, or alternatively `plotm(w,3);` through `plotm(w,6);`.

OPTIONAL

1.2.3 Sphering

As discussed in the lectures, we can perform eigenanalysis on the covariance matrix of a Gaussian distribution's covariance matrix to get an idea of its main axes.

Exercise 1.14 Generate 1000 objects in a 2D Gaussian dataset `a` with zero mean and `Sigma = eye(2)`. Calculate the covariance matrix `C = cov(+a)` and its eigenvectors and eigenvalues, `[E,D] = eig(C)`. If you repeat this a number of times, what do you notice about the eigenvectors, which are stored as *columns* in `E`? Why do you think this is?

Exercise 1.15 **(a)** Repeat the experiment above, with `Sigma = [3 0; 0 1]`. How do the eigenvectors behave now? And the eigenvalues, on the diagonal of `D`?

(b) Make a 2D scatterplot of one of these datasets. Plot in the eigenvectors (using `hold on; plot([0 E(1,1)],[0 E(2,1)],'r-')` if the eigenvectors are stored as columns in `E`). Also plot the eigenvectors multiplied by the eigenvalues.

(c) Make the same plot for `Sigma = [3 -1.5; -1.5 2]`.

In the lectures, it was discussed how eigenanalysis of the covariance matrix can be used to *sphere* the data.

Exercise 1.16 Generate a dataset `a` containing 1000 samples from a 2D Gaussian distribution with `mu = [0 0]` and `Sigma = [3 1.5; 1.5 2]`. Use eigenanalysis to sphere the data: `[E,D] = eig(cov(+a)); b = a*E*(inv(sqrt(D)));` Inspect the original data and the sphered data.

END OPTIONAL

1.3 Parametric density estimation

As discussed in the lectures, we can *estimate* the expectation and variance of any dataset. The question is how reliable these estimates are. We can learn about this by repeating experiments a large number of times and noting how the estimates change.

Exercise 1.17 (a) Generate 1000 1D Gaussian datasets with n samples, zero mean and unit variance. For each dataset `i`, estimate the expectation and variance and store the value in `m(i)` and `v(i)`. Afterwards, calculate the mean and standard deviation (`std`) of the arrays `m` and `v`. Repeat this for $n = 10$, 50, 100, 250 and 500. Finally, plot the values you found for the mean and standard deviation, as a function of n, using `errorbar`:

```
clear all;
n = [10 50 100 250 500];
for i = 1:length(n)
   for j = 1:1000
     a = gauss(n(i),0,1);
     m(j) = mean(a); v(j) = var(a);
   end;
   mean_m(i) = mean(m); std_m(i) = std(m);
   mean_v(i) = mean(v); std_v(i) = std(v);
end;
figure(1); errorbar(n,mean_m,std_m);
figure(2); errorbar(n,mean_v,std_v);
```

(b) How many objects would you say are needed to reliably estimate the mean of a 1D Gaussian dataset? And the variance?

If all went well, in the exercise above you noticed that the standard deviation of the estimate of the mean goes down slowly as n increases. In fact, using the central limit theorem, it's easy to show that if you draw n samples from a $N(0, \sigma)$ distribution, then the standard deviation of the estimate of the mean has itself a Gaussian distribution $N(0, \frac{\sigma}{\sqrt{n}})$.

The method followed above is quite general: if you have sufficient data (or if you can generate data, as was the case above), it is always a good idea to repeat an experiment a number of times and get an idea of the spread of the outcome.[2]

Exercise 1.18 (a) Now generate 10 objects from a 2D, 5D and 10D Gaussian distribution. Compute the covariance matrix. Plot the *condition number* of the estimated covariance matrix (see `cond`) as a function of the number of dimensions d. The condition number of a matrix gives an indication of whether the matrix is becoming *singular* and, as a result, how inaccurate taking the inverse will be. If the condition is bad, the matrix is nearly singular, its inverse cannot be calculated, and you will have a problem in the Gaussian function.

What do you see?

[2] In fact, this should be standard practice for any reported measurement or simulation result.

Exercise 1.19 (a) Consider a Gaussian distribution with mean `mu = [0 0]` and covariance matrix `Sigma = [1 0; 0 4]`. Plot a 2D scatterplot by generating a large amount of objects from this distribution.

(b) Now change the covariance matrix into `Sigma = [1 1; 1 4]` and plot the density again. How do the non-zero values for the covariances influence the distribution?

(c) Can you tell what is wrong with the definition of the covariance matrix `Sigma = [1 1; 0 4]`?

(d) Can you predict what the distribution will look like when the covariance matrix `Sigma = [1 1; 1 10000]`? Check it! (you will probably need the command `axis equal` to force the axis of the scatterplot to have equal ranges).

1.4 Non-parametric density estimation

1.4.1 Histograms

The simplest way to estimate the density of one measurement is by plotting a *histogram*, which you already encountered in Exercise 1.12. It is easiest to do this for one measurement (MATLAB has a command `hist`), but you can create a histogram plot of 2 measurements at the same time.

The main problem in using histograms is choosing the right bin size. Making the bins too broad will hide useful information within single bins; making them too small will result in bins containing too few samples to be meaningful.

Exercise 1.20 (a) Generate 10 1D Gaussian datasets and calculate the average histogram and the standard deviation:

```
for i = 1:10
  a = gauss(n,0);
  h(i,:) = hist(+a,-5:5);
end;
errorbar (-5:5, mean(h), std(h));
```

with different numbers of samples `n` (e.g. 10, 100 and 1000).

(b) For what `n` do you think the histogram starts giving a good impression of the true density?

(c) Repeat the above for data obtained from a Laplacian distribution (which is more peaked than the Gaussian); use `a = laplace(n,1)` to generate `n` samples. Does the same hold? Why?

Note how you used 2 dimensions to visualise the 1D histogram. Similarly, we can create 2D histograms using 3 dimensions. In practice, visualising 3D data or a function of 2D data is often as far as we can go with computers.

Exercise 1.21 Generate a 2D Gaussian dataset and use the `hist2` function provided:

```
a = gauss(100,[0 0]);
hist2(a);
```

Note that `hist2` is not a standard MATLAB function, so it is a bit less flexible than the standard `hist` function. To have a look around the histogram, try `help rotate3d`.

(a) For the standard number of bins (10×10), how many samples would you say are needed to get a good idea of the shape of the histogram? Try this by generating various numbers of samples and inspecting whether the histogram changes if you repeat the experiment.

(b) You can also play with the number of bins; see `help hist2`. For 5×5 bins, how many samples do you need? Is the representation still accurate? And for 20×20 bins?

1.4.2 Parzen density estimation

As discussed during the lectures, the Parzen method estimates a density by centering a local kernel on each object and adding the contributions of all kernels. In PRTOOLS, these kernels are Gaussian.

Exercise 1.22 (a) Start by creating a simple dataset with:

```
>> a = gendats([20 20],1,8);
```

(Type `help gendats` to understand what type of data we have now.)

(b) Define the width parameter h for the Gaussian kernel:

```
>> h = 0.5;
```

(c) The function `parzenm` estimates a density for a given dataset. In most cases a PRTOOLS `dataset` is labeled, and these labels are used in the function `parzenm` to estimate a density for each class. To define a Parzen density estimator with a certain width parameter `h` on the entire dataset, ignoring labels, type:

```
>> w = parzenm(+a,h);
```

This mapping can now be plotted along with the data:

```
>> scatterd(+a); plotm(w,1);
```

If your graphs look a little "bumpy", you can increase the grid size PRTOOLS uses for plotting:

```
>> gridsize(100);
```

and try the above again.

(d) Plot the Parzen density estimate for different values of `h`. What is the best value?

(e) Save the figure as MATLAB file: `saveas(gcf,'figparzen','m')`.

When you want to evaluate a fit of a density model to some data, you have to define a goodness-of-fit measure. One possibility is to use the log-likelihood, defined as:

$$\mathrm{LL}(\boldsymbol{X}) = \log\left(\prod_i \hat{p}(\boldsymbol{x}_i)\right) = \sum_i \log\left(\hat{p}(\boldsymbol{x}_i)\right) \tag{1.2}$$

The better the data $\boldsymbol{x}$ fits in the probability density model $\hat{p}$, the higher the values of $\hat{p}(\boldsymbol{x})$ will be. This will result in a high value of $\sum_i \log\left(\hat{p}(\boldsymbol{x}_i)\right)$. When we have different probability density estimates $\hat{p}$, we will use the one giving the highest value of LL.

Note that using different values for the width parameter h in Parzen density estimation results in different estimates $\hat{p}$. Using the log-likelihood as a criterion, we can optimize the value of this free parameter h to maximise LL.

To get an honest estimate of the log-likelihood, the log-likelihood (1.2) has to be evaluated on a *validation set*. That means that new data has to be created (or measured) from the same distribution as the training data came from.

Exercise 1.23 Use the data from the same distribution as in the previous exercise to train a Parzen density estimator for different for different values of h. Compute the log-likelihood of this training set given the estimated densities (for different h):

```
a = gendats([20 20],1,8);                    % Generate data
hs = [0.01 0.05 0.1 0.25 0.5 1 1.5 2 3 4 5]; % Array of h's to try
for i = 1:length(hs)                         % For each h...
  w = parzenm(+a,hs(i));                     %   estimate Parzen density
  LL(i) = sum(log(+(a*w)));                  %   calculate log-likelihood
end;
plot(hs,LL);                          % Plot log-likelihood as function of h
```

(since `w` is the estimated density mapping `w`, the estimated density $\hat{p}$ for objects in a dataset `a` is given by `+(a*w)`).

(a) What is the optimal value for h, i.e. the maximal likelihood? Is this also the best density estimate for the dataset?

Exercise 1.24 (a) Use the same data as in the previous exercise, but now split the data into a training and test set of equal size. Estimate a Parzen density on the training set and compute the Parzen density for the test set. Compute the log-likelihood on both the training and test sets for $h = [0.1, 0.25, 0.5, 1, 1.5, 2, 3, 4, 5]$. Plot these log-likelihood vs. h curves:

```
a = gendats([20 20],1,8);           % Generate data
[trn,tst] = gendat(a,0.5);          % Split into trn and tst, both 50%
hs = [0.1 0.25 0.5 1 1.5 2 3 4 5];  % Array of h's to try
for i = 1:length(hs)                % For each h...
  w = parzenm(+trn,hs(i));          %   estimate Parzen density on trn
  Ltrn(i) = sum(log(+(trn*w)));     %   calculate trn log-likelihood
  Ltst(i) = sum(log(+(tst*w)));     %   calculate tst log-likelihood
end;
plot(hs,Ltrn,'b-'); hold on;   % Plot trn log-likelihood as function of h
plot(hs,Ltst,'r-');            % Plot tst log-likelihood as function of h
```

What is a good choice for h?

Exercise 1.25 (a) Use the same data as in the previous exercise, but now let PRTools find the optimal h using leave-one-out cross-validation. This can simply be performed by not specifying h, i.e. calling `w = parzenm(+a)`. To find out what value of h the function actually uses, you can call `h = parzenml(+a)`. Does the h found correspond to the one you found to be optimal above?

OPTIONAL

Exercise 1.26 Use the same procedure as in Exercise 1.22. Change the number of training objects from 20 per class to 100 per class. What is now the best value of h?

END OPTIONAL

1.4.3 Nearest neighbour density estimation

Exercise 1.27 (a) Again generate a dataset `a = gendats([20 20],1,8)` and apply the `knnm` density estimator for $k = 1$: `w = knnm(a,1)`. Plot the density mapping using `scatterd(a); plotm(w,1)`. What did you expect to see for $k = 1$? Why don't you see it?

(b) Increase the grid size, e.g. `gridsize(500)`, and plot again.

(c) Try different values of k instead of 1. Judging visually, which do you prefer?

(d) How could you, in theory, optimise k?

Exercise 1.28 Which do you prefer: the Parzen density estimate or the nearest neighbour density estimate?

.

Day 2

Classification

As we discussed in today's lecture, to classify an object $\boldsymbol{x}$, you have to assign it to the class ω_i with the highest class posterior probability $p(\omega_i|\boldsymbol{x})$. In most cases this class posterior probability is rewritten in terms of the class-conditional probability $p(\boldsymbol{x}|\omega)$ using Bayes' rule. First, we will construct a simple normal density plug-in Bayes classifier by hand.

2.1 Bayesian classification

You will now construct a simple Bayesian classifier. The setup of the algorithm will be:

1. create a train and test dataset;
2. fit, to each of the classes in the training set, a class conditional probability density model;
3. estimate the class-conditional probability density for each of the test objects;
4. compute the posterior probability for each of the objects;
5. assign to each test object the class label corresponding to the class with the highest posterior probability.

Exercise 2.1 The Golub set is a classic microarray dataset[1] containing acute lymphatic leukemia (ALL) and acute myeloid leukemia (AML) samples. There is a training set `a` of 38 samples and a test set `b` of 34 samples. This is one of the first microarray datasets to which classifiers were applied.

The data originates from Affymetrix HU-6800 oligonucleotide microarrays, normalised by the Affymetrix software. Originally, there were 7129 genes measured per microarray, but a variation filter (picking out genes that show some variation) left 3051 genes. The data have been $\log_2$-transformed.

[1] Golub TR, Slonim DK, Tamayo P, Huard C, Gaasenbeek M, Mesirov JP, Coller H, Loh ML, Downing JR, Caligiuri MA, Bloomfield CD and Lander ES. Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *Science* 286(5439):531-537, 1999.

An intriguing problem is how to select a small number of predictive genes from the 3051 available; this will be discussed on day 3. For now, we will just pick genes number 1413 and 738.

(a) Load the data and select the two predictive features:

```
>> load golub
>> a = a(:,[1413 738]);
>> b = b(:,[1413 738]);
```

Generate this dataset and make scatterplots of `a` and `b`.

(b) Extract objects from each of the classes by first extracting the labels from the data. Next, `find` the indices of the objects of each of the classes and finally extract the corresponding objects from the dataset:

```
lab = getlab(a);
I = find(lab==1);
a1 = +a(I,:);
```

Write a script which performs these tasks. Extract the two classes from dataset `a` and check it by plotting each separate class in a scatterplot again.

(c) Now we have two data sets, each one corresponding to one of the classes. Estimate on each of the classes a Gaussian density and call the resulting mappings `w1` and `w2`, respectively. Visualise the density estimates in the scatter plot.

(d) For each object in `b` estimate the class-conditional probability for class `1` and for class `2`: `phat1 = +(b*w1); phat2 = +(b*w2);`. Inspect the results.

(e) Estimate the prior probabilities of each of the two classes using the training set.

(f) For each object in `b`, compute the posterior probability. Use the second output argument of the MATLAB function `max` (have a look at `help max`) to find for each object for which class the posterior is the largest. In this way, you assign to each object a label `1` or `2`, according to the highest posterior probability.

Congratulations! You just have constructed your own classifier!

For this simple data, the assumption of a Gaussian distribution per class is reasonably OK. For other datasets we might need more complex and more flexible density estimates, for instance the Parzen density or the nearest neighbor density.

Exercise 2.2 How do you have to adapt your script from exercise 2.1 to not use the Gaussian density, but the Parzen density?

Exercise 2.3 Count the number of differences between the true labels of the test set and the labels you obtained from your classifier. If you have two vectors `lab1` and `lab2` both containing labels `1` and `2`, you can find the differences using::

```
>> diffl = (lab1 ~= lab2);
```

More extensive methods of evaluating the performance of your classifer are more easily performed using a PRTOOLS builtin classifier.

Exercise 2.4 The classifier you constructed above is (roughly) implemented in PRTOOLS as the quadratic normal density-based classifier or `qdc` (section 2.5). That is, `w = qdc(a); testc(b*w)` trains this classifier on the Golub training set and tests it on the test set.

(a) Train the `qdc` classifier on the training set, and test it on both the training set and the test set. What do you notice?

(b) Predict classifier performance on new data by performing 10-fold cross-validation on the trainset: `crossval(a,qdc,10,1)`. How does this compare to the actual test error? Also perform 20-fold cross-validation.

(c) Inspect the Receiver-Operator Characteric (ROC) curve for the training set, using `plote(roc(a,w),'nolegend')`. Do the same for the test set.

OPTIONAL

Exercise 2.5 Another well-known microarray dataset is that of Khan[2]. Load it (`load khan`), look for some interesting features and repeat the above procedures to build and evaluate a classifier.

END OPTIONAL

2.2 The logistic classifier

Exercise 2.6 (a) Given a two-class classification problem with classes ω_1 and ω_2, and given that you know $p(\omega_1|\boldsymbol{x})$, what will be the function for $p(\omega_2|\boldsymbol{x})$? (this is simpler than you may think).

(b) Show that the formula for the logistic classifier as given in the lecture:

$$\log\left(\frac{p(\omega_1|\boldsymbol{x})}{p(\omega_2|\boldsymbol{x})}\right) = w_0 + \boldsymbol{w}^T\boldsymbol{x} \tag{2.1}$$

can be rewritten using a logistic function.

Exercise 2.7 (a) Generate a dataset using `gendats([50 50],2,1)`. Can you explain what type of data distribution you will get, after looking at `help gendats`?

(b) Now train a logistic classifier `loglc` and plot the classifier output using `plotm(w)`. Does this classifier fit the data?

(c) Make a new dataset using `gendats([50 50],2,5)`. What is the difference between this and the previous dataset? Again train a logistic classifier and plot the classifier. What is the difference between this classifier and the one you trained before?

[2]Khan J, Wei JS, Ringner M, Saal L, Ladanyi M, Westermann F, Berthold F, Schwab M, Antonescu C, Peterson C and Weltzer P. Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural network. *Nature Medicine* 7:673-679, 2001.

2.3 Classifiers in PRTools

In PRTools the classifiers based on the Gaussian class distribution, the Parzen density and the nearest neighbor density, are already implemented. They can easily be trained using:

```
>> w1 = qdc(a);
>> w2 = parzenc(a);
>> w3 = knnc(a);
```

The name `qdc`, for the quadratic classifier, will be explained later. For simplicity we left out the additional optional parameters, to specify the width parameter h in the Parzen density or the number of neighbors k in the nearest neighbor classifier.

Furthermore, the toolbox provides simple routines to show the decision boundary for classifiers on two-dimensional data:

```
>> scatterd(a);
>> plotc(w1);
```

Finally, one can find the output classification labels for an object by `labeld` and find the classification performance by `testc`:

```
>> lab = a*w1*labeld
>> e = a*w1*testc
```

Exercise 2.8 (a) Generate the following classification problem:

```
>> a = gendatb([20 20]);
```

Create a scatterplot. Can you imagine what would happen when you apply the different classifiers, `qdc`, `parzenc` or `knnc`?

(b) Train the three classifiers, and plot their decision boundaries in the scatterplot and estimate the classification error for the three classifiers using `testc`. Check if your predictions and expectations are correct.

Exercise 2.9 Change the width parameter h in the `parzenc` and the k in the `knnc`. Plot the decision boundaries of the classifiers. What happens with the classifier when you increase h and k? How does this influence the classification error?

2.4 The curse of dimensionality

You should now have some feeling for what it means to construct a classifier and estimate the classification error. In this section we will investigate a very important phenomenon in pattern recognition: the curse of dimensionality. It appears that it is not always good to increase the number of features. When more and more features are used in a classification problem, the performance on an independent test set will deteriorate.

Exercise 2.10 (a) Generate a training dataset according to `a = gendats([10 10],2,2)` and a test set using `b = gendats([1000 1000],2,2)`. Create a scatterplot of this data. Read the `help` and explain what the second and third input parameter mean. Vary them and check if your explanation is correct.

(b) Train a one-nearest neighbor classifier `knnc` on `a` and test the classifier on both `a` and on `b`. Why are the performances different?

(c) Increase the dimensionality of the dataset by `a = gendats([10 10],5,2)` and `b = gendats([1000 1000],5,2)`. How will the Bayes error change for this dataset?

(d) Train the classifier again and test it on the test set. How do the errors change in comparison to the lower dimensional training set?

(e) Vary the dimensionality of the dataset between 2 and 100, say, `[2 3 5 10 25 100]`. Create a plot of the test error as function of the dimensionality. To get a clearer picture, you can average your results over several random draws of a train and test set. What dimensionality do you prefer?

Now we will investigate in more detail the special case where the class-conditional distributions are assumed to be Gaussian distributions.

2.5 Gaussian-based discriminant functions

In section 2.1, you built a classifier by estimating the Gaussian densities associated with a two-class dataset, applying Bayes' rule and assigning an object to the class with the largest *a posteriori* probability. In the lecture, we derived a closed form expression for $g_i(\boldsymbol{x})$, the function employed in the discriminant rule (under the assumption that $\boldsymbol{x}$ and $\boldsymbol{\mu}_i$ are column vectors):

$$g_i(\boldsymbol{x}) = \log(p(\omega_i)) - \frac{1}{2}\log(\det(\Sigma_i)) - \frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_i)^T \Sigma_i^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_i), \tag{2.2}$$

with $p(\omega_i)$ the *a priori* class probability, $\boldsymbol{\mu}_i$ the mean and Σ_i the covariance of class ω_i. The discriminant rule that is employed for classification is to assign object $\boldsymbol{x}$ to class ω_i if $g_i(\boldsymbol{x}) > g_j(\boldsymbol{x}),\ \forall j \neq i$.

Exercise 2.11 In the lecture it was mentioned that in case of a two-class problem, the quadratic discriminant function can be written as:

$$f(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{W} \boldsymbol{x} + \boldsymbol{w}^T \boldsymbol{x} + w_0. \tag{2.3}$$

(a) Derive the expressions for $\boldsymbol{W}$, $\boldsymbol{w}$ and w_0.

Because the function $f(\boldsymbol{x})$ is quadratic in terms of $\boldsymbol{x}$ it is called a *quadratic* classifier. As it was derived using the assumption of Gaussian (or normal) class distributions using Bayes' rule, this classifier is called the Quadratic Bayes normal-density based classifier, or `qdc` in PRTools.

Exercise 2.12 In the lecture it was stated that in case of a two-class problem and equal covariance matrices, the discriminant function can be written as:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x} + w_0. \tag{2.4}$$

(a) Derive the expressions for $\boldsymbol{w}$ and w_0 given in the lecture.

Exercise 2.13 In the lecture it was explained that the linear classifier with the identity covariance matrix $\hat{\Sigma} = \sigma^2 \boldsymbol{I}$, is identical to the nearest mean classifier.

(a) Give the formula for the nearest mean classifier.

(b) Show that this nearest mean classifier is indeed equivalent to a linear classifier with identity covariance matrix.

The linear discriminant classifier, the Fisher classifier and the nearest mean classifier are also implemented in PRTools, and they are called the Linear Discriminant classifier, `ldc`, Fisher classifier, `fisherc` and the nearest mean classifier, `nmc`. Summarizing, today we have encountered the following classifiers:

PRTools name	description
`nmc`	Nearest mean classifier
`ldc`	Linear discriminant classifier
`qdc`	Quadratic Bayes Normal classifier
`fisherc`	Fisher classifier
`parzenc`	Parzen density classifier
`knnc`	k-nearest neighbor classifier

All these classifiers can be plotted using `plotc` and their classification error can be computed using `testc`.

Exercise 2.14 **(a)** Load the `cigars` dataset. Train a `qdc` and a `ldc` classifier and call the classifiers `wq` and `wl` respectively. Make a scatterplot of the data, and plot both classifiers using: `plotc(wq); plotc(wl)`. What do you think would be the shape of the optimal decision boundary? Which classifier is therefore to be preferred? **(b)** Check the classification error of both classifiers on the training set.

Exercise 2.15 Plot the decision boundaries of a `ldc` and `qdc` trained on the banana data set with 100 objects per class: `a = gendatb([100 100])`.

(a) Is it appropriate to assume a Gaussian distribution for the data? What error rate do you achieve?

(b) Why is this error rate so high?

2.6 Fisher classifier

In the following exercises we will focus on a classifier that is not based on the assumption that the class-conditional probabilities are Gaussian distributions: the Fisher linear discriminant.

Exercise 2.16 For the dataset: `a = gendats([10,10],2,3)`, train a Fisher classifier (`fisherc`), visualize the dataset and the decision boundary. Also estimate the error on the train and test set.

Exercise 2.17 Use the same data as in Exercise 2.16. Instead of the Fisher classifier, use the normal-based linear classifier `ldc`. What are the differences in decision boundary and classification performance? Explain.

OPTIONAL

Exercise 2.18 (a) Generate a Highleyman dataset with `a = gendath`. Compute the Fisher classifier `w` and classify the training set using `c = a*w`. By `d = c(:,1)*invsigm` all distances to the classifier can be found. Why? Inspect the distribution of `d` using `plotf(d,1)`.

(b) Compute the Fisher criterion for these distances. If you cannot find a routine for this, write it.

(c) Compute also the distances to the Nearest Mean Classifier `nmc` and the resulting value for the Fisher criterion. Verify the statement that the Fisher classifier optimizes the Fisher criterion.

END OPTIONAL

2.7 Classification trees

Classification trees are an example of an entirely different type of classifier. They partition the feature space into rectangular regions using splits aligned with the axes.

Exercise 2.19 (a) Reload the Golub data and select the two predictive genes:

```
>> load golub
>> a = a(:,[1413 738]);
>> b = b(:,[1413 738]);
```

(b) Train a `treec` classifier on the training set `a` and evaluate its performance on both `a` and `b`.

(c) Plot the data and the decision boundary of the classification tree. Do you understand the shape of the decision boundary? Explain.

(d) The complexity of a classification tree is determined by the number of leaf nodes. Pruning can be used to generate less complex models. Train classification trees on `a` for various values 0,1,..,5 of the `prune` parameter and evaluate their performance on both `a` and `b`. Which value of `prune` gives the best test results? Plot the data and the decision boundary of the "optimal" model and explain why this model is better.

(e) Optional: repeat the analysis from the previous question on the complete `golub` training data. Try out various options for the binary splitting criterion `crit`. Does it have a large influence on the performance of the model on the test data? Can you explain the results?

2.8 The scaling problem

In this last section we have a brief look at the scaling problem. It appears that some classifiers are sensitive to the scaling of features. That means, that when one of the features is rescaled to very small or very large values, the classifier will change dramatically. It can even mean that the classifier is not capable of finding a good solution. Here we will try to find out, which classifiers are sensitive to scaling, and which are not.

Exercise 2.20 **(a)** Generate a simple 2D dataset (for instance, using `gendatb`), assign it to a variable `train` and plot the decision boundaries of the six classifiers listed in section 2.5 and a decision tree. Use $k = 1$ for the `knnc`.

(b) Make a new dataset in which the second feature is 10 times larger. Do this as follows:

```
>> newtrain = train;
>> newtrain(:,2) = 10*newtrain(:,2)
```

Train the seven classifiers again and plot the decision boundaries.

(c) Which classifiers are affected by this rescaling of the feature space? Why are they affected, and others not?

(d) Is scale invariance an advantage or a disadvantage?

Day 3

Feature selection and extraction

In the lecture, feature selection and extraction were discussed. This afternoon you are going to apply both types of method to a number of simple and a number of more complicated problems.

3.1 Feature selection

In the lectures, it was discussed how for feature selection (i.e. selecting individual measurements), you basically need two things: a criterion function which tells you how good a subset of measurements is, and a search algorithm: a way of creating such subsets. In PRTools, functions to do both are present. The `fsel` function is a function which creates a *mapping* `w` which can be applied to data. For example, if you select $d = 5$ measurements out of the $p = 10$ measurements present in dataset `a` using `w = fsel(a,'individual','NN',5)`, then `b = a*w` will give you a dataset `b` with 5 measurements. It also returns a list of ranked features if you call it like this:

```
[w,list] = fsel(a);
```

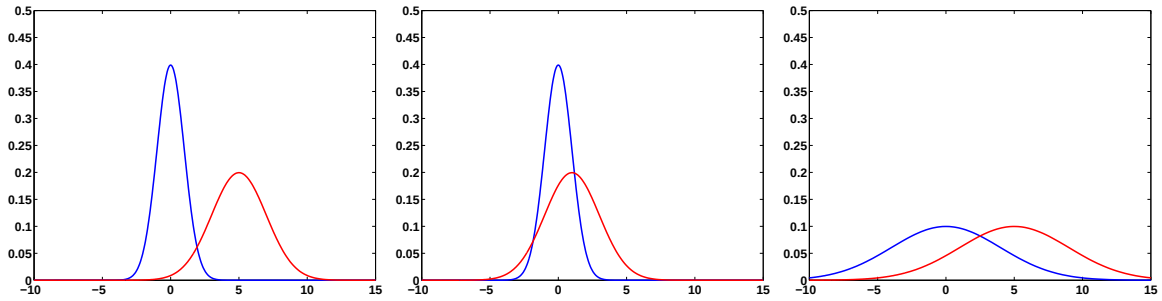
This allows you to create a dataset `b` with the best `d` features like this:

```
b = a*w(:,1:d);
```

By default, `fsel` uses the simplest *search algorithm* available: the "d best" approach. Smarter search algorithms are:

- forward selection, by `fsel(a,'forward');`
- backward selection, by `fsel(a,'backward');`
- plus-l-takeaway-r selection, by `fsel(a,'+l-r');`
- branch & bound selection, by `fsel(a,'b&b');`

PRTools also has a number of feature selection *criterion functions*. All of them are based on optimising a final *classification* result. However, if you have another problem, you can always define your own criterion, as was discussed in the lectures. The two criteria you will use in the exercises below are:



(a) (b) (c)

Figure 3.1: The Mahalanobis distance in (b) and (c) will be smaller than that in (a), as in (a) the means are the furthest apart and the variances are the smallest.

- `'NN'`, the performance of the 1-nearest neighbour classifier. This very simple classifier assigns to a new object the label of the closest object in the training set. In PRTools, it is also possible to train a classifier of your choice for any subset of features and use classification error as a feature selection criterion. However, computing these for any reasonable dataset takes a long time.

- `'maha-s'`:

$$J_{\texttt{maha-s}} = (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T \left(\frac{\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2}{2} \right)^{-1} (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) \tag{3.1}$$

  This distance is also called the *Mahalanobis distance*. The criterion looks complicated, but the idea behind it is simple: see the slides on scatter matrices. For high classification performance, the two densities need to be far apart. For this to happen, the distance between the two means $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ needs to be large. At the same time, the covariance matrices $\boldsymbol{\Sigma}_1$ and $\boldsymbol{\Sigma}_2$ (which indicate the spread around the means) should be small. Figure 3.1 illustrates this.

  Note that J_D is only defined for 2 classes. For more than 2 classes, J_D is calculated for all possible pairs of classes, and the criterion is the sum of all these values (hence `'maha-s'`).

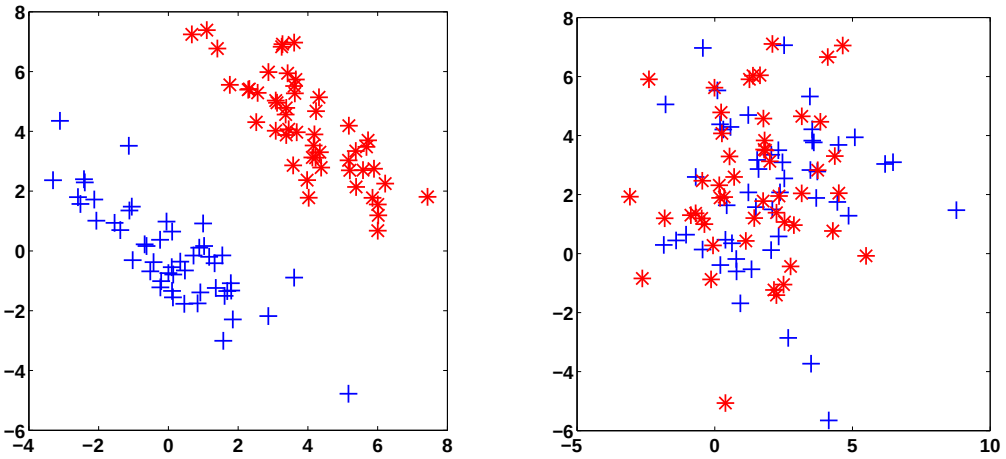
Exercise 3.1 (a) Load the `biomed` dataset and use `fsel` with the various search algorithms listed above to select 2 features. Use the `'maha-s'` criterion. Do the search algorithms give the same results?

You can use this to make a scatterplot; use `scatterd (a*w)`.

(b) Do the same for the `iris` dataset, which has 4 features.

We will now illustrate some points about feature selection. The first dataset we will use is an artificial one. It is carefully constructed to contain p measurements, of which only 2 are useful for classification; see Figure 3.2 for an illustration. The trick is finding these 2 measurements.

Exercise 3.2 (a) Create 100 samples in a 10D "difficult" distribution using `a = gendatdd(100,10)`. Apply feature selection using the possible search algorithms and



(a) (b)

Figure 3.2: The `gendatdd` dataset contains 2 measurements useful for classification (a); all other measurements are useless (b).

the `'maha-s'` criterion. Which features are the best two? Confirm this by making a scatterplot using the best two features, using `b = a*w(:,[1 2]); scatterd(b);` (if your feature selection mapping is stored in `w`).

(b) Do all search algorithms agree on the best two features?

To test whether feature selection really works and how many features are useful, you can train a 1-nearest neighbour classifier on the selected features and see how well it performs. To get a good impression, we cannot just use the same data we used for feature selection to test things. Instead, we have to:

- make a *training set* to perform feature selection;
- train a classifier on the features found;
- make a *test set* and apply the classifier to that set.

Exercise 3.3 (a) Enter a script that

- creates a dataset `a` using `gendatdd` containing 200 objects and 10 measurements
- split it into a randomly drawn training set and a test set, using `[train,test] = gendat(a,0.5);`
- applies forward selection and the `'NN'` criterion (use the `list` output of `fsel`).
- for each subset of size $d = 1, \ldots, 10$, trains a 1-nearest neighbour classifier on the training set and tests it on the test set
- does this 5 times and plots errorbars for the test error:

```
a = gendatdd(200,10);                    % Generate difficult data
for r = 1:5                              % For 5 repetitions...
  [trn,tst] = gendat(a,0.5);             %   split into trn and tst
  [w,list] = fsel(trn,'forward','NN'); %   rank features in list
  for d = 1:10                           %   for each number of features...
    mapped_trn = trn*w(:,1:d);           %     create a training set
    mapped_tst = tst*w(:,1:d);           %     create a test set
    classifier = knnc(mapped_trn,1);     %     train a 1-NN classifier
    error(r,d) = testc(mapped_tst,...    %     and test it
      classifier);
  end;
end;
errorbar(1:10,mean(error),std(error)); % Plot the results
```

What is the optimal number of features?

In this exercise, you should have noticed that the error the classifier makes decreases quickly when the first few features are added, but *increases* again at the end. This might seem paradoxical: you select features based on the nearest neighbour criterion, then train a nearest neighbour classifier – but still the error rises! The effect is caused by the fact that you use only a finite number of samples to train the classifier. If you keep adding features, the classifier will adapt too much to the training set, making it perform worse on the test set. So, even with a criterion which perfectly matches the classifier you are going to use, you cannot be sure your feature selection method will give good results.

You have just performed a more or less standard experiment in feature selection: use a number of different criteria and algorithms, vary the number of features, train classifiers and test them and see where the optimum occurs. However, in real life things are never as easy as this. In the following exercise, you are to apply the script you wrote above to a real-world dataset.[1]

Exercise 3.4 (a) Load the `housing` dataset.[2] This real-world dataset contains 13 fields of information on areas in the suburbs of Boston. The goal is to predict whether the median price of a house in each area is larger than or smaller than $20,000.

There are 506 objects. The fields are shown in Table 3.1; have a look at them. Which one(s) would you predict to be the most informative?

(b) Split the data into a training set (70% of the original samples) and a test set (30%): `[trn,tst] = gendat(a,0.7)`. Apply your script. What is the optimum number of features?

(c) What features are found to be the best (use `list`)? Are they the same as the ones you thought would be the best? How about the worst features? Also, check whether

[1] Even this dataset was still selected by us to show you something. Most real data is pretty bad from an educational point of view.

[2] This dataset is often used to benchmark data analysis tools. It was created by Harrison, D. and Rubinfeld, D.L. for "Hedonic prices and the demand for clean air", Journal of Environmental Economics & Management, vol. 5, pp. 81-102, 1978. The dataset can be downloaded from the UCI Machine Learning repository at `http://archive.ics.uci.edu/ml/`.

#	Name	Description
1	CRIME	Crime rate per capita
2	LARGE	Proportion of area dedicated to lots larger than 25,000 square feet
3	INDUSTRY	Proportion of area dedicated to industry
4	RIVER	1 = borders the Charles river, 0 = does not border the Charles river
5	NOX	Nitric oxides concentration (in parts per 10 million)
6	ROOMS	Average number of rooms per house
7	AGE	Proportion of houses built before 1940
8	WORK	Average distance to employment centres in Boston
9	HIGHWAY	Highway accessibility index
10	TAX	Property tax rate (per $10,000)
11	EDUCATION	Pupil-teacher ratio
12	AA	$1000(A - 0.63)^2$, where A is the proportion of African-Americans
13	STATUS	Percentage of the population which has lower status

Table 3.1: The measurements in the `housing` dataset. The task is to predict whether the median house price in areas of Boston is smaller or larger than $20,000.

the feature selection algorithm ranks the features in the same order if you repeat the splitting of the data and the feature selection.

(d) Change the search algorithm from `'forward'` to `'backward'`. How do the conclusions change?

(e) Use forward selection again, but change the criterion from `'NN'` to `'maha-s'`. How do the conclusions change? Why?

(f) Write down the lowest test error average you have found so far.

This exercise showed you how feature selection can help you in *understanding* a problem by analysing the data: it tells you which measurements are informative and which aren't. At the same time, you will have noticed how dependent the results are on the choice of search algorithm and criterion. Therefore, never trust your results blindly; always think of how they might have been influenced by your choices.

You may have noticed that, on the `housing` data, it already took quite some time to perform feature selection. Still, the dataset was not particularly large: 506 samples of 13 features each. Better feature selection algorithms, such as plus-l-takeaway-r or branch & bound, can take even more time. Feature selection is in fact computationally a horrible problem; even with the computers we have today, feature selection on sets of, say, a couple of thousand samples of 100 measurements each can easily take days. This means that complicated feature selection algorithms can often not be applied to data in real problems.[3]

Exercise 3.5 (a) Load the Golub training and test sets and train a PAM classifier for a

[3]The fact that there is a lot of scientific literature on "optimal" feature selection methods (under certain assumptions, of course) might seem to contradict this. This only shows, however, that often scientists tend to work on finding interesting, complicated non-solutions rather than boring, simple solutions.

range of shrinkage factors:

```
>> load golub;
>> prwarning(0);
>> w = pamc(a,0:0.25:5,[],10,1);
>> testc(b*w)
```

This will take some time. Looking at the output, it seems PAM selects quite a few genes - how many?

(b) If you allow for a slightly higher cross-validation error, you can easily decrease the number of genes used. Pick a Δ for which you find the cross-validation error acceptable, retrain (replace `0:0.25:5` by the single value for Δ you selected) and retest. How many genes are selected now? Does the change have a large influence on the test set error?

(c) The fourth argument returned contains the indices of the non-shrunken genes, `nz`:

```
[w,delta,theta,nz] = pamc(a,delta,[],10,1);
```

Play with Δ, inspect `nz` and find a setting for which roughly 7 genes are selected.

(d) Experiment further with the Golub dataset and a number of alternative classifiers. Can you find a single best classifier?

3.2 Principal Component Analysis

In the lectures, Principal Component Analysis (PCA) was treated in depth. In this section, you will apply PCA to some simple problems.

Exercise 3.6 (a) Apply `pca` to the `gendatdd` data you used in the feature selection exercises (100 samples, 10 measurements). Plot the retained variance `v = pca(a,0)` as a function of the number of dimensions retained. Does this graph tell you anything? Why?

Exercise 3.7 (a) Based on the script you made for the large feature selection experiment[4] (Exercise 3.3), create a script which investigates classifier performance as a function of the number of dimensions d retained: simply replace the line

```
w = fsel(trn,'forward','NN');
```

by

```
w = pca(trn);
```

Now apply this script to the `housing` data, splitting it into a training set (70%) and a test set (30%):

```
a = gendatdd(200,10);
```

becomes

```
load housing;
```

[4] Copy & paste are your friends here...

and

```
[trn,tst] = gendat(a,0.5);
```

becomes

```
[trn,tst] = gendat(a,0.7);
```

(b) How do the results you find here (in terms of test error) compare to the test errors you obtained using feature selection? Why?

(c) According to test error, how many dimensions would you say are needed to obtain good performance?

(d) According to the amount of retained variance `v = pca(a,0)`, how many dimensions would you say are needed according to PCA?

(e) Inspect the first PCA basis vector, using `w.data.rot(:,1)`. What feature(s) are important according to PCA? Why? Use `var(a)` to learn more.

(f) Try PCA on normalised data: after loading `housing`, apply the following:

```
[n,p] = size(a);
a = (a - ones(n,1)*mean(a)) ./ (ones(n,1)*std(a));
```

or, equivalently,

```
a = a*scalem(a,'variance');
```

(what does this do?). Are your conclusions still valid?

The `nistdigs` dataset contains 16×16 pixel images of handwritten digits – pre-processed versions of the NIST database digits. There are 2,000 samples in total of 10 classes ("0" ... "9"). See the slides for a few examples.

Exercise 3.8 **(a)** Load the `nistdigs` dataset. Can you find out what the intrinsic dimensionality of this dataset is?

(b) Inspect all PCA basis vectors, using `show(w)`. What do you observe?

3.3 Supervised linear feature extraction

In this section, you will experiment with the PRTools implementation of the Fisher mapping, `fisherm`. The Fisher mapping (often called linear discriminant analysis or LDA) can also be considered to be a variant of the Karhunen-Loève transformation.

Exercise 3.9 Load the `iris` dataset and train a Fisher mapping: `w = fisherm(a)`. Plot the mapped data using `scatterd(a*w)`.

(a) Why is the data mapped from 4 to 2 dimensions?

Exercise 3.10 Load the `nistdigs` dataset.

(a) How many dimensions can a Fisher mapping of this dataset have at most?

Change the script you created for Exercise 3.7 to use Fisher mapping instead of PCA: replace `pca` by `fisherm` and change the second `for`-loop to go up to the maximum number of dimensions instead of 10.

(b) Run the script. How many dimensions are optimal for classification?

(c) Run the script again, but select only 10% of the data as training set instead of 70% (change `[trn,tst] = gendat(a,0.7);` into `[trn,tst] = gendat(a,0.1);`). What happens?

(d) Inspect the first two dimensions of the mapped training data: `scatterd(trn*w(:,1:2))`. What do you notice? Why do you think this happens?

(e) Plot the mapped test data in a different figure: `figure; scatterd(tst*w(:,1:2))`. What do you see? Can you now explain the poor performance?

3.4 Multi-dimensionial scaling

As discussed in the lectures, multi-dimensional scaling (MDS) is a technique for nonlinear mapping. That means that the mapping cannot be expressed as a linear operation, for example in the form `new_data = data * A`. In the exercises below, you will mostly be looking at data and judging MDS mappings visually.

Multi-dimensional scaling works on distance matrices rather than datasets. A routine is available to calculate a *squared* distance matrix for you: `distm`. To actually perform multi-dimensional scaling, you need two more things:

- a stress function, and
- an optimisation technique.

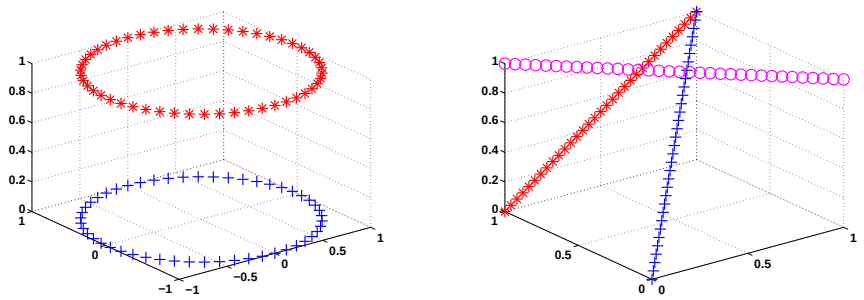
As these are quite hard to program, a routine has been prepared for you to experiment with: `mds`. In `mds`, you can supply a structure `opt` with a number of settings. The parameter `opt.q` decides which stress function to use. Recall from the lectures that a general stress function is:

$$\text{Stress} = \frac{1}{\sum_i \sum_{j>i} d_{ij}^{(q+2)}} \sum_i \sum_{j>i} d_{ij}^q \left(\delta_{ij} - d_{ij}\right)^2 \tag{3.2}$$

where d_{ij} is the distance between two objects $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ in the original space; and δ_{ij} is the distance between the mapped objects $\boldsymbol{y}_i$ and $\boldsymbol{y}_j$ in the lower-dimensional space. The parameter q controls whether short distances are emphasised ($q < 0$) or large distances ($q > 0$). In `mds`, `opt.q` can be varied between `-2` and `2`.

The other thing needed for MDS is the optimisation technique. In `mds`, this is fixed; but you can choose whether the mapped objects $\boldsymbol{y}$ are initialised using PCA (`'pca'`) or randomly (`'random'`).

Now you know all this, it is time to put MDS to the test:



(a) (b)

Figure 3.3: Two artificial datasets, generated by `circles3d` (a) and `lines5d` (b).

Exercise 3.11 Create 100 objects in a very simple 3D dataset using `circles3d` (see Figure 3.3 (a)); call it `a`. Use `scatterd` to visualise it. The goal is to find out how MDS "distorts" the two circles, compared to other methods.

(a) First, apply PCA to visualise the dataset in 2D: `scatterd(a*pca(a,2))`. What do you see?

(b) Next, apply MDS with `opt.q = 0` to visualise the dataset in 2D; use `mds(sqrt(distm(a)),2);` How does the result compare to PCA? What distortions are introduced? Remember the stress value you find.

(c) Try the same with `opt.q = -2` (stressing small distances) and `opt.q = 2` (stressing large distances). How do the results differ? Are the stress values the same? Why (not)?

(d) Apply MDS using `opt.q = 0` again, but this time using random initialisation (supply a 100 x 2 matrix of random values as starting configuration, see the `help` of `mds`). Do you find the same stress and visualisation? How does the algorithm behave?

(e) Repeat the exercise using data created by `lines5d`, which contains objects on 3 non-crossing, non-parallel lines in 5D. What is the advantage of MDS over PCA?

You should now have a bit of a feeling of what MDS can do. It can help you see structure in the data that linear methods cannot show you; but it also holds the danger of distorting structure. You should always be aware of this when applying non-linear techniques. The next exercise shows how this may happen on real-world data.

Exercise 3.12 (a) Load the dataset `nederland`, which contains a distance matrix `D` between 12 Dutch cities. Calculate an MDS mapping `w` to 2D and visualise the end result using `scatterd(D*w,'both')`. If you are confused about what city should go where when looking at the results, run the command `nlkaart` to show a (unfortunately mirrored; you correct this by View - Property Editor - Y Axis and uncheck Reverse) map of The Netherlands, or see Figure 3.4. If you get strange results with MDS, try `rotate3d`. For which `opt.q` does the result look best?

(b) Now try to visualise the cities in 3D (use `scatterd(D*w,3,'both')`). Compare the stress values and inspect the result. What do you notice when comparing it to the 2D result? Why do you think this is?

Figure 3.4: A map of The Netherlands.

(c) Try a couple of random initialisations with `opt.q = -2` and `opt.q = 2`. What do you see? How does the algorithm behave?

This last exercise shows ones of the problems of MDS, in that it works on distances only. These distances may be calculated from a real dataset, for example using `distm`, but in some problems you may *only* have a distance matrix. If MDS starts to calculate object positions which minimise the stress, it is not clear beforehand how many dimensions these objects should have. You can try to find out in how many dimensions the data can be embedded, though, by repeating experiments for different dimensions d:

Exercise 3.13 For the `nederland` dataset, pick a value for `opt.q`. Next, using standard initialisation, calculate the stress for $d = 1, 2, \ldots, 5$ dimensions. Plot the stress as a function of d. What do you see?

Exercise 3.14 Load the `nistdigs` dataset. To avoid waiting for hours for the algorithms to finish, randomly select 1000 samples out of it.

Apply PCA and MDS to map these samples to 2D. Use the `'both'` option in `scatterd` to plot the labels of the PCA-projected samples. What is the difference between PCA- and MDS-projected data?

Day 4

Clustering & hidden Markov models

Today we will first discuss the problem of clustering. This practical session is intended to familiarise you with cluster validation and the different techniques that were discussed during this morning's lecture, more specifically hierarchical clustering, the K-means algorithm, and mixtures-of-Gaussians. In the second part we will move towards modeling biological sequences using (hidden) Markov models.

4.1 Hierarchical clustering

Yesterday you learned how to select features, i.e. you learned which *variables* "contain most information". Today we will shift the emphasis to determining which *objects* belong together in groups or clusters. This clustering process enables us to extract structure from the data, and to exploit this structure for various purposes such as building a classifier or creating a taxonomy of objects.

The most difficult part of clustering is to determine whether there is *truly* any structure present in the data and if so, what this structure is. To this end, we will also employ cluster validity measures to estimate the quality of the clustering we have obtained.

In the lectures we discussed hierarchical clustering at length. There are basically two choices that need to be made in hierarchical clustering in order to construct a dendrogram:

1. the dissimilarity measure;
2. the type of linkage.

In this lab session, we will only employ the Euclidean distance between two samples as a measure of dissimilarity. As you will recall from the lecture, there are three types of linkage: complete, single and average linkage. Once the dendrogram has been constructed, we need to cut the dendrogram at an appropriate level to obtain a clustering. Simple interactive routines are available to create, visualise and cut the dendrograms.

Exercise 4.1 Start with the `hall` dataset, an artificial dataset with clear structure.

(a) Load the dataset (`load hall`). Use `scatterd` to visualise it. How many clusters are visible in the plot?

(b) What is the most suitable clustering?

An interactive clustering function is provided; it is called `interactclust`. Look at the `help` to familiarise yourself with the inputs. This function performs hierarchical clustering, draws a dendrogram in MATLAB Figure 1 and then waits for user input. The input is provided by positioning the cross-hairs in Figure 1 at the desired vertical level, corresponding to the desired number of clusters. The number of vertical stems of the dendrogram intersected by the horizontal line of the cross-hairs, corresponds to the number of clusters.

After positioning the cross-hairs and clicking on the left-mouse button, the chosen clustering is displayed in two ways. First, coloured rectangles are drawn on the dendrogram to indicate samples that are clustered together. In Figure 2, a scatterplot, colour-coded in the same fashion, is made of the samples, thus revealing the clustering (note that when there are more than two features in the dataset, the clustering is performed taking *all* features into account, but that only the first two features are displayed in Figure 2). The program remains in a loop where new clusterings can be obtained by left-mouse clicking on the dendrogram. To terminate the program, right-click anywhere on the dendrogram.

Exercise 4.2 Load dataset `rnd`. Make a scatterplot. This is a uniformly distributed dataset, with no apparent cluster structure. We will hierarchically cluster this dataset to get an idea of what a dendrogram looks like when there is no structure in the data.

(a) Perform hierarchical clustering with `interactclust` on the `rnd` dataset with complete linkage: `interactclust(+a,'c');`

(b) Cut the dendrogram at arbitrary levels to see how the samples are clustered. Look at the structure of the dendrogram. What is apparent?

(c) Repeat this for single and average linkage. Do you observe the same behavior as with complete linkage?

Exercise 4.3 (a) Perform hierarchical clustering with `interactclust` on the `hall` dataset with complete linkage: what do the lengths of the vertical stems in the dendrogram tell us about the clustering?

(b) Cut the dendrogram at different levels, i.e. inspect different numbers of clusters by left-mouse clicking at the desired level. Can you think of ways in which a good clustering can be defined?

(c) Can you devise a simple rule-of-thumb (in terms of vertical stem lengths) for finding a good clustering in the dendrogram?

Exercise 4.4 (a) Now perform single linkage hierarchical clustering: (`interactclust(+a,'s');`) on the `hall` dataset. Do you notice any significant differences with the complete linkage dendrogram?

(b) Do you notice any differences with the average linkage dendrogram (`interactclust(+a,'a')`)?

Exercise 4.5 **(a)** Load and plot the `cigars` dataset. What is an appropriate number of clusters? Why?

(b) Perform single linkage hierarchical clustering on the `cigars` dataset. Can you obtain the desired clustering with single linkage hierarchical clustering?

(c) Now perform complete linkage hierarchical clustering on the `cigars` data. Does the optimal number of clusters obtained with single linkage hierarchical clustering also produce acceptable results in complete linkage hierarchical clustering? If not, why not?

Exercise 4.6 **(a)** Perform complete linkage hierarchical clustering on the `messy` dataset. What is an appropriate number of clusters? Why?

(b) Now perform single linkage clustering on this dataset. Select the same number of clusters that you determined to be optimal with complete linkage. Is it also optimal in this case?

(c) What is an appropriate number of clusters for the single linkage case? Why?

OPTIONAL

Exercise 4.7 Perform hierarchical clustering on a dataset of your choice, for example the Golub or Khan datasets. Bear in mind that only the first two dimensions of higher dimensional datasets are displayed in the scatterplot. To visualise other dimensions, reorganise the columns in your dataset so that the appropriate variables are in columns one and two.

END OPTIONAL

4.2 K-means clustering

The operation of this algorithm was explained at length in the lecture. In this practical session, we will familiarise ourselves with the K-means algorithm by applying it in various settings. A function `kmclust` is provided to perform K-means clustering. Take a moment to familiarise yourself with the input, output and operation of this function.

Exercise 4.8 The initialisation of the K-means algorithm (positioning of the prototypes) can be done in several ways. Two simple ways are:

1. placing the prototypes uniformly distributed in the domain (approximated by the bounding box) of the data; or
2. selecting a number of the samples at random as prototypes.

For reasons of simplicity we recommend the second option.

(a) Can you think of possible advantages and disadvantages to these two initialisation approaches?

(b) Apply `kmclust` with `plotflag` and `stepflag` set on the `cigars` and `messy` datasets for different numbers of prototypes. Is K-means better suited to one of these datasets, and if so, why?

4.2.1 The local minimum problem

You might recall that the K-means algorithm is a simple iterative way of attempting to find those prototype positions that correspond to the global minimum of the total within-scatter. However, sometimes K-means ends up in a local minimum. The following exercise illustrates this.

Exercise 4.9 (a) Load dataset `triclust`. Inspect it with the aid of a scatterplot. There are clearly three equally spaced, spherical clusters.

(b) Run your K-means algorithm several times, for $g = 3$ clusters, until a solution is found where there are two prototypes in a single cluster, and the third is positioned between the remaining two clusters. Why is this a local minimum?

(c) Does hierarchical clustering also suffer from this problem?

(d) What can we do to overcome the local minimum problem?

4.3 Cluster validation

Above, you applied hierarchical clustering with different linkage types and the Euclidean dissimilarity measure as well as K-means clustering to several datasets. One aspect of clustering that we only touched briefly upon was the determination of the "optimal" number of clusters in the dataset. More specifically, given a particular hierarchical clustering, one needs to determine where to cut the dendrogram to produce a clustering of the data and one needs to know which value of g to use as input in the K-means clustering algorithm.

The following exercises are intended to familiarise you with the different cluster validity measures that were discussed in the lecture.

4.3.1 Fusion graphs

The *fusion graph* plots the *fusion level* as a function of the number of clusters (g). For example, the fusion level at $g = 2$ represents the (single, complete, average) link distance between the clusters that are merged to create two clusters from three clusters. A simple heuristic to determine the number of clusters in hierarchical clustering is to cut the dendrogram at the point where we observe a large jump in the fusion graph.

Exercise 4.10 Why is this a reasonable heuristic to employ?

The following three exercises focus on the estimation of the number of clusters based on the fusion graph.

Exercise 4.11 (a) Load the `triclust` dataset. Perform single linkage hierarchical clustering and display the fusion graph by setting the third optional argument: `interactclust(triclust,'s',1)`. Where do you observe the largest jump?

(b) Cut the dendrogram at this position, to check whether this is a reasonable number of clusters. Now perform complete linkage hierarchical clustering. Does the fusion graph give a clear indication of the number of clusters in the data? If not, why not?

Exercise 4.12 (a) Load the `hall` dataset. Perform single linkage hierarchical clustering and display the fusion graph. What do you observe in the fusion graph?

Exercise 4.13 (a) Finally, load the `messy` dataset. Perform single linkage hierarchical clustering. According to the fusion graph, where should the dendrogram be cut?

(b) Does a satisfactory clustering result from cutting the dendrogram at this point? Motivate.

(c) Now perform complete linkage clustering. Is the clustering suggested by the fusion graph better or worse than the clustering obtained with single linkage clustering?

4.3.2 The within-scatter criterion

Exercise 4.14 (a) Now, generate a two-cluster, 2D dataset: `a = +gendats([50,50],2,d)`, for `d = 10`. Make a scatterplot of the data.

(b) Recall that the function `kmclust` outputs the within-scatter associated with the obtained clustering. For the generated two-cluster dataset, compute the cluster within-scatter for $g = [1, 2, 3, 5, 10, 20, 40, 50, 100]$, and compute the *normalised* within-scatter values as a function of g (normalise by dividing all within-scatter values by the within-scatter for $g = 1$). Repeat this 10 times, and employ the `errorbar` function to plot the average and standard deviation of the cluster within-scatter as a function of g.

(c) What are the most prominent characteristics of this curve?

(d) Now for `d = 5`, generate a new dataset. For this dataset, generate the same normalised within-scatter plot as in the previous exercise, i.e. the within-scatter as a function of g. What is the biggest difference between the datasets? What is the biggest difference between the within-scatter curves?

OPTIONAL

Exercise 4.15 Generate a dataset consisting of a single Gaussian, i.e. use `d = 0`. Plot the same within-scatter curve. How does this curve differ from the other curves?

END OPTIONAL

4.3.3 The Davies-Bouldin index

D.L. Davies and D.W. Bouldin[1] introduced a cluster separation measure which is based on both the within-scatter of the clusters in a given clustering and the separation between the clusters. Formally, this measure is known as the Davies-Bouldin index (DBI). It assumes that clusters are spherical, and that a desirable clustering consists of compact clusters that are well-separated.

Suppose we wish to compute the DBI for a clustering consisting of n objects assigned to g clusters. We can compute a score for every possible pair of clusters in this clustering, which

[1] IEEE Transactions on Pattern Analysis and Machine Intelligence 1, pp. 224–227, 1979.

is inversely proportional to the distance between the cluster means and directly proportional to the sum of the within-scatters in the pair of clusters. This score is given by

$$R_{jk} = \frac{\sigma_j + \sigma_k}{\|\boldsymbol{\mu}_j - \boldsymbol{\mu}_k\|}, \quad j,k = 1,2,\ldots,g; \quad k \neq j. \tag{4.1}$$

Here $\boldsymbol{\mu}_j$ is the mean of all the objects in cluster j and σ_j is the within scatter of cluster j, given by:

$$\sigma_j = \sqrt{\frac{1}{n_j} \sum_{\boldsymbol{x}_i \in C_j} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|^2}, \tag{4.2}$$

where C_j is the set of objects associated with cluster j and n_j is the number of objects in cluster j. The score, R_{jk}, is small when the means of clusters j and k are far apart and the sum of the within-scatter for these clusters is small. Since cluster j can be paired with $g-1$ other clusters, resulting in $g-1$ pair-scores, $R_{jk}, j = 1,2,\ldots,g; k \neq j$, a *conservative* estimate of the cluster score for cluster j, when paired with all other clusters, is obtained by assigning the maximal pair-score with cluster j:

$$R_j = \max_{k=1,2,\ldots,g;\ k\neq j} R_{jk}. \tag{4.3}$$

The Davies-Bouldin index of the complete clustering is then determined by averaging these maximal pair-scores for all clusters:

$$I_{DB} = \frac{1}{g} \sum_{j=1}^{g} R_j. \tag{4.4}$$

Exercise 4.16 We will employ the Davies-Bouldin index to evaluate the clustering produced by hierarchical clustering. We will do so for a range of clusters in order to determine the optimal number of clusters, i.e. the best level to cut the dendrogram at. To achieve this we employ the function `hdb()`.

(a) The function `hdb()` has as inputs a dataset, the linkage type, the distance measure and the maximum number of clusters for which you wish to evaluate the quality of the clustering. It computes, for each clustering, the DBI. Familiarize yourself with the operation of this function.

(b) Load the `triclust` dataset and make a scatterplot. What do you expect the DBI curve as a function of the number of clusters to look like?

(c) Now apply `hdb` to this dataset with Euclidean distance, complete linkage clustering, starting at 2 clusters and stopping at 10. What is evident from the DBI curve?

(d) Apply `hdb` to the `hall` dataset with Euclidean distance, complete linkage clustering, starting at 2 clusters and stopping at 20. What do you observe in the obtained curve? Can you explain your observation?

(e) Finally, apply `hdb` to the `cigars` dataset with Euclidean distance, complete linkage clustering, starting at 2 clusters and stopping at 20. Inspect the DBI curve. Do you detect the expected number of clusters? Now try single linkage clustering. Does this help? Why (not)?

4.4 Clustering with a mixture-of-Gaussians

During the lectures, the concept of clustering based on the quality of a mixture-of-Gaussian density fit to the data was discussed. The operation of the Expectation-Maximisation (EM) algorithm, which is employed to estimate the parameters of the mixture model, was also discussed in detail. In the following set of exercises, mixture model clustering will be explored with the aid of the function `em`.

Exercise 4.17 (a) Load the `triclust` dataset and play around with the function `em` (`em(data,nmodels,type,0,1,0);`). Vary the number of Gaussians employed (`nmodels`) in the mixture model, and also vary the `type` of Gaussian employed. Relate the `type` (`'circular'`,`'aligned'`,`'gauss'`) of the Gaussian to its covariance matrix.

(b) On the `cigars` dataset, fit an unconstrained Gaussian (`type = 'gauss'`) mixture model using the function `em`. For the number of clusters g, assume the following values: $g = 1, 2, 3, 4, 5$. Which g do you expect to be the best?

(c) Repeat this 20 times (without plotting, i.e. setting the `plot_mix` and `plot_resp` arguments to zero) and plot the average log-likelihood (first output argument of `em`) and its standard deviation as a function of g (hint: use the `errorbar` function). What characteristic of the log-likelihood function indicates the preferable number of models (clusters)?

(d) Do the same for the `rnd` dataset (which has no structure) to confirm the relationship between the log-likelihood curve and the number of clusters. What is the behavior of the curve for the random dataset?

(e) Now try clustering the `messy` dataset. What is the best shape to employ for the Gaussians? What is the optimal number of clusters?

4.5 Hidden Markov models

In the last part of today's lab session you will look into various probabilistic models for biological sequences presented in the lecture: weight matrices, Markov models, and hidden Markov models.

Exercise 4.18 Assume that we have made three different models for a signal of length five:

1. all positions are identically and independently distributed (iid),
2. a weight matrix,
3. a first-order Markov chain.

(a) For each of the models and each of the sequences `CCGAT` and `CATAT` find the probability of the sequence given the model. The three models are:

- IID: $P(A) = 0.2, P(C) = 0.1, P(G) = 0.1, P(T) = 0.6$

- Weight matrix:

$$W = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0.2 & 0.3 & 0.2 & 0.1 & 0.1 \\ 0.1 & 0.2 & 0.15 & 0.6 & 0.6 \\ 0.3 & 0.4 & 0.6 & 0.1 & 0.15 \\ 0.4 & 0.1 & 0.05 & 0.2 & 0.15 \end{pmatrix} \end{array}$$

- First-order Markov chain with initial distribution $P(A) = 0.2, P(C) = 0.1, P(G) = 0.1, P(T) = 0.6$ and transition matrix

$$A = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} 0.1 & 0.8 & 0.05 & 0.05 \\ 0.35 & 0.1 & 0.1 & 0.45 \\ 0.3 & 0.2 & 0.2 & 0.3 \\ 0.6 & 0.1 & 0.02 & 0.05 \end{pmatrix} \end{array}$$

Exercise 4.19 This exercise illustrates how you can use Markov chains for deciding whether a stretch of sequence comes from a CpG island or not. CpG denotes neighbouring nucleotides `C` and `G` along one DNA strand. Due to biochemical reasons CpG is relatively rare in most DNA sequences. However, in parts of the genome, such as promoters or start regions of many genes, CpG dinucleotides are more frequent. These regions are called CpG islands. The ability to identify CpG islands along a chromosome will therefore help us find regions of interest.

An oracle gave us two matrices describing the probability of dinucleotides in CpG islands ($M+$) and in non-CpG islands ($M-$), respectively.

$$M+ = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} 0.180 & 0.274 & 0.426 & 0.120 \\ 0.171 & 0.368 & 0.274 & 0.187 \\ 0.161 & 0.339 & 0.375 & 0.125 \\ 0.079 & 0.355 & 0.384 & 0.182 \end{pmatrix} \end{array} \quad M- = \begin{array}{c} \\ A \\ C \\ G \\ T \end{array} \begin{array}{c} \begin{array}{cccc} A & C & G & T \end{array} \\ \begin{pmatrix} 0.300 & 0.205 & 0.285 & 0.210 \\ 0.322 & 0.298 & 0.078 & 0.302 \\ 0.248 & 0.246 & 0.298 & 0.208 \\ 0.177 & 0.239 & 0.292 & 0.292 \end{pmatrix} \end{array}$$

where, for example, the first row contains the probability with which an `A` is followed by each of the four bases. Note that the transition probability of `C` to `G` in CpG islands ($M+$) is relatively high compared to non-CpG islands ($M-$), as you would expect.

(a) What criterion does any transition matrix have to satisfy?

(b) Draw the Markov chain corresponding to transition matrix M+.

(c) Start the script `cpgislands`. The script starts by generating 50 sequences of random length (but less than 1000 bases long) from each of the two Markov chains. From these sequences we can estimate the transition probabilities of the chain (see lecture). Compare the estimated transition matrices with the ones given by the oracle. How do you explain the differences?

(d) From a noisy version of the transition matrices given above, we generate again 100 (50 CpG/50 non-CpG) sequences of random length (but less than 1000 bases long). You can use the log-odds ratio matrix (see lecture) to calculate a score for each sequence

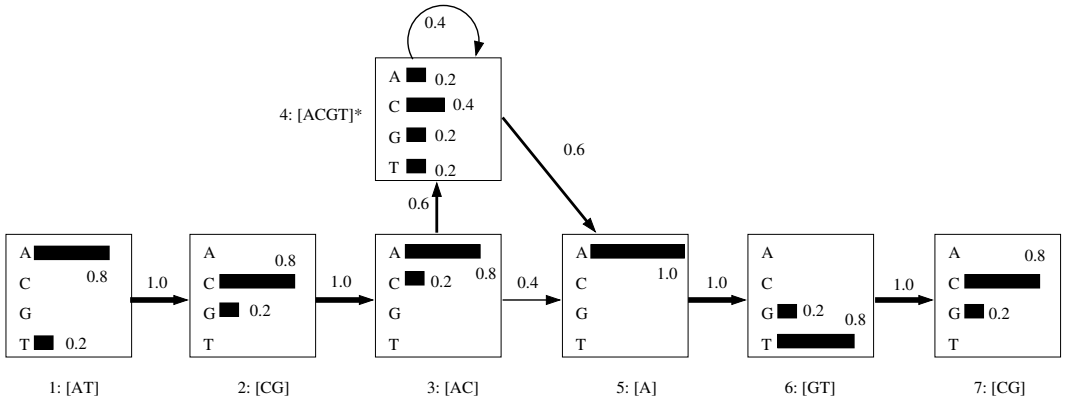


Figure 4.1: Hidden Markov model derived from the alignment discussed in the lecture. A box is called a state. The state number and the corresponding terms of the regular expression are given below the states. Transitions are shown with arrows the thickness of which indicates their probability. In each state the histogram shows the probabilities of the four nucleotides.

and decide whether a sequence is a CpG island or not. Look at the log-odds ratio for `C` followed by `G`. Does this correspond to what you would expect? Explain. Now plot the log-odds scores for the 50 CpG and the 50 non-CpG islands. The score discriminates reasonably well between the two types of sequences.

(e) A more likely problem is that you have a long unannotated sequence and want to find CpG islands in it. We can use our Markov chains for this purpose, by calculating the log-odds scores using a sliding window. CpG islands should then stand out as positive scores. Executing the last part of the script `cpgislands` does exactly this for an almost 100000 bases long sequence with a sliding window of 50 nucleotides. You might want to visualize the result:

```
load 'seq_a.results'
plot(seq_a(:,1),seq_a(:,2))
```

Can you find two stretches of positive log-odds scores of length at least 100?

(f) What is a disadvantage of using Markov chains to detect CpG islands in unannotated sequences? Would a hidden Markov model be better suited for this problem? Why or why not?

In a Markov chain each state corresponds to just one symbol, as in the CpG island finder above. Hidden Markov models can be interpreted as a combination of weight matrices and Markov chains in one model by making the states generate symbols according to a certain probability distribution. Such models turn out to be a convenient tool for modeling uncertainty in (biological) sequences. A well-known example in the field of bioinformatics are so-called *profile* HMMs. A profile HMM can be used to model a "family" of sequences from a given reliable multiple alignment. Using the forward algorithm explained in the lecture, one can search a set of sequences for other likely members of the family. The other principal use of a profile HMM is to align a new sequence to it. This can be done by finding the most probable state sequence in the profile HMM using the Viterbi algorithm.

Exercise 4.20 What is the optimal state sequence for the consensus sequence `ACACATC` in

the profile HMM of Figure 4.1? Once you have done this manually, you can verify your result by running the script `profile_vit`.

Day 5

Selected topics

5.1 Artificial neural networks

In the lecture, the multilayer perceptron was discussed in depth, including the design choices one has to make when applying them. PRTools offers a number of wrapper functions around the Matlab neural network toolbox, which do something similar. Weights are initialised to "sensible" values, the learning rate is adapted automatically (increased if training goes well, decreased if it goes wrong) and training is stopped automatically as well. See the `help` of `bpxnc` or `lmnc` for more information. PRTools even has a function which automates almost everything, `neurc`.

Exercise 5.1 In this exercise, you will investigate the influence of the number of hidden layers and the number of units they contain. It is best to optimise these using cross-validation, but this is very time-consuming. Here, you will manually change the settings. For the same reason of speed, you will use the `lmnc` ANN routine here, which optimises using the Levenberg-Marquardt algorithm, rather than backpropagation (`bpxnc`). Enter the following code in a script:

```
prprogress on; w = []; u = 1;
rand('state',99); randn('state',99); a = gendatb(100);
for i = 1:20
   w = lmnc(a,u,5,w);
   figure(1); clf; scatterd(a); plotc(w); drawnow;
end;
```

The variable `u` contains the number of hidden units.

(a) Run the script; stop it by pressing Ctrl-C when the output does not change much anymore. What do you see?

(b) Change `u` to 2, 3, 5 and 10 respectively and run the script again. What do you notice now?

(c) Now change `u` to `[10 10]`, i.e. 2 hidden layers of 10 units each. What do you see?

(d) How many hidden layers would you use, containing how many hidden units?

Exercise 5.2 (a) Use the data you generated in the previous exercise and train a radial basis function network (`rbnc`) with 5, 10, 20 and 50 Gaussians. Plot the data and classifiers. What do you see?

OPTIONAL

Exercise 5.3 (a) Load the Golub or Khan dataset, split into a training and test set, select 5 good features and train a multilayer perceptron. What architecture performs best?

(b) Compare performance to a number of standard classifiers (`nmc`, `ldc`, `qdc`, `knnc`).

(c) Repeat this for 10 and 20 features. Do the conclusions change?

END OPTIONAL

5.2 Support vector classifiers

In PRTools, the support vector classifier is implemented in the function `svc`. Several different kernels are provided as well; type `help svc` to find out which. The final parameter, C, will be kept at 1 for these exercises.

Exercise 5.4 (a) Make a linearly separable dataset using `a=gendats([20 20],2,5)`. Train a linear support vector classifier `[w,I] = svc(a,'p',1)`. Scatter the data and plot the classifier.

(b) Looking at the data, which objects are the support vectors? Check with the index vector `I`, which contains the indices of the support vectors.

Exercise 5.5 (a) Make a slightly more challenging dataset `a=gendatb([30 30])`. Train a linear support vector classifier and plot it. Does the model fit well?

(b) Use polynomial kernels of higher degrees and plot them. Which one looks acceptable?

(c) Also try the radial basis kernel for a few values of the σ. For which values of σ do you get acceptable results?

OPTIONAL

Exercise 5.6 Support vector classifiers have been used extensively in integrative bioinformatics, for example to predict protein interactions. The input consists of a number of standard features on protein pairs (e.g. experimental data such as TAP, immunoprecipitation, interologs, mRNA expression correlation etc. However, it is also possible to calculate specific kernels (such as sequence similarity kernels) and sum the kernel matrices to combine the features.

Here we use a dataset of 46 features[1] of protein pairs, as well as three sequence kernel matrices (spectrum, eMotif, PFAM) calculated on proteins, which still have to be converted to protein *pair* kernels using the pairwise kernel function[2]

(a) Load the datasets: `load vanberlo.mat` loads the feature data, `load hulsman.mat` loads the kernel data. Note that the full dataset is given in `trainsets`, split into 7 parts used for cross-validation. However, using the full dataset would take too much time, so a small selected training set `a` and a small selected test set `b` is given as well. Inspect the features: calculate mean, standard deviation and make some scatterplots. What do you notice?

(b) Train a standard support vector classifier on the feature data. We use a non-PRTools version of the classifier, which allows precomputed kernel matrices as input:

```
>> wk = proxm(a,'homogeneous');
>> Ka = a*wk;
>> Kb = b*wk;
>> w = svc_kernel(Ka,1);
```

Calculate the error of this classifier on the training set and test set. Also plot the ROC. What do you see? What part of the ROC do you think will be of interest if you would use this classifier to guide wet lab experiments?

(c) Calculate the pairwise kernel on the first protein sequence kernel matrix:

```
>> wk = pairwise_km(a,Kp{1},'n');
>> Kpwa = a*wk;
>> Kpwb = b*wk;
>> w = svc_kernel(Kpwa);
```

Investigate the performance of this classifier on the training set and test set as well.

(d) Now combine the kernels:

```
>> Kca = combine_kernels({Ka,Kpwa},[0.5 0.5]);
>> Kcb = combine_kernels({Kb,Kpwb},[0.5 0.5]);
>> w = svc_kernel(Kca);
```

Experiment with the combination weights – the last parameter of `combine_kernel` – and see how this influences results. Also try to add the other two protein kernels, `Kp{2}` and `Kp{3}`, after converting them to pair kernels using the pairwise kernel function. Can you improve performance?

Exercise 5.7 Perform exercise 5.3, but use a support vector classifier and optimise the kernel. Try to play with the parameter C – what effect does it have?

END OPTIONAL

[1] van Berlo RJP, Wessels LFA, de Ridder D and Reinders MJT. Protein complex prediction using an integrative bioinformatics approach. *Journal of Bioinformatics and Computational Biology* 5(4):839-864, 2007.

[2] Hulsman M, Reinders MJT and de Ridder D. Evolutionary optimization of kernel weights improves protein complex co-membership prediction. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 6(3):427-437, 2009.

5.3 Classifier combination

Classifiers to be combined may be trained in the same feature space, or in different feature spaces. If `w` is a classifier, the output of `a*w*classc` can be interpreted as estimates for the posterior probabilities of the objects in `a`. Different classifiers produce different posterior probabilities. This illustrated by the following exercise.

Exercise 5.8 (a) Generate a dataset of 50 points per class by `gendatb`. Train two linear classifiers: `w1`, e.g. by `nmc`, and `w2`, e.g. by `fisherc`. Determine the posterior probabilities by `p1 = a*w1*classc` and `p2 = a*w2*classc`. Combine them into one dataset `p = [p1 p2]` which has four features (why?).

(b) Make a scatter plot of the features 1 and 3. Study this plot. Next, make an ROC plot with the combined classifier (`help roc`). The original classifiers correspond to horizontal and vertical lines at 0.5. There may be other straight lines, combining the two classifiers, that perform better.

PRTools offers three ways of combining classifiers, called sequential, parallel and stacked.

- In sequential combining, classifiers operate directly on the outputs of other classifiers, e.g. `w = w1*w2`. So the features of `w2` are the outputs of `w1`.
- In stacked combining, typically classifiers computed for the same feature space are combined. They are constructed by `w = [w1, w2, w3]`. If applied as `a*w`, the result is `p = [a*w1 a*w2 a*w3]`.
- In parallel combining, typically classifiers computed for different feature spaces are combined. They are constructed as `w = [w1; w2; w3]`. If applied as `a*w` then `a` should be the combined dataset `a = [a1 a2 a3]`, in which `a1`, `a2` and `a3` are datasets defined for the feature spaces in which `w1`, `w2`, respectively `w3` are found. As a result, `p = a*w` is equivalent with `p = [a1*w1 a2*w2 a3*w3]`.

Parallel and stacked combining may be followed by a combining rule. The above constructed datasets of posterior probabilities `p` contain multiple columns (features) for each of the classes. A combining rule reduces this to a single set of posterior probabilities, one for each class, by combining all columns referring to the same class. PRTools offers the following fixed rules for combining:

- `maxc` maximum selection
- `minc` minimum selection
- `medianc` median selection
- `meanc` mean combiner
- `prodc` product combiner
- `votec` voting combiner

If the base classifiers (`w1`, `w2`, ...) do not produce posterior probabilities, but for instance distances, then these combining rules operate similarly. This is illustrated by the following exercises:

OPTIONAL

Exercise 5.9 Generate a small dataset, e.g. `a = gendatb`. Train three classifiers, e.q. `w1 = nmc(a)*classc, w2 = fisherc(a)*classc, w3 = qdc(a)*classc`.

Create a combining classifier `v = [w1, w2, w3]*meanc`.

Generate a testset `b` and compare the performances of `w1`, `w2`, `w3` individually with that of `v`.

(a) Inspect the architecture of the combined classifier by `parsc(v)`.

END OPTIONAL

Exercise 5.10 Load three of the `mfeat` datasets and generate training and test sets, e.g.:

```
>> a = mfeat_kar; [b1,c1] = gendat(a,0.25)
>> a = mfeat_zer; [b2,c2] = gendat(a,0.25)
>> a = mfeat_mor; [b3,c3] = gendat(a,0.25)
```

Note the differences in feature sizes of these sets. Train three nearest mean classifiers

```
>> w1 = nmc(b1)*classc; w2 = nmc(b2)*classc; w3 = nmc(b3)*classc;
```

and compute the combined classifier

```
>> v = [w1; w2; w3]*meanc
```

Compare the performance of the combining classifier with the three individual classifiers:

```
>> [c1 c2 c3]*v*testc
>> c1*w1*testc, c2*w2*testc, c3*w3*testc
```

Instead of using fixed combining rules like `maxc` and `meanc`, it is also possible to use a trained combiner. In this case the outputs of the base classifier are used to train a combining classifier like `nmc` or `fisherc`. This demands the following operations:

```
>> a = gendatb(50)
>> w1 = nmc(a)*classc, w2 = fisherc(a)*classc, w3 = qdc(a)*classc
>> a_out = [a*w1 a*w2 a*w3]
>> v1 = [w1 w2 w3]*fisherc(a_out)
```

PRTools offers the possibility to define untrained combining classifiers:

```
>> v = [nmc*classc fisherc*classc qdc*classc]*fisherc
```

Such a classifier can simply be trained by `v2 = a*v`.

OPTIONAL

Exercise 5.11 Stacked combining.

(a) Load the `mfeat_zer` dataset. Split it into a training and a test set of equal size. Train the following classifiers: `nmc`, `ldc`, `qdc`, `knnc([],3)`, `treec`.

Determine the performance of each of these base classifiers.

(b) Try to find a combining classifier that performance better than the best base classifier.

Exercise 5.12 Parallel combining.

(a) Load all `mfeat` datasets. Split the data into training and test sets of equal size. Make sure that these sets relate to the same objects, e.g. by resetting the random seed each time by `rand('seed',1)` before calling `gendat`.

Compute for each dataset the nearest mean classifier and estimate their performances.

(b) Try to find a combining classifier that performance better than the best base classifier.

Exercise 5.13 Bootstrapping and averaging.

The routine `baggingc` computes a set of classifiers on a single training set by bootstrapping and averaging all coefficients. Compare the performance of a simple classifier like `nmc` with its bagged version for a 2-dimensional dataset of 20 objects generated by `gendatd`. Use a a test set of 200 objects. Study the performance for bagging sets of sizes between 10 and 200.

Exercise 5.14 Bootstrapping and aggregating.

The routine `baggingc` can also be used to combine a set of classifiers based on bootstrapping. using the posterior probability estimates. Combining rules like `voting`, `min`, `max`, `mean`, and product can be used. Compare the performance of a simple classifier like `nmc` with its bagged version for a datasets generated by `gendatd`. Study the scatter and classifier plots.

Exercise 5.15 Decision stumps.

A decision stump is a simplified decision tree, trained to a small depth, usually just for a single split. The command `stumpc` constructs a decision tree classifier until a specified depth. Generate objects according to the banana dataset (`gendatb`), make a scatterplot and plot the decision stump classifiers for the depth levels 1, 2 and 3. Estimate the classification errors using an independent test set and compare the plots and the resulting error with a full size decision tree (`treec`).

END OPTIONAL

Exercise 5.16 Weak classifiers.

A family of weak classifiers is available through the command `w = weakc(a,alf,iter,r)` in which `alf (0 < alf < 1)` determines the size of a randomly selected subset of the training set `a` to train a classifier determined by `r`, with `r = 0: nmc`; `r = 1: fisherc`; `r = 2: udc` and `r = 3: qdc`. In total, `iter` classifiers are trained and the best one according to the total set `a` is selected and

returned in `w`. Define a set of linear classifiers (`r = 0,1`) for increasing `iter`, and include the strong version of the classifier:

```
>> v1 = weakc([],0.5,1,0); v1 = setname(v1,'weak0-1');
>> v2 = weakc([],0.5,3,0); v2 = setname(v2,'weak0-3');
>> v3 = weakc([],0.5,20,0); v3 = setname(v3,'weak0-20');
>> w = {nmc,v1,v2,v3};
```

Generate some datasets, e.g. by `a=gendath` and `a=gendatb`. Train and plot these classifiers by `v = a*w` and `plotc(v)` in the scatterplot (`scatterd(a)`).

OPTIONAL

Exercise 5.17 (a) Using `cleval`, compute and plot learning curves for the Highleyman data averaged over 5 repetitions of crossvalidation for the above defined set of classifiers.

(b) Compute and plot learning curves for the circular classes (`gendatc`) averaged over 5 repetitions of crossvalidation for a set of quadratic weak classifiers.

Exercise 5.18 Adaboost.

The Adaboost classifier `[w,v] = adaboostc(a,base-classf,n,comb-rule)` uses the untrained (weak) classifier `base-classf` for generating `n` base classifiers by the training set `a`, iteratively updating the weights for the objects in `a`. These weights are used as object prior probabilities for generating subsets of `a` for training. The entire set of base classifiers is returned in `v`. They are combined by `base-classf` into a single classifier `w`. Default is the standard weighted voting combiner.

Study the Adaboost classifier for two datasets: `gendatb` and `gendatc`. Use as base classifier `stumpc` (decision stump), `weakc([],[],1,1)` and `weakc([],[],1,2)`.

Plot the final classifier in the scatterplot by `plotc(w,'r',3)`. Plot also the unweighted voting combiner by `plotc(v*votec,'g',3)` and the trained Fisher combiner by `plotc(a*(v*fisherc),'b',3)`. It might be needed to improve the quality of the plotted classifiers by giving `gridsize(300)`, before `plotc` is executed.

Exercise 5.19 Compute the Adaboost error curve for the `sonar` dataset for some numbers of boosting steps, e.g. 5 and 100 (advanced users may try to write a script that plots an entire error curve). Use `stumpc` as a base-classifier and weighted voting for combining. Try to improve the result by using other base classifiers and other combiners.

END OPTIONAL

5.4 Complexity

5.4.1 Learning curves

Exercise 5.20 (a) Generate a large training set from the Highleyman distribution: `a = gendatb([200 200])`. Generate a learning curve using `cleval` for several classifiers,

like the `nmc`, `ldc`, `qdc` and `parzenc` over a large range of training set sizes, `TR=[2 3 5 10 25 50 100]`.

(b) Which classifier has the best performance for 2 training objects per class? And which for 100 per class?

(c) Which classifier would benefit most from more training samples?

(d) Which classifier is the most simple, which is the most complex?

OPTIONAL

Exercise 5.21 **(a)** Create learning curves for a number of PRTools classifiers on the Golub and Khan datasets (select a small number of interesting features first). What do you notice?

(b) Use the test set as a "tuning" set in the call to `cleval`. What effect does this have?

5.4.2 Bias-variance trade-off

The bias-variance trade-off can be observed in *all* methods which fit a model to some data. When a model is fitted to some data, there will always be some fitting errors. It is always possible to distinguish two types of fitting errors. To illustrate this, the mean-squared-error in regression problems will be used, because here it can be demonstrated best.

The mean-squared error between a function $f(\boldsymbol{x}_i; \boldsymbol{\beta})$ and some desired output y_i for input object $\boldsymbol{x}_i$ is defined as:

$$e_{MSE}(f(\boldsymbol{x}_i; \boldsymbol{\beta}), y_i) = (f(\boldsymbol{x}_i; \boldsymbol{\beta}) - y_i)^2 \tag{5.1}$$

This error can be rewritten, and there two contributions to the error appear: bias and variance.

Bias : the first contribution is due to the basic mismatch between the model $f(\boldsymbol{x}_i; \boldsymbol{\beta})$ and the data y_i.

Variance : the second contribution is due to the difficulty in estimating the optimal weights $\boldsymbol{\beta}$ from a limited set of data.

The derivation is very simple, you can only be confused by the notation. To simplify the notation, assume that the outputs in the training set y_i are noise-free.[3] Then the expected mean squared error $E_{\mathcal{X}^{tr}}[e_{MSE}(f, \boldsymbol{\beta}, \mathcal{X}^{tr})]$ can be computed over all training sets $\mathcal{X}^{tr}$ and we

[3] If that is not the case, we have to decompose the observed outputs into a model and a noise contribution: $y_i = E[y|\boldsymbol{x}_i] + \epsilon$. The first term $E[y|\boldsymbol{x}_i]$ represents the deterministic part of the data which we try to model. The second term ϵ contains all stochastic contributions and is often assumed to have zero mean. The bias-variance decomposition considers the difference between what a function f can represent and the deterministic part of the data $E[y|\boldsymbol{x}]$. When y_i contains some stochastic elements, replace y_i by $E[y|\boldsymbol{x}_i]$ in the coming decomposition.

can expand to:

$$
\begin{aligned}
&E_{\mathcal{X}^{tr}}[e_{MSE}(f, \boldsymbol{\beta}, \mathcal{X}^{tr})] = E_{\mathcal{X}^{tr}}\left[\frac{1}{n}\sum_i \left(f(\boldsymbol{x}_i;\boldsymbol{\beta}) - y_i\right)^2\right] \\
&= E_{\mathcal{X}^{tr}}\left[\frac{1}{n}\sum_i \left(f(\boldsymbol{x}_i;\boldsymbol{\beta}) - E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})] + E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})] - y_i\right)^2\right] \\
&= E_{\mathcal{X}^{tr}}\left[\frac{1}{n}\sum_i \left(f(\boldsymbol{x}_i;\boldsymbol{\beta}) - E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})]\right)^2\right] + E_{\mathcal{X}^{tr}}\left[\frac{1}{n}\sum_i \left(E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})] - y_i\right)^2\right] \\
&\quad + E_{\mathcal{X}^{tr}}\left[\frac{1}{n}\sum_i 2\left(f(\boldsymbol{x}_i;\boldsymbol{\beta}) - E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})]\right)\left(E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})] - y_i\right)\right] \qquad (5.2)
\end{aligned}
$$

Here $E_{\mathcal{X}^{tr}}[f(\boldsymbol{x};\boldsymbol{\beta})]$ is the expectation of the output of the classifier $f(\boldsymbol{x};\boldsymbol{\beta})$ over all possible training sets (with the same sample size n).

The expected mean square error was computed over all possible training sets $\mathcal{X}^{tr}$ to investigate the average behavior. If one is lucky, one might have a very good training set, which results in a very good generalization performance. On the other hand, for a very atypical training set, only poor generalization results can be expected. To investigate how well a function fits the data, these contributions have to be averaged out.

Because of the averaging over all the training sets, the second part of the last term, $(E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})] - y_i)$ in ((5.2)), can be removed from the expectation since this does not depend on $\mathcal{X}^{tr}$. Then the first part of the last term becomes zero: $E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta}) - E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})]] = 0$ and therefore the third term in ((5.2)) vanishes:

$$
\begin{aligned}
&E_{\mathcal{X}^{tr}}\left[e_{MSE}(f, \boldsymbol{\beta}, \mathcal{X}^{tr})\right] \\
&= E_{\mathcal{X}^{tr}}\left[\frac{1}{n}\sum_i \left(f(\boldsymbol{x}_i;\boldsymbol{\beta}) - E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})]\right)^2\right] + E_{\mathcal{X}^{tr}}\left[\frac{1}{n}\sum_i \left(E_{\mathcal{X}^{tr}}[f(\boldsymbol{x}_i;\boldsymbol{\beta})] - y_i\right)^2\right] \\
&= \text{variance} + (\text{bias})^2 \qquad (5.3)
\end{aligned}
$$

This means that:

Bias : can be estimated by computing the difference between the average actual output and the "truth".

Variance : can be estimated by calculating the difference between the actual output and the average output.

For other types of errors, like the log-likelihood, a bias-variance decomposition can be made as well. This indicates that the bias-variance trade-off is a fundamental problem in fitting a model to some data.

To investigate the bias-variance trade-off, the outputs of many classifiers or regressors have to be compared with the ground truth. When the ground truth is not available, it cannot find out where errors are made consistently. Furthermore, as it is necessary to train models

on several training sets, we have to be able to generate datasets over and over again. This means that for looking at the bias-variance trade-off in the exercises below, artificial data has to be used.

Exercise 5.22 In this exercise, the emphasis is on interpretation rather than programming: most of the code will be given, but do not enter it without understanding what it does.

(a) To get a feeling for the data we are working with, generate some data using `genregres` (say 25 training objects), fit a linear function and plot it. Run this a number of times. Is the line a good fit? Does it change a lot when you run it over and over?

(b) Now write a script containing the following code:

```
clear all;
n = 25; deg = 1;          % Parameters: dataset size (n), degree (deg)
gx = (0:0.05:1)';               % Grid
ftrue = genregres(gx,0);        % Generate true function
ytrue = gettargets(ftrue);      %  and extract the y-values

for i = 1:100
     x = genregres(n);         % Generate a regression dataset
     w = linearr(x,deg);       % Fit a polynomial with DEG degrees
     f(i,:) = +(gx*w)';        %   and calculate predictions
end;
f = f';
bias2    = (mean(f,2)-ytrue).^2;
variance = mean((f-repmat(mean(f,2),1,100).^2),2);
err      = sum(bias2+variance);

figure(2); clf; scatterr(ftrue,'b-'); hold on;
plot(gx,bias2,'g-'); plot(gx,variance,'r-');
legend('Function','Bias','Variance');
title(['Total error:' num2str(err)]);
```

(c) Explain how the variance is calculated; what does it measure?

(d) Explain how the squared bias is calculated; what does it measure?

(e) Explain the plot: why is the bias high on the sides and low in the middle? Why is the variance low?

(f) Put the code above in a loop which performs this for `deg = 0 ... 6`. Put the resulting code in a loop which repeats this 10 times. Store the error found for each repeat, for each degree. At the end, make an errorbar plot of the error as a function of the number of degrees. Is there a clear optimum?

END OPTIONAL

Appendix A

Introduction to Matlab

Ela Pękalska and Marjolein van der Glas

A.1 Getting started with Matlab

MATLAB is a tool for mathematical (technical) calculations. First, it can be used as a scientific calculator. Next, it allows you to plot or visualize data in many different ways, perform matrix algebra, work with polynomials or integrate functions. You can create, execute and save a sequence of commands to make your computational process automatic. Finally, MATLAB is also a user-friendly programming language, which gives the possibility to handle mathematical calculations in an easy way. In summary, as a computing/programming environment, MATLAB is especially designed to work with data sets as a whole such as vectors, matrices and images.

On most systems, after logging in, you can enter MATLAB with the system command `matlab` and exit with the command `exit` or `quit`. Running MATLAB creates one or more windows on your screen. The most important is the MATLAB *Command Window*, which is the place where you interact with MATLAB. The string `>>` (or `EDU>>` for the Student Edition) is the MATLAB prompt. When the Command window is active, a cursor appears after the prompt, indicating that MATLAB is waiting for your command.

A.1.1 Input via the command-line

MATLAB is an interactive system; commands followed by `Enter` are executed immediately. The results are, if desired, displayed on the screen. Execution of a command is only possible when the command is typed according to the rules. Table A.1 shows a list of commands used to solve indicated mathematical equations (a, b, x and y are numbers). Below you find basic information to help you starting with MATLAB:

- Commands in MATLAB are executed by pressing `Enter` or `Return`. The output will be displayed on the screen immediately. Try the following:

  ```
  >> 3 + 7.5
  >> 18/4 - (3 *  7)
  ```

  Note that spaces are **not** important in MATLAB.

- The result of the last performed computation is ascribed to the variable `ans`, which is an example of a MATLAB built-in variable. It can be used on the next command line. For instance:

```
>> 14/4
ans =
     3.5000
>> ans^(-6)
ans =
   5.4399e-04
```

  `5.4399e-04` is a computer notation of $5.4399 * 10^{-4}$. Note that `ans` is always overwritten by the last command.

- You can also define your own variables (more on variables in Section A.2.1), e.g. `a` and `b` as:

```
>> a = 14/4
a =
     3.5000
>> b = a^(-6)
b =
   5.4399e-04
```

- When the command is followed by a semicolon ';', the output is suppressed. Compare:

```
>> 3 + 7.5
>> 3 + 7.5;
```

- It is possible to execute more than one command at the same time; the commands should then be separated by commas (to display the output) or by semicolons (to suppress the output display), e.g.:

```
>> sin(pi/4), cos(pi); sin(0)
ans =
      0.7071
ans =
      0
```

  Note that the value of `cos(pi)` is not printed.

- By default, MATLAB displays only 5 digits. The command `format long` increases this number to 15, `format short` reduces it to 5 again. For instance:

```
>> 312/56
ans =
    5.5714
>> format long
>> 312/56
ans =
   5.57142857142857
```

- The output may contain some empty lines; this can be suppressed by the command `format compact`. In contrast, the command `format loose` will insert extra empty lines.

- MATLAB is case sensitive, for example, a is written as `a` in MATLAB; `A` will result then in an error.

Mathematical notation	MATLAB command
$a + b$	`a + b`
$a - b$	`a - b`
ab	`a * b`
$\frac{a}{b}$	`a / b` or `b \ a`
x^b	`x^b`
$\sqrt{x}$	`sqrt(x)` or `x^0.5`
$\|x\|$	`abs(x)`
π	`pi`
$4 \cdot 10^3$	`4e3` or `4*10^3`
i	`i` or `j`
$3 - 4i$	`3-4*i` or `3-4*j`
e, e^x	`exp(1)`, `exp(x)`
$\ln x$, $\log x$	`log(x)`, `log10(x)`
$\sin x$, $\arctan x$, ...	`sin(x)`, `atan(x)`,...

Table A.1: Translation of mathematical notation to MATLAB commands.

- All text after a percent sign `%` until the end of a line is treated as a comment. Enter e.g. the following:

  ```
  >> sin(3.14159)            % this is an approximation of sin(pi)
  ```

  You should skip comments while typing the examples.
- Previous commands can be fetched back with the $\boxed{\uparrow}$-key. The command can also be changed, the $\boxed{\leftarrow}$ and $\boxed{\rightarrow}$-keys may be used to move around in a line and edit it.

A.1.2 `help`-facilities

MATLAB provides assistance through extensive online help. The `help` command is the simplest way to get help. It displays the list of all possible topics. To get a more general introduction to `help`, try:

```
>> help help
```

If you already know the topic or command, you can ask for a more specified help. For instance:

```
>> help elfun
```

provides information on the elementary math functions of MATLAB. The topic you want help on must be exact. The `lookfor` command is useful if you do not know the exact name of the command or topic, e.g.:

```
>> lookfor inverse
```

displays a list of all commands, with a short description, for which the word `inverse` is included in its help-text. Besides the `help` and `lookfor` commands, there is also a separate mouse driven help. The `helpwin` command opens a new window on screen which can be browsed in an interactive way.

A.1.3 Interrupting a command or program

Sometimes you might spot an error in your command or program. Due to this error it can happen that the command or program does not stop. Pressing `Ctrl-C` (or `Ctrl-Break` on PC) forces MATLAB to stop the process. After this, the MATLAB prompt (`>>`) re-appears. This may take a while, though.

A.1.4 Workspace issues

If you work in the Command Window, MATLAB memorizes all commands that you entered and all variables that you created. These commands and variables reside in the MATLAB*workspace* and they might be easily recalled when needed, e.g. to recall previous commands, the ↑-key is used. Variables can be verified with the commands `who`, which gives a list of variables present, and `whos`, which includes also extra information. Assuming that you performed all commands given in Section A.1, after typing `who` you would obtain:

```
>> who
Your variables are:
a         ans       b         x
```

The command `clear <name>` deletes the variable `<name>` from the MATLAB workspace, `clear` or `clear all` removes all variables. This is useful when starting a new task. For example:

```
>> clear a x
>> who
Your variables are:
ans       b
```

A.1.5 Saving and loading data

The easiest way to save or load MATLAB variables is by using (clicking) the **File** menu-bar, and then the **Save Workspace as...** and **Load Workspace...** items. Alternatively, the MATLAB commands `save` and `load` can be used. `save` allows for saving your workspace variables either into a binary file or an ASCII file. Binary files automatically get the `'.mat'` extension. For ASCII files, it is recommended to add a `'.txt'` or `'.dat'` extension. Study the use of the `save` command:

```
>> s1 = sin(pi/4);
>> c1 = cos(pi/4); c2 = cos(pi/2);
>> str = 'hello world';       % a string
>> save data                  % saves all variables in binary format to data.mat
>> save numdata s1 c1         % saves numeric variables: s1 and c1 to numdata.mat
>> save allcos.dat c* -ascii  % saves c1,c2 in 8-digit ascii format to allcos.dat
```

The `load` command allows for loading variables into the workspace. It uses the same syntax as `save`. It might be useful to clear the workspace before each `load` and check which variables are present in the workspace (use `who`) after loading. Assuming that you have created the files above, the variables can be now loaded as:

```
>> load data                  % loads all variables from data.mat
>> load data s1 c1            % loads only specified variables from data.mat
```

It is also possible to read ASCII files that contain rows of space separated values. Such a file may contain comments that begin with a percent character. The resulting data is placed into a variable with *the same* name as the ASCII file (without the extension). Check, for example:

```
>> load allcos.dat            % loads data from allcos.dat into variable allcos
>> who                        % lists variables present in the workspace now
```

A.2 Mathematics with vectors and matrices

The basic element of MATLAB is a matrix (or an array). Special cases are:

- a 1×1-matrix: a scalar or a single number;
- a matrix existing only of one row or one column: a vector.

Note that MATLAB may behave differently depending on input, whether it is a number, a vector or a 2D matrix.

A.2.1 Assignments and variables

Formally, there is no need to declare (i.e. define the name, size and the type of) a new variable in MATLAB. A variable is simply created by an assignment (e.g. `a = 15/4`). Each newly created numerical variable is *always* of the `double` type. You can change this type by converting it into e.g. the `single` type[1].

Bear in mind that *undefined* values cannot be assigned to a variable. So, the following is not possible:

```
>> clear x;                   % to make sure that x does not exist
>> f = x^2 + 4 * sin (x)
```

It becomes possible by:

```
>> x = pi / 3;  f = x^2 + 4 * sin (x)
```

Here are some examples of different types of MATLAB variables (check their types by using `whos`):

```
>> M  = [1 2; 3 4; 5 6]       % a matrix
>> 2t = 8                     % what is the problem with this command?
>> c  = 'E'                   % a character
>> str = 'Hello world'        % a string
```

There is also a number of built-in variables, e.g. `pi`, `eps` or `i`, presented in Table A.2. In addition to creating variables by assigning values to them, another possibility is to copy one variable, e.g. `b` into another, e.g. `a`. In this way, the variable `a` is automatically created (if `a` already existed, its previous value is now lost):

```
>> b = 10.5;
>> a = b;
```

[1] A variable `a` is converted into a different type by performing e.g. `a = single(a)`, etc.

Variable name	Value/meaning
`ans`	the default variable name used for storing the last result
`pi`	$\pi = 3.14159...$
`eps`	the smallest positive number that added to 1, creates a result larger than 1
`inf`	representation for positive infinity, e.g. `1/0`
`nan` or `NaN`	representation for `n`ot-`a`-`n`umber, e.g. `0/0`
`i` or `j`	$i = j = \sqrt{-1}$
`nargin/nargout`	number of function input/output arguments used
`realmin/realmax`	the smallest/largest usable positive real number

Table A.2: Built-in variables in MATLAB.

If `min` is the name of a function (see Section A.4.2), then `a` defined as:

```
>> b = 5; c = 7;
>> a = min (b,c);
```

will call that function, with the values `b` and `c` as parameters. The result of this function (its return value) will be written (assigned) into `a`.

A.2.2 Vectors

Row vectors are lists of numbers separated either by commas or by spaces. They are examples of simple arrays. First element has index 1. The number of entries is known as the *length* of the vector (the command `length` exists as well). Their entities are referred to as *elements* or *components*. The entries must be enclosed in `[ ]`:

```
>> v = [-1 sin(3) 7]
v =
   -1.0000    0.1411    7.0000
```

A number of operations can be performed on vectors. A vector can be multiplied by a scalar, or added/subtracted to/from another vector with *the same* length, or a number can be added/subtracted to/from a vector. A particular value can be also changed or displayed. All these operations are carried out element-by-element. Vectors can be also built from the already existing ones:

```
>> v = [-1 2 7]; w = [2 3 4];
>> z = v + w                    % an element-by-element sum
z =
     1     5    11
>> t = [2*v, -w+2]
ans =
    -2     4    14     0    -1    -2
>> v(2) = -1                    % change the 2nd element of v
v =
    -1    -1     7
```

A colon notation is an important shortcut, used when producing row vectors (see Table A.3 and `help colon`):

```
>> 2:5
ans =
     2     3     4     5
```

In general, `first:step:last` produces a vector of entities with the value `first`, incrementing by the `step` until it reaches `last`:

```
>> -0.2:0.5:1.7
ans =
   -0.2000    0.3000    0.8000    1.3000
>> 5.5:-2:-2.5                  % a negative step is also possible
ans =
    5.5000    3.5000    1.5000   -0.5000   -2.5000
```

Parts of vectors can be extracted by using a colon notation:

```
>> r = [-1:2.5:6, 2, 3, -2];    % r = [-1 1.5 4 2 3 -2]
>> r(2:2:6)                     % get the elements of r from the positions 2, 4 and 6
ans =
     1.5     2     -2
```

To create *column* vectors, you should separate entries by a semicolon ';' or by new lines. The same operations as on row vectors can be done on column vectors. However, you cannot for example add a column vector to a row vector. To do that, you need an operation called *transposing*, which converts a column vector into a row vector and vice versa:

```
>> f = [-1; 3; 5]               % a column vector
f =
    -1
     3
     5
>> f'                           % f' is a row vector
ans =
    -1     3     5
>> v = [-1 2 7];                % a row vector
>> f + v                        % you cannot add a column vector f to a row vector v
??? Error using ==> +
Matrix dimensions must agree.
>> f' + v
ans =
    -2     5    12
```

You can now compute the inner product between two vectors `x` and `y`, $\mathtt{x}^T\mathtt{y} = \sum_i \mathtt{x}_i\,\mathtt{y}_i$, in a simple way:

```
>> v = [-1; 2; 7];          % v is now a column vector; f is also a column vector
>> f' * v                   % this is the inner product! f * v' is the outer product
ans =
     42
```

Command	Result
`A(i,j)`	A_{ij}
`A(:,j)`	j-th column of A
`A(i,:)`	i-th row of A
`A(k:l,m:n)`	$(l-k+1) \times (n-m+1)$ matrix with elements A_{ij} with $k \leq i \leq l$, $m \leq j \leq n$
`v(i:j)'`	'vector-part' $(v_i, v_{i+1}, \ldots, v_j)$ of vector v

Table A.3: Manipulation of (groups of) matrix elements.

A.2.3 Matrices

An $n \times k$ matrix is a rectangular array of numbers having n rows and k columns. Defining a matrix in MATLAB is similar to defining a vector. The generalization is straightforward, if you know that a matrix consists of row vectors (or column vectors). Commas or spaces are used to separate elements in a row, and semicolons are used to separate individual rows. For example, a matrix can be defined as:

```
>> A = [1 2 3; 4 5 6; 7 8 9]   % row by row input
A =
   1   2   3
   4   5   6
   7   8   9
```

Transposing a vector changes it from a row to a column or the other way around. This idea can be extended to a matrix, where transposing interchanges rows with the corresponding columns, as in the example:

```
>> A2 = [1:4; -1:2:5], A2'
A2 =
     1     2     3     4
    -1     1     3     5
ans =
     1    -1
     2     1
     3     3
     4     5
>> size(A2), size(A2')        % returns the size (dimensions) of a matrix
ans =
     2     4
ans =
     4     2
```

There are also built-in matrices of size specified by the user (see Table A.4). A few examples

Command	Result
`n = rank(A)`	n becomes the rank of matrix A
`x = det(A)`	x becomes the determinant of matrix A
`x = size(A)`	x becomes a row-vector with 2 elements: the number of rows and columns of A
`x = trace(A)`	x becomes the trace (sum of diagonal elements) of matrix A
`x = norm(v)`	x becomes the Euclidean length of vector v
`C = A + B`	sum of two matrices
`C = A - B`	subtraction of two matrices
`C = A * B`	multiplication of two matrices
`C = A .* B`	'element-by-element' multiplication (A and B are of equal size)
`C = A ^ k`	power of a matrix ($k \in Z$; can also be used for A^{-1})
`C = A .^ k`	'element-by-element' power of a matrix
`C = A'`	the transposed of a matrix; A^T
`C = A ./ B`	'element-by-element' division (A and B are of equal size)
`X = A \ B`	finds the solution in the least squares sense to the system of equations $AX = B$
`X = B / A`	finds the solution of $XA = B$, analogous to the previous command
`C = inv(A)`	C becomes the inverse of A
`C = null(A)`	C is an orthonormal basis for the null space of A obtained from the singular value decomposition
`C = orth(A)`	C is an orthonormal basis for the range of A
`C = rref(A)`	C is the reduced row echelon form of A
`L = eig(A)`	L is a vector containing the (possibly complex) eigenvalues of a square matrix A
`[X,D] = eig(A)`	produces a diagonal matrix D of eigenvalues and a full matrix X whose columns are the corresponding eigenvectors of A
`S = svd(A)`	S is a vector containing the singular values of A
`[U,S,V] = svd(A)`	S is a diagonal matrix with nonnegative diagonal elements in decreasing order; columns of U and V are the accompanying singular vectors
`x = linspace(a,b,n)`	generates a vector x of n equally spaced points between a and b
`x = logspace(a,b,n)`	generates a vector x starting at 10^a and ended at 10^b containing n values
`A = eye(n)`	A is an $n \times n$ identity matrix
`A = zeros(n,m)`	A is an $n \times m$ matrix with zeros (default $m = n$)
`A = ones(n,m )`	A is an $n \times m$ matrix with ones (default $m = n$)
`A = diag(v)`	gives a diagonal matrix with the elements $v_1, v_2, \ldots, v_n$ on the diagonal
`X = tril(A)`	X is lower triangular part of A
`X = triu(A)`	X is upper triangular part of A
`A = rand(n,m)`	A is an $n \times m$ matrix with elements uniformly distributed between 0 and 1
`A = randn(n,m)`	ditto - with elements standard normal distributed
`v = max(A)`	v is a vector with the maximum value of the elements in each column of A or v is the maximum of all elements if A is a vector
`v = min(A)`	ditto - with minimum
`v = sum(A)`	ditto - with sum

Table A.4: Frequently used matrix operations and functions.

are given below:

```
>> I = eye(2)                  % the 2-by-2 identity matrix
I =
     1     0
     0     1
>> B = randn(1,2)              % a vector of normally distributed random numbers
B =
   -0.4326   -1.6656
>> r = [1 3 -2]; R = diag(r)   % create a diagonal matrix with r on the diagonal
R =
     1     0      0
     0     3      0
     0     0     -2
```

It is often needed to build a larger matrix from the smaller ones:

```
>> x = [4; -1]; y = [-1 3]; X = [x y']
X =                            % X consists of the columns: x and y'
     4    -1
    -1     3
>> T = [-1 3 4; 4 5 6]; t = 1:3;
>> A = [[T; t] ones(3,2)]]        % add a row vector t, then add two columns of 1's
A =
    -1     3     4     1     1
     4     5     6     1     1
     1     2     3     1     1
```

It is possible to manipulate (groups of) matrix elements (see Table A.3). A part can be extracted from a matrix in a similar way as from a vector. Each element in the matrix is indexed by a row and a column to which it belongs, e.g. `A(i,j)` denotes the element from the i-th row and the j-th column of the matrix A.

```
>> A(3,4)
ans =
     1
>> A(4,3)                      % A is a 3-by-5 matrix! A(4,3) does not exist
???  Index exceeds matrix dimensions.
```

It is easy to extend a matrix automatically. For the matrix A it can be done e.g. as follows:

```
>> A(4,3) = -2                 % assign -2 to the position (4,3); the uninitialized
A =                            % elements become zeros
    -1     3     4     1     1
     4     5     6     1     1
     1     2     3     1     1
     0     0    -2     0     0
>> A(4,[1:2,4:5]) = 4:7;       % assign A(4,1)=4, A(4,2)=5, A(4,4)=6 and A(4,5)=7
```

Different parts of the matrix A can be now extracted:

```
>> A(:,2)                       % extract the 2nd column of A; what will you get?
>> A(4,:)                       % extract the 4th row of A
ans =
     4     5    -2     6     7
>> A(2:3,[1,5])
     4    -1
     1     1
```

Table A.4 shows some frequently used matrix operations and functions. A few examples are given below:

```
>> B = [1 4 -3; -7 8 4]; C = [1 2; 5 1; 5 6];
>> (B + C')./4                  % add matrices and divide all elements of result by 4
ans =
    0.5000    2.2500    0.5000
   -1.2500    2.2500    2.5000
>> D = [1 -1 4; 7 0 -1];
>> D = B * D'; D^3              % D^3 is equivalent to D * D * D
ans =
    -4205       38390
     3839     -150087
>> D.^3 - 2                     % for all elements: raise to the power 3, subtract 2
ans =
    -3377         998
       -1     -148879
```

The concept of an empty matrix `[]` is also very useful in MATLAB. For instance, a few columns or rows can be removed from a matrix by assigning an empty matrix to it. Try for example:

```
>> C = [1 2 3 4; 5 6 7 8; 1 1 1 1];
>> D = C; D(:,2) = []           % now a copy of C is in D; remove the 2nd column of D
>> C ([1,3],:) = []             % remove the 1st and 3rd rows from C
```

A.3 Control flow

A control flow structure is a block of commands that allows conditional code execution and making loops.

A.3.1 Logical and relational operators

To use control flow commands, it is necessary to perform operations that result in logical values: TRUE or FALSE, which in MATLAB correspond to `1` or `0` respectively (see Table A.5 and `help relop`). The relational operators `<, <=, >, >=, ==` and `~=` can be used to compare two arrays of the same size or an array to a scalar. The logical operators `&, |` and `~` allow for the logical combination or negation of relational operators. The logical `&` and `|` have equal

Command	Result
`a = (b > c)`	`a` is `1` if `b` is larger than `c`. Similar are: `<`, `>=` and `<=`
`a = (b == c)`	`a` is `1` if `b` is equal to `c`
`a = (b ~= c)`	`a` is `1` if `b` is not equal `c`
`a = ~b`	logical complement: `a` is `1` if `b` is `0`
`a = (b & c)`	logical AND: `a` is `1` if `b` = TRUE AND `c` = TRUE
`a = (b | c)`	logical OR: `a` is `1` if `b` = TRUE OR `c` = TRUE

Table A.5: Relational and logical operations.

precedence in MATLAB, which means that those operators associate from left to right. In addition, three functions are also available: `xor`, `any` and `all` (use `help` to find out more).

The command `find` You can extract all elements from the vector or the matrix satisfying a given condition, e.g. equal to 1 or larger than 5, by using logical addressing. The same result can be obtained via the command `find`, which return the positions (indices) of such elements. For instance:

```
>> x = [1 1 3 4 1];
>> i = (x == 1)
i =
     1     1     0     0     1
>> j = find(x == 1)        % j holds indices of the elements satisfying x == 1
j =
     1     2     5
>> y = x(i), z = x(j)
y =
     1     1     1
z =
     1     1     1
```

`find` operates in a similar way on matrices. It reshapes first the matrix `A` into a column vector, i.e. all columns all concatenated one after another. Therefore, e.g. `k = find (A > 2)` is a list of indices of elements larger than 2 and `A(k)` gives the values. The row and column indices can be returned by `[I,J] = find (A > 2)`.

A.3.2 Conditional code execution

Selection control structures, `if`-blocks, are used to decide which instruction to execute next depending whether *expression* is TRUE or not. The general description is given below. In the examples below the command `disp` is frequently used. This command displays on the screen the text between the quotes.

- `if ... end`

Syntax

```
if logical_expression
   statement1
   statement2
   ....
end
```

Example

```
if (a > 0)
  b = a;
  disp ('a is positive');
end
```

- `if ... else ... end`

Syntax

```
if logical_expression
   block of statements
   evaluated if TRUE
else
   block of statements
   evaluated if FALSE
end
```

Example

```
if (temperature > 100)
  disp ('Above boiling.');
  toohigh = 1;
else
  disp ('Temperature is OK.');
  toohigh = 0;
end
```

- `if ... elseif ... else ... end`

Syntax

```
if logical_expression1
   block of statements evaluated
   if logical_expression1 is TRUE
elseif logical_expression2
   block of statements evaluated
   if logical_expression2 is TRUE
else
   block of statements evaluated
   if no other expression is TRUE
end
```

Example

```
if (height > 190)
  disp ('very tall');
elseif (height > 170)
  disp ('tall');
elseif (height < 150)
  disp ('small');
else
  disp ('average');
end
```

Another selection structure is `switch`, which switches between several cases depending on an expression, which is either a scalar or a string. Learn more by using `help`.

A.3.3 Loops

Iteration control structures, *loops*, are used to repeat a block of statements until some condition is met:

- the `for` loop that repeats a group of statements a *fixed* number of times;

Syntax

```
for index = first:step:last
  block of statements
end
```

Example

```
sumx = 0;
for i=1:length(x)
   sumx = sumx + x(i);
end
```

You can specify any `step`, including a negative value. The `index` of the `for`-loop can be also a vector. See some examples of possible variations:

Example 1

```
for i=1:2:n
  ...
end
```

Example 2

```
for i=n:-1:3
  ....
end
```

Example 3

```
for x=0:0.5:4
  disp(x^2);
end
```

Example 4

```
for x=[25 9 81]
  disp(sqrt(x));
end
```

- `while` loop, which evaluates a group of commands as long as *expression* is TRUE.

Syntax

```
while expression
  statement1
  statement2
  statement3
  ...
end
```

Example

```
N = 100;
iter = 1;
msum = 0;
while iter <= N
  msum = msum + iter;
  iter = iter + 1;
end;
```

A.4 Script and function m-files

A.4.1 Script m-files

Script m-files are useful when the number of commands increases or when you want to change values of some variables and re-evaluate them quickly. Formally, a script is an external file that contains a sequence of MATLAB commands. However, it is not a function, since there are no input/output parameters and the script variables remain in the workspace. **When you run a script, the commands in it are executed as if they have been entered through the keyboard.** To create a script, open the MATLAB editor (go to the **File** menubar, choose the **New** option and then **m-file** or **Script**; the MATLAB Editor Window will appear), enter the lines listed below and save as `sinplot.m`:

```
x = 0:0.2:6; y = sin(x); plot(x,y);
title('Plot of y = sin(x)');
```

The script can be run by calling `sinplot`. Note that m-script file must be saved in one of the directories in MATLAB's path. The `sinplot` script affects the workspace. Check:

```
>> clear                % all variables are removed from the workspace
>> who                  % no variables present
>> sinplot              % run the script
>> who
Your variables are:
x         y
```

These generic names, `x` and `y`, may be easily used in further computations and this can cause *side effects*. They occur, in general, when a set of commands change variables other than the input arguments. Remember that the commands within a script have access to all variables in the workspace and all variables created in this script become a part of the workspace. Therefore, it is better to use function m-files (see Section A.4.2) for a specific problem to be solved.

A.4.2 Function m-file

Functions m-files are true subprograms, as they take input arguments and/or return output parameters. They can call other functions, as well. Variables defined and used inside a function, different from the input/output arguments, are invisible to other functions and to the command environment. The general syntax is:

```
function [outputArgs] = function_name (inputArgs)
```

`outputArgs` are enclosed in `[ ]`:

- a comma-separated list of variable names;
- `[ ]` is optional when only one argument is present;
- functions without `outputArgs` are legal[2].

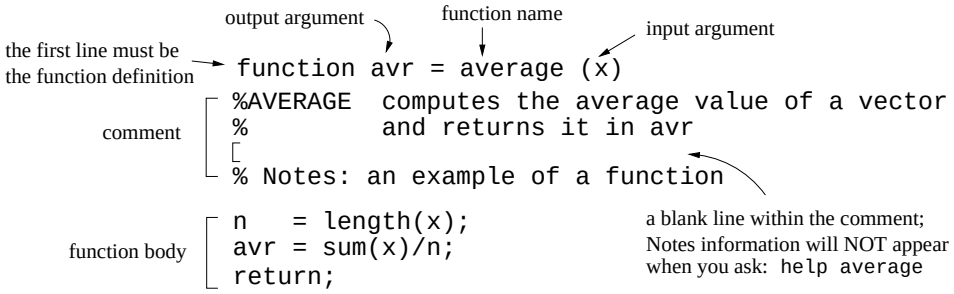
`inputArgs` are enclosed in `( )`:

- a comma-separated list of variable names;
- functions without `inputArgs` are legal.

MATLAB provides a structure for creating your own functions. The first line should be a definition of a new function (called a *header*). After that, a continuous sequence of comment lines, explaining what the function does, should appear. Since they are displayed in response to the `help` command, also the expected input parameters, returned output parameters and synopsis should be described there. Finally, the remainder of the function is called *the body*. Function m-files terminate execution when they reach the end of the file or, alternatively,

[2] In other programming languages, functions without output arguments are called *procedures*.

when the command `return` is encountered. The name of the function and the name of the file stored should be *identical*. As an example, the function `average`, stored in a file `average.m`, is defined as:



Here is an another example of a function:

```
function [avr,sd] = stat(x)
%STAT Simple statistics; computes the average and standard deviation of a vector x.
n   = length(x);
avr = sum(x)/n;
sd  = sqrt(sum((x - avr).^2)/n);
return;
```

Warning: The functions `mean` and `std` already exist in MATLAB. As long as a function name is used as a variable name, MATLAB can not perform the function. Many other, easily appealing names, such as `sum` or `prod` are reserved by MATLAB functions, so be careful when choosing your names.

When controlling the proper use of parameters, the function `error` may become useful. It displays an error message, aborts function execution, and returns to the command environment. Here is an example:

```
if (a >= 1)
  error ('a must be smaller than 1');
end;
```

A.4.3 Scripts vs. functions

The most important difference between a script and a function is that *all* script's parameters and variables are externally accessible (i.e. in the workspace), where function variables are not. Therefore, a script is a good tool for documenting work, designing experiments and testing. In general, create a function to solve a given problem for arbitrary parameters; use a script to run functions for specific parameters required by the assignment.

Appendix B

PRTools

B.1 Datasets

This section will give a brief overview of methods to generate standard datasets, and a number of other datasets available for loading. For more information on the `dataset` object, please refer to Day 1.

B.1.1 Data generation methods

Notation	
`p`	number of features, e.g. `p = 2`
`n`	number of samples (`na`, `nb` for classes A and B), e.g. `n = 20`
`c`	number of classes, e.g. `c = 2`
`u`	class mean: (1,p) vector (`ua`, `ub` for classes A and B), e.g. `u = [0,0]`
`v`	variance value, e.g. `v = 0.5`
`s`	class feature deviations: `(1,p)` vector, e.g. `s = [1,4]`
`G`	covariance matrix, size `(p,p)`, e.g. `G = [1 1; 1 4]`
`a`	dataset, size `(n,p)`
`lab`	label vector, size `(n,1)`

`dataset` generating routines	
`gauss`	Generation of multivariate Gaussian distributed data
`gendatb`	Generation of banana shaped classes in 2D
`gendatc`	Generation of circular classes
`gendatd`	Generation of two difficult classes
`gendath`	Generation of Higleyman classes in 2D
`gendatl`	Generation of Lithuanian classes in 2D
`gendatm`	Generation of 8 classes in 2D
`gendats`	Generation of two Gaussian distributed classes
`gencirc`	Generation of circle with radial noise in 2D
`lines5d`	Generation of three lines in 5D
`boomerang`	Generation two boomerang-shaped classes in 3D

Resampling routines	
`gendatk`	Nearest neighbour data generation
`gendatp`	Parzen density data generation
`gendat`	Generation of subsets of a given dataset

Data generation examples	
`a = rand(n,p).*(ones(n,1)*s)+ ones(n,1)*u`	Uniform distribution
`a = randn(n,p)*(ones(n,1)*s)+ ones(n,1)*u`	Normal distribution with diagonal covariance matrix `(s.*s)`
`lab = genlab(n,lablist)`	Generate a set of labels, `n(i)` times `lablist(i,:)`, for all values of `i`
`a = dataset(a,lab,featlab)`	Define a dataset from an array of feature vectors `a` and a set of labels `lab`, one for each datavector. Feature labels can be stored in `featlab`
`a = gauss(n,u,G)`	Arbitrary normal distribution
`a = gencirc(n,s)`	Noisy data on the perimeter of a circle
`a = gendatc([na,nb],p,ua)`	Two circular normally distributed classes
`a = gendatd([na,nb],p,d1,d2)`	Two 'difficult' normally distributed classes (pancakes)
`a = gendath(na,nb)`	Two classes of Highleyman (fixed normal distributions)
`a = gendatm(n)`	Generation of `n` objects in 8 normally distributed classes (means are generated randomly for each call)
`a = gendats([na,nb],p,d)`	Two 'simple' normally distributed classes, distance `d`
`a = gendatl([na,nb],v)`	Generate two 2D 'sausages'
`a = gendatk(b,n,k,v)`	Random generation by 'adding noise' to a given dataset `b` using the `k`-nearest neighbor method. The standard deviation is `v` $\times$ the nearest neighbour distance
`a = gendatp(b,n,v,G)`	Random generation from a Parzen density distribution based on the dataset `b` and smoothing parameter `v`. In case `G` is given it is used as covariance matrix of the kernel
`[b,c] = gendat(a,n)`	Generate at random two datasets out of one. The set `b` will have `n` objects per class, the remaining ones are stored in `c`

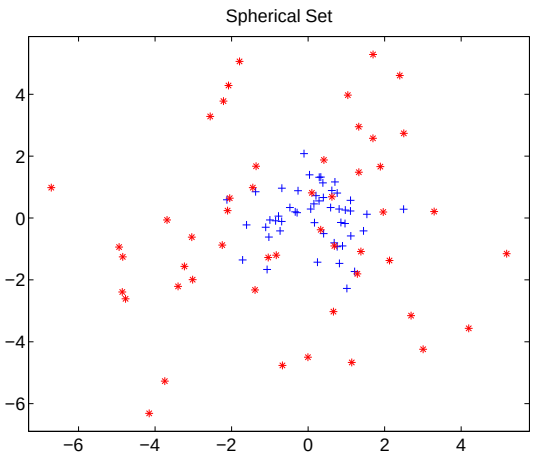
B.1.2 Datasets

In the table below, a list of datasets is given that can be stored in the variable `a` provided `prdatasets` is added to the path, e.g.:

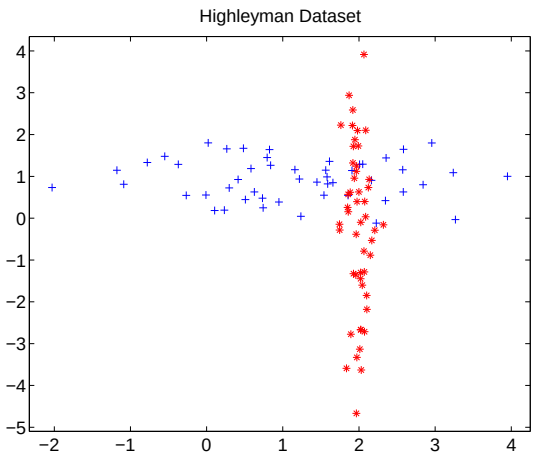
```
a = iris;
>> a
Iris plants, 150 by 4 dataset with 3 classes: [50  50  50]
```

Public domain datasets	
x80	45 by 8 with 3 classes: [15 15 15]
auto_mpg	398 by 6 with 2 classes: [229 169]
malaysia	291 by 8 with 20 classes
biomed	194 by 5 with 2 classes: [127 67]
breast	683 by 9 with 2 classes: [444 239]
cbands	12000 by 30 with 24 classes: [500 each]
chromo	1143 by 8 with 24 classes
circles3d	100 by 3 with 2 classes: [50 50]
diabetes	768 by 8 with 2 classes: [500 268]
ecoli	272 by 7 with 3 classes: [143 77 52]
glass	214 by 9 with 4 classes: [163 51]
heart	297 by 13 with 2 classes: [160 137]
imox	192 by 8 with 4 classes: [48 48 48 48]
iris	150 by 4 with 3 classes: [50 50 50]
ionosphere	351 by 34 with 2 classes: [225 126]
liver	345 by 6 with 2 classes: [145 200]
mfeat_fac	2000 by 216 with 10 classes: [200 each]
mfeat_fou	2000 by 76 with 10 classes: [200 each]
mfeat_kar	2000 by 64 with 10 classes: [200 each]
mfeat_mor	2000 by 6 with 10 classes: [200 each]
mfeat_pix	2000 by 240 with 10 classes: [200 each]
mfeat_zer	2000 by 47 with 10 classes: [200 each]
mfeat	2000 by 649 with 10 classes: [200 each]
nederland	12 by 12 with 12 classes: [1 each]
ringnorm	7400 by 20 with 2 classes: [3664 3736]
sonar	208 by 60 with 2 classes: [97 111]
soybean1	266 by 35 with 15 classes
soybean2	136 by 35 with 4 classes: [16 40 40 40]
spirals	194 by 2 with 2 classes: [97 97]
twonorm	7400 by 20 with 2 classes: [3703 3697]
wine	178 by 13 with 3 classes: [59 71 48]

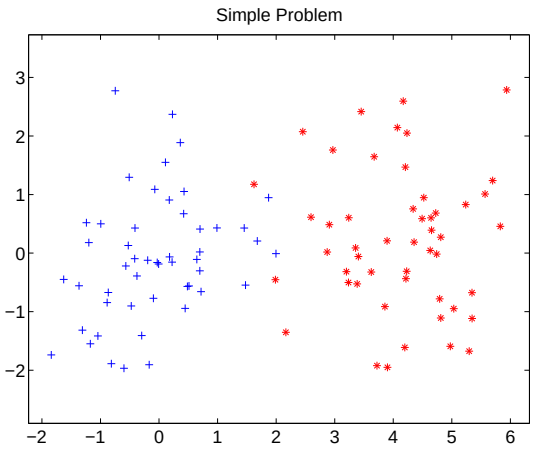
Other data loading routines	
prdataset	Read dataset stored in mat-file
prdata	Read data from file



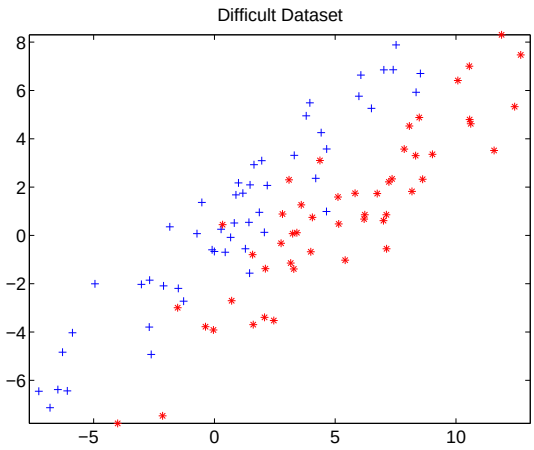

```
a = gendatc([50,50]); scatterd(a);
```



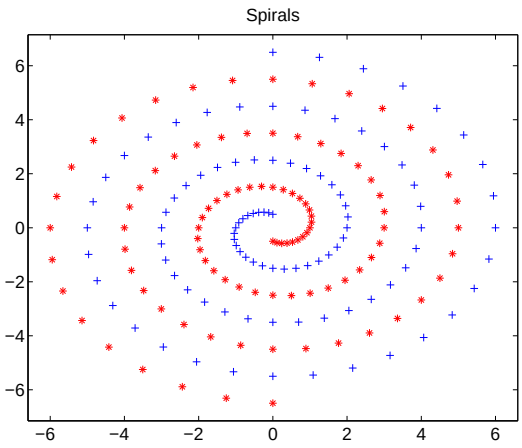

```
a = gendath([50,50]); scatterd(a);
```



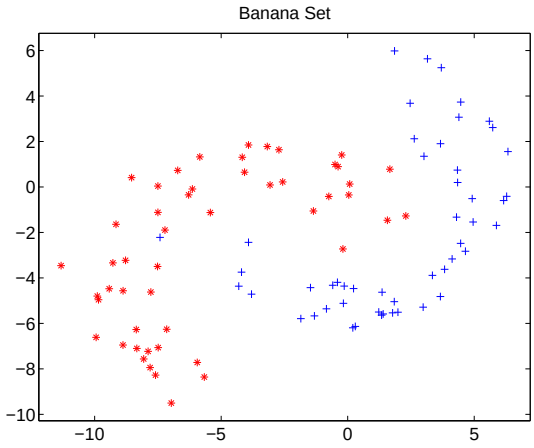

```
a = gendats([50,50],2,4);
scatterd(a); axis('equal');
```



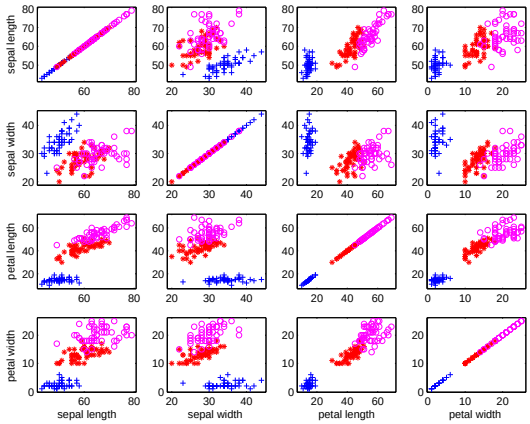

```
a = gendatd([50,50],2);
scatterd(a); axis('equal');
```




```
a = spirals; scatterd(a);
```




```
a = gendatb([50,50]); scatterd(a);
```




```
a = iris;
scatterd(a,'gridded');
```

B.2 Mappings

Data structures of the class `mapping` store trained classifiers, feature extraction results, data scaling definitions, nonlinear projections, etc. They are usually denoted by `w`.

`mapping`	
`mapping`	Define mapping
`getlab`	Retrieve labels assigned by mapping

A mapping `w` is often created by training a classifier on some data. For instance, the nearest mean classifier `nmc` is trained on some data `a` by:

```
>> a = gendatb(20);
>> w = nmc(a)
Nearest Mean, 2 to 2 trained classifier --> affine
```

`w` by itself, or `display(w)`, lists the size and type of a classifier as well as the routine which is used for computing the mapping `a*w`.

When a mapping is trained, it can be applied to a dataset, using the operator `*`:

```
>> b = a*w
Banana Set, 20 by 2 dataset with 2 classes: [7  13]
```

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`.

`mapping`s and classifiers	
`classc`	Converts a mapping into a classifier
`labeld`	General classification routine for trained classifiers
`testc`	General error estimation routine for trained classifiers

All routines operate on multi-class problems. For mappings which change the labels of the objects (so the mapping is actually a classifier) the routines `labeld` and `testc` are useful. `labeld` and `testc` are the general classification and testing routines respectively. They can handle any classifier from any routine.

Linear and polynomial classifiers	
`klldc`	Linear classifier by KL expansion of common cov matrix
`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
`pfsvc`	Pseudo-Fisher support vector classifier
`qdc`	Normal densities based quadratic (multi-class) classifier
`udc`	Uncorrelated normal densities based quadratic classifier

Nonlinear classifiers	
`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.
`parzen_map`	Parzen mapping routine
`testp`	Error estimation for Parzen classifier
`edicon`	Edit and condense training sets
`treec`	Construct binary decision tree classifier
`tree_map`	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

B.3 Training and testing

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

> if `a` is an `n` by `p` dataset (`n` objects in a `p`-dimensional space)
> and `w` is a `p` by `d` mapping (map from `p` to `d` dimensions)
> then `a*w` is an `n` by `d` dataset (`n` objects in a `d`-dimensional space).

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

Mappings are of the data type `mapping`, have a size of `[p,d]` if they map from `p` to `d` 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

OPTIONAL

The mapping of a test set `b` using `b*w1` is equivalent to `b*(a*v1)` or even, irregularly but sometimes useful, 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.

END OPTIONAL

B.4 Example

In this example a 2D 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`. Three classifiers are computed: a linear one and a quadratic one, both assuming normal densities (which is correct in this case) and a Parzen classifier. Note that the data generation use the random generator. 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.

```
%PREX_PLOTC  PRTools example on the dataset scatter and classifier plot
help prex_plotc
echo on
              % Generate Higleyman data
A = gendath([100 100]);
              % Split the data into the training and test sets
[C,D] = gendat(A,[20 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
              % Store classifiers in a cell
W = w1,w2,w3,w4;
              % Plot errors
disp(D*W*testc);
              % Plot the data and classifiers
figure
              % Make a scatter-plot
scatterd(A);
              % Plot classifiers
plotc(w1,w2,w3,w4);
echo off
```