CENTER FOR
MACHINE PERCEPTION

CZECH TECHNICAL
UNIVERSITY

RESEARCH REPORT

ISSN 1213-2365

Statistical Pattern Recognition Toolbox for Matlab

User's guide

Vojtěch Franc and Václav Hlaváč

{xfrancv,hlavac}@cmp.felk.cvut.cz

CTU–CMP–2004–08

June 24, 2004

Available at
ftp://cmp.felk.cvut.cz/pub/cmp/articles/Franc-TR-2004-08.pdf

The authors were supported by the European Union under project IST-2001-35454 ECVision, the project IST-2001-32184 ActIPret and by the Czech Grant Agency under project GACR 102/03/0440 and by the Austrian Ministry of Education under project CONEX GZ 45.535.

Research Reports of CMP, Czech Technical University in Prague, No. 8, 2004

Published by

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Statistical Pattern Recognition Toolbox for Matlab

Vojtěch Franc and Václav Hlaváč

June 24, 2004

Contents

Chapter 1

Introduction

1.1 What is the Statistical Pattern Recognition Toolbox and how it has been developed?

The Statistical Pattern Recognition Toolbox (abbreviated STPRtool) is a collection of pattern recognition (PR) methods implemented in Matlab.

The core of the STPRtool is comprised of statistical PR algorithms described in the monograph *Schlesinger, M.I., Hlaváč, V: Ten Lectures on Statistical and Structural Pattern Recognition, Kluwer Academic Publishers, 2002* [26]. The STPRtool has been developed since 1999. The first version was a result of a diploma project of Vojtěch Franc [8]. Its author became a PhD student in CMP and has been extending the STPRtool since. Currently, the STPRtool contains much wider collection of algorithms.

The STPRtool is available at

`http://cmp.felk.cvut.cz/cmp/cmp_software.html`.

We tried to create concise and complete documentation of the STPRtool. This document and associated help files in *.html* should give all information and parameters the user needs when she/he likes to call appropriate method (typically a Matlab *m-file*). The intention was not to write another textbook. If the user does not know the method then she/he will likely have to read the description of the method in a referenced original paper or in a textbook.

1.2 The STPRtool purpose and its potential users

- The principal purpose of STPRtool is to provide basic statistical pattern recognition methods to (a) researchers, (b) teachers, and (c) students. The STPRtool is likely to help both in research and teaching. It can be found useful by practitioners who design PR applications too.

- The STPRtool offers a collection of the basic and the state-of-the art statistical PR methods.
- The STPRtool contains demonstration programs, working examples and visualization tools which help to understand the PR methods. These teaching aids are intended for students of the PR courses and their teachers who can easily create examples and incorporate them into their lectures.
- The STPRtool is also intended to serve a designer of a new PR method by providing tools for evaluation and comparison of PR methods. Many standard and state-of-the art methods have been implemented in STPRtool.
- The STPRtool is open to contributions by others. If you develop a method which you consider useful for the community then contact us. We are willing to incorporate the method into the toolbox. We intend to moderate this process to keep the toolbox consistent.

1.3 A digest of the implemented methods

This section should give the reader a quick overview of the methods implemented in STPRtool.

- **Analysis of linear discriminant function:** Perceptron algorithm and multi-class modification. Kozinec's algorithm. Fisher Linear Discriminant. A collection of known algorithms solving the Generalized Anderson's Task.
- **Feature extraction:** Linear Discriminant Analysis. Principal Component Analysis (PCA). Kernel PCA. Greedy Kernel PCA. Generalized Discriminant Analysis.
- **Probability distribution estimation and clustering:** Gaussian Mixture Models. Expectation-Maximization algorithm. Minimax probability estimation. K-means clustering.
- **Support Vector and other Kernel Machines:** Sequential Minimal Optimizer (SMO). Matlab Optimization toolbox based algorithms. Interface to the SVM^{light} software. Decomposition approaches to train the Multi-class SVM classifiers. Multi-class BSVM formulation trained by Kozinec's algorithm, Mitchell-Demyanov-Molozenov algorithm and Nearest Point Algorithm. Kernel Fisher Discriminant.

1.4 Relevant Matlab toolboxes of others

Some researchers in statistical PR provide code of their methods. The Internet offers many software resources for Matlab. However, there are not many compact and well documented packages comprised of PR methods. We found two useful packages which are briefly introduced below.

NETLAB is focused to the Neural Networks applied to PR problems. The classical PR methods are included as well. This Matlab toolbox is closely related to the popular PR textbook [3]. The documentation published in book [21] contains detailed description of the implemented methods and many working examples. The toolbox includes: Probabilistic Principal Component Analysis, Generative Topographic Mapping, methods based on the Bayesian inference, Gaussian processes, Radial Basis Functions (RBF) networks and many others.

The toolbox can be downloaded from `http://www.ncrg.aston.ac.uk/netlab/`.

PRTools is a Matlab Toolbox for Pattern Recognition [7]. Its implementation is based on the object oriented programming principles supported by the Matlab language. The toolbox contains a wide range of PR methods including: analysis of linear and non-linear classifiers, feature extraction, feature selection, methods for combining classifiers and methods for clustering.

The toolbox can be downloaded from `http://www.ph.tn.tudelft.nl/prtools/`.

1.5 STPRtool document organization

The document is divided into chapters describing the implemented methods. The last chapter is devoted to more complex examples. Most chapters begin with definition of the models to be analyzed (e.g., linear classifier, probability densities, etc.) and the list of the implemented methods. Individual sections have the following fixed structure: (i) definition of the problem to be solved, (ii) brief description and list of the implemented methods, (iii) reference to the literature where the problem is treaded in details and (iv) an example demonstrating how to solve the problem using the STPRtool.

1.6 How to contribute to STPRtool?

If you have an implementation which is consistent with the STPRtool spirit and you want to put it into public domain through the STPRtool channel then send an email to xfrancv@cmp.felk.cvut.cz.

Please prepare your code according to the pattern you find in STPRtool implementations. Provide piece of documentation too which will allow us to incorporate it to STPRtool easily.

Of course, your method will be bound to your name in STPRtool.

1.7 Acknowledgements

First, we like to thank to Prof. M.I. Schlesinger from the Ukrainian Academy of Sciences, Kiev, Ukraine who showed us the beauty of pattern recognition and gave us many valuable advices.

Second, we like to thank for discussions to our colleagues in the Center for Machine Perception, the Czech Technical University in Prague, our visitors, other STPRtool users and students who used the toolbox in their laboratory exercises.

Third, thanks go to several Czech, EU and industrial grants which indirectly contributed to STPRtool development.

Forth, the STPRtool was rewritten and its documentation radically improved in May and June 2004 with the financial help of ECVision European Research Network on Cognitive Computer Vision Systems (IST-2001-35454) lead by D. Vernon.

Notation

Upper-case bold letters denote matrices, for instance $\mathbf{A}$. Vectors are implicitly column vectors. Vectors are denoted by lower-case bold italic letters, for instance $\boldsymbol{x}$. The concatenation of column vectors $\boldsymbol{w} \in \mathbb{R}^n$, $\boldsymbol{z} \in \mathbb{R}^m$ is denoted as $\boldsymbol{v} = [\boldsymbol{w}; \boldsymbol{z}]$ where the column vector $\boldsymbol{v}$ is $(n+m)$-dimensional.

$\langle \boldsymbol{x} \cdot \boldsymbol{x}' \rangle$	Dot product between vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$.
$\boldsymbol{\Sigma}$	Covariance matrix.
$\boldsymbol{\mu}$	Mean vector (mathematical expectation).
$\mathbf{S}$	Scatter matrix.
$\mathcal{X} \subseteq \mathbb{R}^n$	n-dimensional input space (space of observations).
$\mathcal{Y}$	Set of c hidden states – class labels $\mathcal{Y} = \{1, \dots, c\}$.
$\mathcal{F}$	Feature space.
$\boldsymbol{\phi}: \mathcal{X} \to \mathcal{F}$	Mapping from input space $\mathcal{X}$ to the feature space $\mathcal{F}$.
$k: \mathcal{X} \times \mathcal{X} \to \mathbb{R}$	Kernel function (Mercer kernel).
$\mathcal{T}_{XY}$	Labeled training set (more precisely multi-set) $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_l, y_l)\}$.
$\mathcal{T}_X$	Unlabeled training set $\mathcal{T}_X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_l\}$.
$q: \mathcal{X} \to \mathcal{Y}$	Classification rule.
$f_y: \mathcal{X} \to \mathbb{R}$	Discriminant function associated with the class $y \in \mathcal{Y}$.
$f: \mathcal{X} \to \mathbb{R}$	Discriminant function of the binary classifier.
$\lvert \cdot \rvert$	Cardinality of a set.
$\lVert \cdot \rVert$	Euclidean norm.
$\det(() \cdot)$	Matrix determinant.
$\delta(i,j)$	Kronecker delta $\delta(i,j) = 1$ for $i = j$ and $\delta(i,j) = 0$ otherwise.
$\bigvee$	Logical or.

Chapter 2

Linear Discriminant Function

The linear classification rule $q\colon \mathcal{X} \subseteq \mathbb{R}^n \to \mathcal{Y} = \{1, 2, \ldots, c\}$ is composed of a set of discriminant functions

$$f_y(\boldsymbol{x}) = \langle \boldsymbol{w}_y \cdot \boldsymbol{x} \rangle + b_y \,, \qquad \forall y \in \mathcal{Y} \,,$$

which are linear with respect to both the input vector $\boldsymbol{x} \in \mathbb{R}^n$ and their parameter vectors $\boldsymbol{w} \in \mathbb{R}^n$. The scalars b_y, $\forall y \in \mathcal{Y}$ introduce bias to the discriminant functions. The input vector $\boldsymbol{x} \in \mathbb{R}^n$ is assigned to the class $y \in \mathcal{Y}$ its corresponding discriminant function f_y attains maximal value

$$y = \operatorname*{argmax}_{y' \in \mathcal{Y}} f_{y'}(\boldsymbol{x}) = \operatorname*{argmax}_{y' \in \mathcal{Y}} \left(\langle \boldsymbol{w}_{y'} \cdot \boldsymbol{x} \rangle + b_{y'} \right) \,. \tag{2.1}$$

In the particular binary case $\mathcal{Y} = \{1, 2\}$, the linear classifier is represented by a single discriminant function

$$f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b \,,$$

given by the parameter vector $\boldsymbol{w} \in \mathbb{R}^n$ and bias $b \in \mathbb{R}$. The input vector $\boldsymbol{x} \in \mathbb{R}^n$ is assigned to class $y \in \{1, 2\}$ as follows

$$q(\boldsymbol{x}) = \left\{ \begin{array}{lll} 1 & \text{if} & f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b \geq 0 \,, \\ 2 & \text{if} & f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b < 0 \,. \end{array} \right. \tag{2.2}$$

The data-type used to describe the linear classifier and the implementation of the classifier is described in Section 2.1. Following sections describe supervised learning methods which determine the parameters of the linear classifier based on available knowledge. The implemented learning methods can be distinguished according to the type of the training data:

I. The problem is described by a finite set $\mathcal{T} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ containing pairs of observations $\boldsymbol{x}_i \in \mathbb{R}^n$ and corresponding class labels $y_i \in \mathcal{Y}$. The methods using this type of training data is described in Section 2.2 and Section 2.3.

II. The parameters of class conditional distributions are (completely or partially) known. The Anderson's task and its generalization are representatives of such methods (see Section 2.4).

The summary of implemented methods is given in Table 2.1.
References: The linear discriminant function is treated throughout for instance in books [26, 6].

Table 2.1: Implemented methods: Linear discriminant function

`linclass`	Linear classifier (linear machine).
`andrerr`	Classification error of the Generalized Anderson's task.
`androrig`	Original method to solve the Anderson-Bahadur's task.
`eanders`	Epsilon-solution of the Generalized Anderson's task.
`ganders`	Solves the Generalized Anderson's task.
`ggradander`	Gradient method to solve the Generalized Anderson's task.
`ekozinec`	Kozinec's algorithm for eps-optimal separating hyperplane.
`mperceptr`	Perceptron algorithm to train multi-class linear machine.
`perceptron`	Perceptron algorithm to train binary linear classifier.
`fld`	Fisher Linear Discriminant.
`fldqp`	Fisher Linear Discriminant using Quadratic Programming.
`demo_linclass`	Demo on the algorithms learning linear classifiers.
`demo_anderson`	Demo on Generalized Anderson's task.

2.1 Linear classifier

The STPRtool uses a specific structure array to describe the binary (see Table 2.2) and the multi-class (see Table 2.3) linear classifier. The implementation of the linear classifier itself provides function `linclass`.

Example: Linear classifier

The example shows training of the Fisher Linear Discriminant which is the classical example of the linear classifier. The Riply's data set `riply_trn.mat` is used for training. The resulting linear classifier is then evaluated on the testing data `riply_tst.mat`.

Table 2.2: Data-type used to describe binary linear classifier.

Binary linear classifier (structure array):	
`.W` $[n \times 1]$	The normal vector $\boldsymbol{w} \in \mathbb{R}^n$ of the separating hyperplane $f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b$.
`.b` $[1 \times 1]$	Bias $b \in \mathbb{R}$ of the hyperplane.
`.fun = 'linclass'`	Identifies function associated with this data type.

Table 2.3: Data-type used to describe multi-class linear classifier.

Multi-class linear classifier (structure array):	
`.W` $[n \times c]$	Matrix of parameter vectors $\boldsymbol{w}_y \in \mathbb{R}^n$, $y = 1, \ldots, c$ of the linear discriminant functions $f_y(\boldsymbol{x}) = \langle \boldsymbol{w}_y \cdot \boldsymbol{x} \rangle + b$.
`.b` $[c \times 1]$	Parameters $b_y \in \mathbb{R}$, $y = 1, \ldots, c$.
`.fun = 'linclass'`	Identifies function associated with this data type.

```
  trn = load('riply_trn');            % load training data
  tst = load('riply_tst');            % load testing data
  model = fld( trn );                 % train FLD classifier
  ypred = linclass( tst.X, model );   % classify testing data
  cerror( ypred, tst.y )              % evaluate testing error

ans =
    0.1080
```

2.2 Linear separation of finite sets of vectors

The input training data $\mathcal{T} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ consists of pairs of observations $\boldsymbol{x}_i \in \mathbb{R}^n$ and corresponding class labels $y_i \in \mathcal{Y} = \{1, \ldots, c\}$. The implemented methods solve the task of (i) training separating hyperplane for binary case $c = 2$, (ii) training ε-optimal hyperplane and (iii) training the multi-class linear classifier. The definitions of these tasks are given below.

The problem of training the binary ($c = 2$) linear classifier (2.2) with zero training error is equivalent to finding the hyperplane $\mathcal{H} = \{\boldsymbol{x} \colon \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b = 0\}$ which separates the training vectors of the first $y = 1$ and the second $y = 2$ class. The problem is formally defined as solving the set of linear inequalities

$$\begin{array}{rcll} \langle \boldsymbol{w} \cdot \boldsymbol{x}_i \rangle + b & \geq & 0\,, & y_i = 1\,, \\ \langle \boldsymbol{w} \cdot \boldsymbol{x}_i \rangle + b & < & 0\,, & y_i = 2\,. \end{array} \tag{2.3}$$

with respect to the vector $\boldsymbol{w} \in \mathbb{R}^n$ and the scalar $b \in \mathbb{R}$. If the inequalities (2.3) have a solution then the training data $\mathcal{T}$ is *linearly separable*. The Perceptron (Section 2.2.1) and Kozinec's (Section 2.2.2) algorithm are useful to train the binary linear classifiers from the linearly separable data.

If the training data is linearly separable then the optimal separating hyperplane can be defined the solution of the following task

$$\begin{aligned}(\boldsymbol{w}^*, b^*) &= \operatorname*{argmax}_{\boldsymbol{w},b} m(\boldsymbol{w}, b) \\ &= \operatorname*{argmax}_{\boldsymbol{w},b} \min\left(\min_{i\in\mathcal{I}_1} \frac{\langle \boldsymbol{w}\cdot\boldsymbol{x}_i\rangle + b}{\|\boldsymbol{w}\|}, \min_{i\in\mathcal{I}_2} -\frac{\langle \boldsymbol{w}\cdot\boldsymbol{x}_i\rangle + b}{\|\boldsymbol{w}\|}\right) ,\end{aligned} \tag{2.4}$$

where $\mathcal{I}_1 = \{i: y_i = 1\}$ and $\mathcal{I}_2 = \{i: y_i = 2\}$ are sets of indices. The optimal separating hyperplane $\mathcal{H}^* = \{\boldsymbol{x} \in \mathbb{R}^n: \langle \boldsymbol{w}^* \cdot \boldsymbol{x}\rangle + b^* = 0\}$ separates training data $\mathcal{T}$ with maximal margin $m(\boldsymbol{w}^*, b^*)$. This task is equivalent to training the linear Support Vector Machines (see Section 5). The optimal separating hyperplane cannot be found exactly except special cases. Therefore the numerical algorithms seeking the approximate solution are applied instead. The ε-optimal solution is defined as the vector $\boldsymbol{w}$ and scalar b such that the inequality

$$m(\boldsymbol{w}^*, b^*) - m(\boldsymbol{w}, b) \leq \varepsilon , \tag{2.5}$$

holds. The parameter $\varepsilon \geq 0$ defines closeness to the optimal solution in terms of the margin. The ε-optimal solution can be found by Kozinec's algorithm (Section 2.2.2).

The problem of training the multi-class linear classifier $c > 2$ with zero training error is formally stated as the problem of solving the set of linear inequalities

$$\langle \boldsymbol{w}_{y_i} \cdot \boldsymbol{x}_i\rangle + b_{y_i} > \langle \boldsymbol{w}_y \cdot \boldsymbol{x}_i\rangle + b_y , \qquad i = 1, \dots, l , \quad y_i \neq y , \tag{2.6}$$

with respect to the vectors $\boldsymbol{w}_y \in \mathbb{R}^n$, $y \in \mathcal{Y}$ and scalars $b_y \in \mathbb{R}$, $y \in \mathcal{Y}$. The task (2.6) for linearly separable training data $\mathcal{T}$ can be solved by the modified Perceptron (Section 2.2.3) or Kozinec's algorithm. An interactive demo on the algorithms separating the finite sets of vectors by a hyperplane is implemented in `demo_linclass`.

2.2.1 Perceptron algorithm

The input is a set $\mathcal{T} = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_l, y_l)\}$ of binary labeled $y_i \in \{1, 2\}$ training vectors $\boldsymbol{x}_i \in \mathbb{R}^n$. The problem of training the separating hyperplane (2.3) can be formally rewritten to a simpler form

$$\langle \boldsymbol{v} \cdot \boldsymbol{z}_i\rangle > 0 , \qquad i = 1, \dots, l , \tag{2.7}$$

where the vector $\boldsymbol{v} \in \mathbb{R}^{n+1}$ is constructed as

$$\boldsymbol{v} = [\boldsymbol{w}; b] , \tag{2.8}$$

and transformed training data $\mathcal{Z} = \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_l\}$ are defined as

$$\boldsymbol{z}_i = \begin{cases} [\boldsymbol{x}_i; 1] & \text{if} \quad y_i = 1 \,, \\ -[\boldsymbol{x}_i; 1] & \text{if} \quad y_i = 2 \,. \end{cases} \tag{2.9}$$

The problem of solving (2.7) with respect to the unknown vector $\boldsymbol{v} \in \mathbb{R}^{n+1}$ is equivalent to the original task (2.3). The parameters $(\boldsymbol{w}, b)$ of the linear classifier are obtained from the found vector $\boldsymbol{v}$ by inverting the transformation (2.8).

The Perceptron algorithm is an iterative procedure which builds a series of vectors $\boldsymbol{v}^{(0)}, \boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(t)}$ until the set of inequalities (2.7) is satisfied. The initial vector $\boldsymbol{v}^{(0)}$ can be set arbitrarily (usually $\boldsymbol{v} = \boldsymbol{0}$). The Novikoff theorem ensures that the Perceptron algorithm stops after finite number of iterations t if the training data are linearly separable. Perceptron algorithm is implemented in function `perceptron`.

References: Analysis of the Perceptron algorithm including the Novikoff's proof of convergence can be found in Chapter 5 of the book [26]. The Perceptron from the neural network perspective is described in the book [3].

Example: Training binary linear classifier with Perceptron

The example shows the application of the Perceptron algorithm to find the binary linear classifier for synthetically generated 2-dimensional training data. The generated training data contain 50 labeled vectors which are linearly separable with margin 1. The found classifier is visualized as a separating hyperplane (line in this case) $\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^2 \colon \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b = 0\}$. See Figure 2.1.

```
data = genlsdata( 2, 50, 1);    % generate training data
model = perceptron( data );     % call perceptron
figure;
ppatterns( data );              % plot training data
pline( model );                 % plot separating hyperplane
```

2.2.2 Kozinec's algorithm

The input is a set $\mathcal{T} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ of binary labeled $y_i \in \{1, 2\}$ training vectors $\boldsymbol{x}_i \in \mathbb{R}^n$. The Kozinec's algorithm builds a series of vectors $\boldsymbol{w}_1^{(0)}, \boldsymbol{w}_1^{(1)}, \ldots, \boldsymbol{w}_1^{(t)}$ and $\boldsymbol{w}_2^{(0)}, \boldsymbol{w}_2^{(1)}, \ldots, \boldsymbol{w}_2^{(t)}$ which converge to the vector $\boldsymbol{w}_1^*$ and $\boldsymbol{w}_2^*$ respectively. The vectors $\boldsymbol{w}_1^*$ and $\boldsymbol{w}_2^*$ are the solution of the following task

$$\boldsymbol{w}_1^*, \boldsymbol{w}_2^* = \operatorname*{argmin}_{\boldsymbol{w}_1 \in \overline{\mathcal{X}_1}, \boldsymbol{w}_2 \in \overline{\mathcal{X}_2}} \|\boldsymbol{w}_1 - \boldsymbol{w}_2\| \,,$$

where $\overline{\mathcal{X}_1}$ stands for the convex hull of the training vectors of the first class $\mathcal{X}_1 = \{\boldsymbol{x}_i \colon y_i = 1\}$ and $\overline{\mathcal{X}_2}$ for the convex hull of the second class likewise. The vector $\boldsymbol{w}^* =$

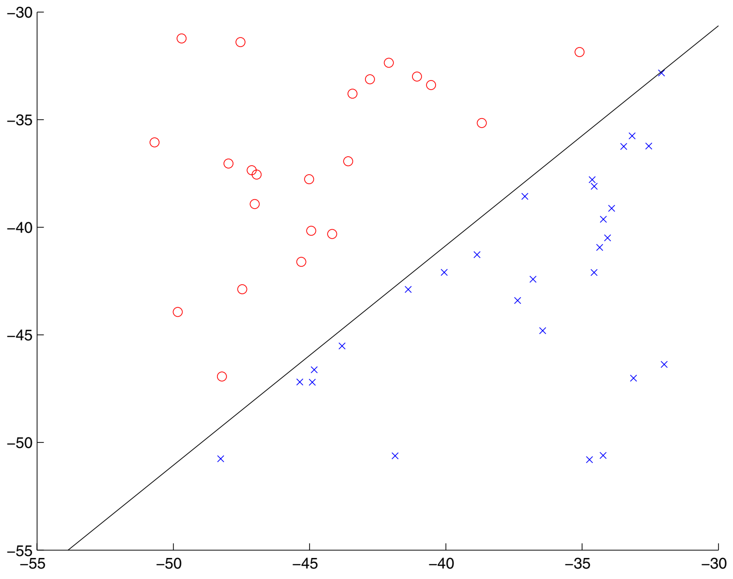


Figure 2.1: Linear classifier trained by the Perceptron algorithm

$\boldsymbol{w}_1^* - \boldsymbol{w}_2^*$ and the bias $b^* = \frac{1}{2}(\|\boldsymbol{w}_2^*\|^2 - \|\boldsymbol{w}_1^*\|^2)$ determine the optimal hyperplane (2.4) separating the training data with the maximal margin.

The Kozinec's algorithm is proven to converge to the vectors $\boldsymbol{w}_1^*$ and $\boldsymbol{w}_2^*$ in infinite number of iterations $t = \infty$. If the ε-optimal optimality stopping condition is used then the Kozinec's algorithm converges in finite number of iterations. The Kozinec's algorithm can be also used to solve a simpler problem of finding the separating hyperplane (2.3). Therefore the following two stopping conditions are implemented:

I. The separating hyperplane (2.3) is sought for ($\varepsilon < 0$). The Kozinec's algorithm is proven to converge in a finite number of iterations if the separating hyperplane exists.

II. The ε-optimal hyperplane (2.5) is sought for ($\varepsilon \geq 0$). Notice that setting $\varepsilon = 0$ force the algorithm to seek the optimal hyperplane which is generally ensured to be found in an infinite number of iterations $t = \infty$.

The Kozinec's algorithm which trains the binary linear classifier is implemented in the function `ekozinec`.

References: Analysis of the Kozinec's algorithm including the proof of convergence can be found in Chapter 5 of the book [26].

Example: Training ε-optimal hyperplane by the Kozinec's algorithm

The example shows application of the Kozinec's algorithm to find the ($\varepsilon = 0.01$)-optimal hyperplane for synthetically generated 2-dimensional training data. The generated training data contains 50 labeled vectors which are linearly separable with margin 1. The found classifier is visualized (Figure 2.2) as a separating hyperplane $\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^2 \colon \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b = 0\}$ (straight line in this case). It could be verified that the found hyperplane has margin `model.margin` which satisfies the ε-optimality condition.

```
data = genlsdata(2,50,1);           % generate data
% run ekozinec
model = ekozinec(data, struct('eps',0.01));
figure;
ppatterns(data);                    % plot data
pline(model);                       % plot found hyperplane
```

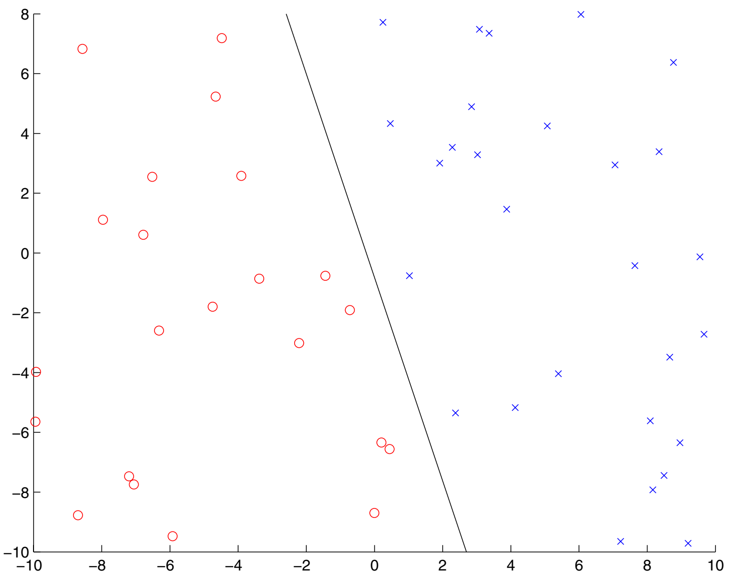


Figure 2.2: ε-optimal hyperplane trained with the Kozinec's algorithm.

2.2.3 Multi-class linear classifier

The input training data $\mathcal{T} = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_l, y_l)\}$ consists of pairs of observations $\boldsymbol{x}_i \in \mathbb{R}^n$ and corresponding class labels $y_i \in \mathcal{Y} = \{1, \dots, c\}$. The task is to design the linear classifier (2.1) which classifies the training data $\mathcal{T}$ without error. The problem of training (2.1) is equivalent to solving the set of linear inequalities (2.6). The inequalities (2.6) can be formally transformed to a simpler set

$$\langle \boldsymbol{v} \cdot \boldsymbol{z}_i^y \rangle > 0 \,, \qquad i = 1, \dots, l \,,\ y = \mathcal{Y} \setminus \{y_i\} \,, \tag{2.10}$$

where the vector $\boldsymbol{v} \in \mathbb{R}^{(n+1)c}$ contains the originally optimized parameters

$$\boldsymbol{v} = [\boldsymbol{w}_1; b_1; \boldsymbol{w}_2; b_2; \dots; \boldsymbol{w}_c; b_c] \,.$$

The set of vectors $\boldsymbol{z}_i^y \in \mathbb{R}^{(n+1)c}$, $i = 1, \ldots, l$, $y \in \mathcal{Y} \setminus \{y_i\}$ is created from the training set $\mathcal{T}$ such that

$$\boldsymbol{z}_i^y(j) = \begin{cases} [\boldsymbol{x}_i; 1]\,, & \text{for} \quad j = y_i\,, \\ -[\boldsymbol{x}_i; 1]\,, & \text{for} \quad j = y\,, \\ \mathbf{0}\,, & \text{otherwise.} \end{cases} \tag{2.11}$$

where $\boldsymbol{z}_i^y(j)$ stands for the j-th slot between coordinates $(n+1)(j-1)+1$ and $(n+1)j$. The described transformation is known as the Kesler's construction.

The transformed task (2.10) can be solved by the Perceptron algorithm. The simple form of the Perceptron updating rule allows to implement the transformation implicitly without mapping the training data $\mathcal{T}$ into $(n+1)c$-dimensional space. The described method is implemented in function `mperceptron`.

References: The Kesler's construction is described in books [26, 6].

Example: Training multi-class linear classifier by the Perceptron

The example shows application of the Perceptron rule to train the multi-class linear classifier for the synthetically generated 2-dimensional training data `pentagon.mat`. A user's own data can be generated interactively using function `createdata('finite',10)` (number 10 stands for number of classes). The found classifier is visualized as its separating surface which is known to be piecewise linear and the class regions should be convex (see Figure 2.3).

```
data = load('pentagon');       % load training data
model = mperceptron(data);     % run training algorithm
figure;
ppatterns(data);               % plot training data
pboundary(model);              % plot decision boundary
```

2.3 Fisher Linear Discriminant

The input is a set $\mathcal{T} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ of binary labeled $y_i \in \{1, 2\}$ training vectors $\boldsymbol{x}_i \in \mathbb{R}^n$. Let $\mathcal{I}_y = \{i : y_i = y\}$, $y \in \{1, 2\}$ be sets of indices of training vectors belonging to the first $y = 1$ and the second $y = 2$ class, respectively. The class separability in a direction $\boldsymbol{w} \in \mathbb{R}^n$ is defined as

$$F(\boldsymbol{w}) = \frac{\langle \boldsymbol{w} \cdot \mathbf{S_B} \boldsymbol{w} \rangle}{\langle \boldsymbol{w} \cdot \mathbf{S_W} \boldsymbol{w} \rangle}\,, \tag{2.12}$$

where $\mathbf{S_B}$ is the between-class scatter matrix

$$\mathbf{S_B} = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T\,, \qquad \boldsymbol{\mu}_y = \frac{1}{|\mathcal{I}_y|} \sum_{i \in \mathcal{I}_y} \boldsymbol{x}_i\,,\ y \in \{1, 2\}\,,$$

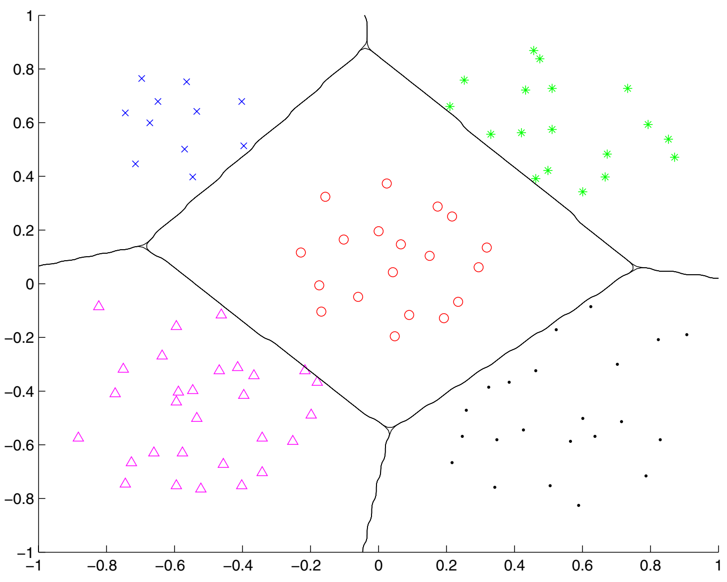


Figure 2.3: Multi-class linear classifier trained by the Perceptron algorithm.

and $\mathbf{S}_W$ is the within class scatter matrix defined as

$$\mathbf{S_W} = \mathbf{S}_1 + \mathbf{S}_2 \,, \qquad \mathbf{S}_y = \sum_{i \in \mathcal{Y}_y} (\boldsymbol{x}_i - \boldsymbol{\mu}_y)(\boldsymbol{x}_i - \boldsymbol{\mu}_y)^T \,, \; y \in \{1, 2\} \,.$$

In the case of the Fisher Linear Discriminant (FLD), the parameter vector $\boldsymbol{w}$ of the linear discriminant function $f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b$ is determined to maximize the class separability criterion (2.12)

$$\boldsymbol{w} = \operatorname*{argmax}_{\boldsymbol{w}'} F(\boldsymbol{w}') = \operatorname*{argmax}_{\boldsymbol{w}'} \frac{\langle \boldsymbol{w}' \cdot \mathbf{S_B} \boldsymbol{w}' \rangle}{\langle \boldsymbol{w}' \cdot \mathbf{S_W} \boldsymbol{w}' \rangle} \,. \tag{2.13}$$

The bias b of the linear rule must be determined based on another principle. The STPRtool implementation, for instance, computes such b that the equality

$$\langle \boldsymbol{w} \cdot \mu_1 \rangle + b = -(\langle \boldsymbol{w} \cdot \mu_2 \rangle + b) \,,$$

holds. The STPRtool contains two implementations of FLD:

I. The classical solution of the problem (2.13) using the matrix inversion

$$\boldsymbol{w} = \mathbf{S_W}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \,.$$

This case is implemented in the function `fld`.

II. The problem (2.13) can be reformulated as the quadratic programming (QP) task

$$\boldsymbol{w} = \operatorname*{argmin}_{\boldsymbol{w}'} \langle \boldsymbol{w}' \cdot \mathbf{S_W} \boldsymbol{w}' \rangle \,,$$

subject to

$$\langle \boldsymbol{w}' \cdot (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \rangle = 2 \,.$$

The QP solver `quadprog` of the Matlab Optimization Toolbox is used in the toolbox implementation. This method can be useful when the matrix inversion is hard to compute. The method is implemented in the function `fldqp`.

References: The Fisher Linear Discriminant is described in Chapter 3 of the book [6] or in the book [3].

Example: Fisher Linear Discriminant

The binary linear classifier based on the Fisher Linear Discriminant is trained on the Riply's data set `riply_trn.mat`. The QP formulation `fldqp` of FLD is used. However, the function `fldqp` can be replaced by the standard approach implemented in the function `fld` which would yield the same solution. The found linear classifier is visualized (see Figure 2.4) and evaluated on the testing data `riply_tst.mat`.

```
   trn = load('riply_trn');        % load training data
   model = fldqp(trn);             % compute FLD
   figure;
   ppatterns(trn); pline(model);   % plot data and solution
   tst = load('riply_tst');        % load testing data
   ypred = linclass(tst.X,model);  % classify testing data
   cerror(ypred,tst.y)             % compute testing error

ans =

    0.1080
```

2.4 Generalized Anderson's task

The classified object is described by the vector of observations $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^n$ and a binary hidden state $y \in \{1,2\}$. The class conditional distributions $p_{X|Y}(\boldsymbol{x}|y)$, $y \in \{1,2\}$ are known to be multi-variate Gaussian distributions. The parameters $(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ and $(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ of these class distributions are unknown. However, it is known that the parameters $(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ belong to a certain finite set of parameters $\{(\boldsymbol{\mu}^i, \boldsymbol{\Sigma}^i)\colon i \in \mathcal{I}_1\}$. Similarly $(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ belong to a finite set $\{(\boldsymbol{\mu}^i, \boldsymbol{\Sigma}^i)\colon i \in \mathcal{I}_2\}$. Let $q\colon \mathcal{X} \subseteq \mathbb{R}^n \rightarrow \{1,2\}$

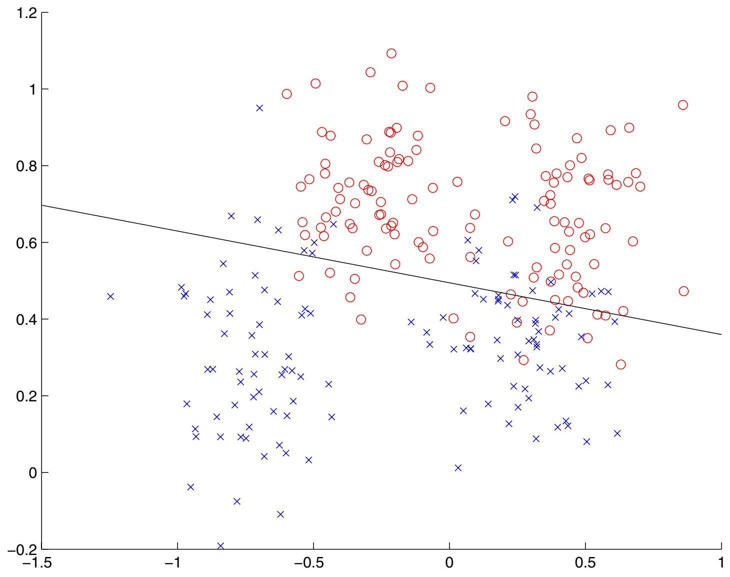


Figure 2.4: Linear classifier based on the Fisher Linear Discriminant.

be a binary linear classifier (2.2) with discriminant function $f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b$. The probability of misclassification is defined as

$$\mathrm{Err}(\boldsymbol{w}, b) = \max_{i \in \mathcal{I}_1 \cup \mathcal{I}_2} \varepsilon(\boldsymbol{w}, b, \boldsymbol{\mu}^i, \boldsymbol{\Sigma}^i) \,,$$

where $\varepsilon(\boldsymbol{w}, b, \boldsymbol{\mu}^i, \boldsymbol{\Sigma}^{\mathbf{i}})$ is probability that the Gaussian random vector $\boldsymbol{x}$ with mean vector $\boldsymbol{\mu}^i$ and the covariance matrix $\boldsymbol{\Sigma}^i$ satisfies $q(\boldsymbol{x}) = 1$ for $i \in \mathcal{I}_2$ or $q(\boldsymbol{x}) = 2$ for $i \in \mathcal{I}_1$. In other words, it is the probability that the vector $\boldsymbol{x}$ will be misclassified by the linear rule q.

The Generalized Anderson's task (GAT) is to find the parameters $(\boldsymbol{w}^*, b^*)$ of the linear classifier

$$q(\boldsymbol{x}) = \begin{cases} 1\,, & \text{for} \quad f(\boldsymbol{x}) = \langle \boldsymbol{w}^* \cdot \boldsymbol{x} \rangle + b^* \geq 0\,, \\ 2\,, & \text{for} \quad f(\boldsymbol{x}) = \langle \boldsymbol{w}^* \cdot \boldsymbol{x} \rangle + b^* < 0\,, \end{cases}$$

such that the error $\mathrm{Err}(\boldsymbol{w}^*, b^*)$ is minimal

$$(\boldsymbol{w}^*, b^*) = \operatorname*{argmin}_{\boldsymbol{w}, b} \mathrm{Err}(\boldsymbol{w}, b) = \operatorname*{argmin}_{\boldsymbol{w}, b} \max_{i \in \mathcal{I}_1 \cup \mathcal{I}_2} \varepsilon(\boldsymbol{w}, b, \boldsymbol{\mu}^i, \boldsymbol{\Sigma}^i) \,. \tag{2.14}$$

The original Anderson's task is a special case of (2.14) when $|\mathcal{I}_1| = 1$ and $|\mathcal{I}_2| = 1$.

The probability $\varepsilon(\boldsymbol{w}, b, \boldsymbol{\mu}^i, \boldsymbol{\Sigma}^i)$ is proportional to the reciprocal of the Mahalanobis distance r^i between the $(\boldsymbol{\mu}^i, \boldsymbol{\Sigma}^i)$ and the nearest vector of the separating hyperplane $\mathcal{H} = \{\boldsymbol{x} \in \mathbb{R}^n \colon \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b = 0\}$, i.e.,

$$r^i = \min_{\boldsymbol{x} \in \mathcal{H}} \langle (\boldsymbol{\mu}^i - \boldsymbol{x}) \cdot (\boldsymbol{\Sigma}^i)^{-1} (\boldsymbol{\mu}^i - \boldsymbol{x}) \rangle = \frac{\langle \boldsymbol{w} \cdot \boldsymbol{\mu}^i \rangle + b}{\sqrt{\langle \boldsymbol{w} \cdot \boldsymbol{\Sigma}^i \boldsymbol{w} \rangle}} \,.$$

The exact relation between the probability $\varepsilon(\boldsymbol{w}, b, \boldsymbol{\mu}^i, \boldsymbol{\Sigma}^{\mathbf{i}})$ and the corresponding Mahalanobis distance r^i is given by the integral

$$\varepsilon(\boldsymbol{w}, b, \boldsymbol{\mu}^i, \boldsymbol{\Sigma}^{\mathbf{i}}) = \int_{r^i}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2} dt \, . \tag{2.15}$$

The optimization problem (2.14) can be equivalently rewritten as

$$(\boldsymbol{w}^*, b^*) = \operatorname*{argmax}_{\boldsymbol{w},b} F(\boldsymbol{w}, b) = \operatorname*{argmax}_{\boldsymbol{w},b} \min_{i \in \mathcal{I}_1 \cup \mathcal{I}_2} \frac{\langle \boldsymbol{w} \cdot \boldsymbol{\mu}^i \rangle + b}{\sqrt{\langle \boldsymbol{w} \cdot \boldsymbol{\Sigma}^i \boldsymbol{w} \rangle}} \, .$$

which is more suitable for optimization. The objective function $F(\boldsymbol{w}, b)$ is proven to be convex in the region where the probability of misclassification $\mathrm{Err}(\boldsymbol{w}, b)$ is less than 0.5. However, the objective function $F(\boldsymbol{w}, b)$ is not differentiable.

The STPRtool contains implementations of the algorithm solving the original Anderson's task as well as implementations of three different approaches to solve the Generalized Anderson's task which are described bellow. An interactive demo on the algorithms solving the Generalized Anderson's task is implemented in `demo_anderson`.

References: The original Anderson's task was published in [1]. A detailed description of the Generalized Anderson's task and all the methods implemented in the STPRtool is given in book [26].

2.4.1 Original Anderson's algorithm

The algorithm is implemented in the function `androrig`. This algorithm solves the original Anderson's task defined for two Gaussians only $|\mathcal{I}_1| = |\mathcal{I}_2| = 1$. In this case, the optimal solution vector $\boldsymbol{w} \in \mathbb{R}^n$ is obtained by solving the following problem

$$\begin{aligned} \boldsymbol{w} &= ((1-\lambda)\boldsymbol{\Sigma}_1 + \lambda\boldsymbol{\Sigma}_2)^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \, , \\ \frac{1-\lambda}{\lambda} &= \sqrt{\frac{\langle \boldsymbol{w} \cdot \boldsymbol{\Sigma}_2 \boldsymbol{w} \rangle}{\langle \boldsymbol{w} \cdot \boldsymbol{\Sigma}_1 \boldsymbol{w} \rangle}} \, , \end{aligned}$$

where the scalar $0 < \lambda < 1$ is unknown. The problem is solved by an iterative algorithm which stops while the condition

$$\left| \gamma - \frac{1-\lambda}{\lambda} \right| \le \varepsilon \, , \qquad \gamma = \sqrt{\frac{\langle \boldsymbol{w} \cdot \boldsymbol{\Sigma}_2 \boldsymbol{w} \rangle}{\langle \boldsymbol{w} \cdot \boldsymbol{\Sigma}_1 \boldsymbol{w} \rangle}} \, ,$$

is satisfied. The parameter ε defines the closeness to the optimal solution.

Example: Solving the original Anderson's task

The original Anderson's task is solved for the Riply's data set `riply_trn.mat`. The parameters $(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ and $(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ of the Gaussian distribution of the first and the second class are obtained by the Maximum-Likelihood estimation. The found linear classifier is visualized and the Gaussian distributions (see Figure 2.5). The classifier is validated on the testing set `riply_tst.mat`.

```
  trn = load('riply_trn');            % load training data
  distrib = mlcgmm(data);             % estimate Gaussians
  model = androrig(distrib);          % solve Anderson's task
  figure;
  pandr( model, distrib );            % plot solution
  tst = load('riply_tst');            % load testing data
  ypred = linclass( tst.X, model );   % classify testing data
  cerror( ypred, tst.y )              % evaluate error

ans =

    0.1150
```

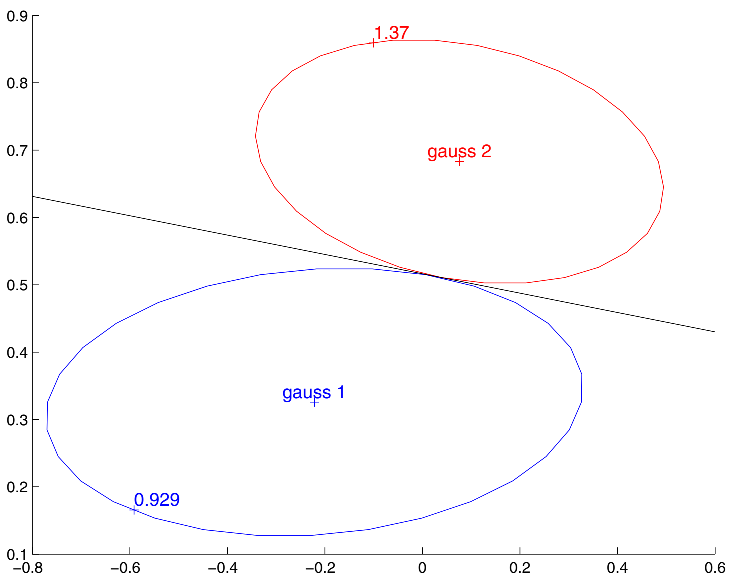


Figure 2.5: Solution of the original Anderson's task.

2.4.2 General algorithm framework

The algorithm is implemented in the function `ganders`. This algorithm repeats the following steps:

I. An improving direction $(\boldsymbol{\Delta w}, \Delta b)$ in the parameter space is computed such that moving along this direction increases the objective function F. The `quadprog` function of the Matlab optimization toolbox is used to solve this subtask.

II. The optimal movement k along the direction $(\boldsymbol{\Delta w}, \Delta b)$. The optimal movement k is determined by $1D$-search based on the cutting interval algorithm according to the Fibonacci series. The number of interval cuttings of is an optional parameter of the algorithm.

III. The new parameters are set

$$\boldsymbol{w}^{(t+1)} := \boldsymbol{w}^{(t)} + k\boldsymbol{\Delta w}\,, \qquad b^{(t+1)} := b^{(t)} + k\Delta b\,.$$

The algorithm iterates until the minimal change in the objective function is less than the prescribed ε, i.e., the condition

$$F(\boldsymbol{w}^{(t+1)}, b^{(t+1)}) - F(\boldsymbol{w}^{(t)}, b^{(t)}) > \varepsilon\,,$$

is violated.

Example: Solving the Generalized Anderson's task

The Generalized Anderson's task is solved for the data `mars.mat`. Both the first and second class are described by three $2D$-dimensional Gaussian distributions. The found linear classifier as well as the input Gaussian distributions are visualized (see Figure 2.6).

```
distrib = load('mars');       % load input Gaussians
model = ganders(distrib);     % solve the GAT
figure;
pandr(model,distrib);         % plot the solution
```

2.4.3 ε-solution using the Kozinec's algorithm

This algorithm is implemented in the function `eanders`. The ε-solution is defined as a linear classifier with parameters $(\boldsymbol{w}, b)$ such that

$$F(\boldsymbol{w}, b) < \varepsilon$$

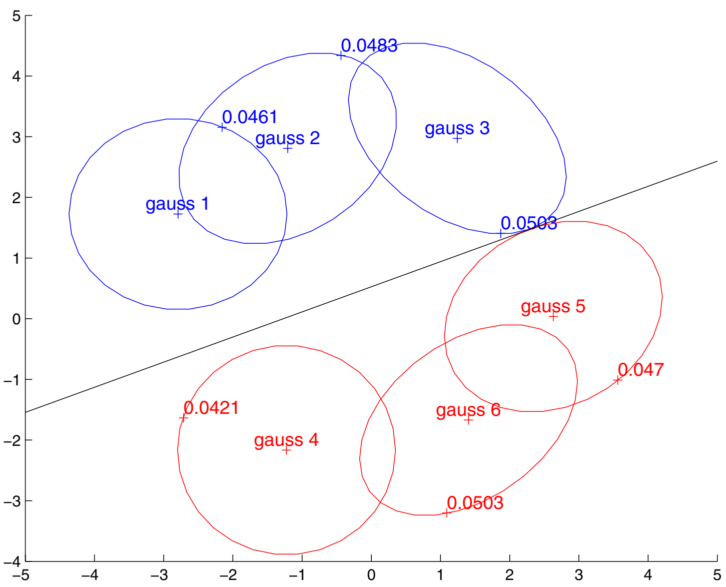


Figure 2.6: Solution of the Generalized Anderson's task.

is satisfied. The prescribed $\varepsilon \in (0, 0.5)$ is the desired upper bound on the probability of misclassification. The problem is transformed to the task of separation of two sets of ellipsoids which is solved the Kozinec's algorithm. The algorithm converges in finite number of iterations if the given ε-solution exists. Otherwise it can get stuck in an endless loop.

Example: ε-solution of the Generalized Anderson's task

The ε-solution of the generalized Anderson's task is solved for the data `mars.mat`. The desired maximal possible probability of misclassification is set to 0.06 (it is 6%). Both the first and second class are described by three $2D$-dimensional Gaussian distributions. The found linear classifier and the input Gaussian distributions are visualized (see Figure 2.7).

```
distrib = load('mars');              % load Gaussians
options = struct('err',0.06');       % maximal desired error
model = eanders(distrib,options);    % training
figure; pandr(model,distrib);        % visualization
```

2.4.4 Generalized gradient optimization

This algorithm is implemented in the function `ggradandr`. The objective function $F(\boldsymbol{w}, b)$ is convex but not differentiable thus the standard gradient optimization cannot

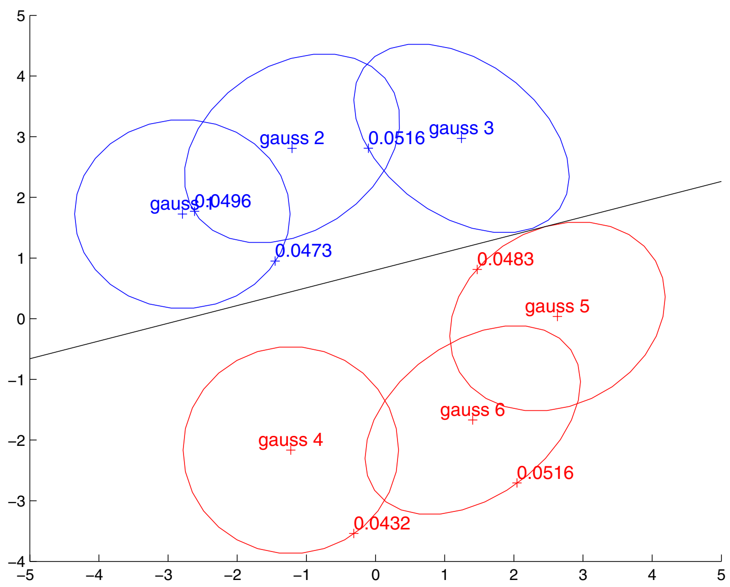


Figure 2.7: ε-solution of the Generalized Anderson's task with maximal 0.06 probability of misclassification.

be used. However, the generalized gradient optimization method can be applied. The algorithm repeats the following two steps:

I. The generalized gradient $(\Delta \boldsymbol{w}, \Delta b)$ is computed.

II. The optimized parameters are updated

$$\boldsymbol{w}^{(t+1)} := \boldsymbol{w}^{(t)} + \Delta \boldsymbol{w} \,, \qquad b^{(t+1)} = b^{(t)} + \Delta b \,.$$

The algorithm iterates until the minimal change in the objective function is less than the prescribed ε, i.e., the condition

$$F(\boldsymbol{w}^{(t+1)}, b^{(t+1)}) - F(\boldsymbol{w}^{(t)}, b^{(t)}) > \varepsilon \,,$$

is violated. The maximal number of iterations can be used as the stopping condition as well.

Example: Solving the Generalized Anderson's task using the generalized gradient optimization

The generalized Anderson's task is solved for the data `mars.mat`. Both the first and second class are described by three $2D$-dimensional Gaussian distributions. The maximal number of iterations is limited to `tmax=10000`. The found linear classifier as well as the input Gaussian distributions are visualized (see Figure 2.8).

```
distrib = load('mars');                 % load Gaussians
options = struct('tmax',10000);         % max # of iterations
model = ggradandr(distrib,options); % training
figure;
pandr( model, distrib );                % visualization
```

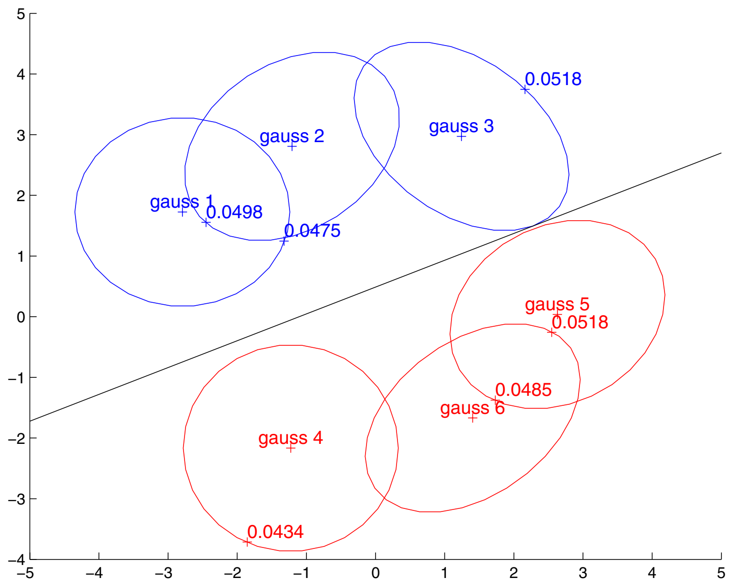


Figure 2.8: Solution of the Generalized Anderson's task after 10000 iterations of the generalized gradient optimization algorithm.

Chapter 3

Feature extraction

The feature extraction step consists of mapping the input vector of observations $\boldsymbol{x} \in \mathbb{R}^n$ onto a new feature description $\boldsymbol{z} \in \mathbb{R}^m$ which is more suitable for given task. The linear projection

$$\boldsymbol{z} = \mathbf{W}^T \boldsymbol{x} + \boldsymbol{b}\,, \tag{3.1}$$

is commonly used. The projection is given by the matrix $\mathbf{W}$ $[n \times m]$ and bias vector $\boldsymbol{b} \in \mathbb{R}^m$. The Principal Component Analysis (PCA) (Section 3.1) is a representative of the unsupervised learning method which yields the linear projection (3.1). The Linear Discriminant Analysis (LDA) (Section 3.2) is a representative of the supervised learning method yielding the linear projection. The linear data projection is implemented in function `linproj` (See examples in Section 3.1 and Section 3.2). The data type used to describe the linear data projection is defined in Table 3.

The quadratic data mapping projects the input vector $\boldsymbol{x} \in \mathbb{R}^n$ onto the feature space $\mathbb{R}^m$ where $m = n(n+3)/2$. The quadratic mapping $\boldsymbol{\phi}\colon \mathbb{R}^n \to \mathbb{R}^m$ is defined as

$$\begin{array}{rclcccc}
\boldsymbol{\phi}(\boldsymbol{x}) & = & [& x_1, & x_2, & \ldots, & x_n, \\
 & & & x_1x_1, & x_1x_2, & \ldots, & x_1x_n, \\
 & & & & x_2x_2, & \ldots, & x_2x_n, \\
 & & & & & \vdots & \\
 & & & & & & x_nx_n\,]^T\,,
\end{array} \tag{3.2}$$

where x_i, $i = 1, \ldots, n$ are entries of the vector $\boldsymbol{x} \in \mathbb{R}^n$. The quadratic data mapping is implemented in function `qmap`.

The kernel functions allow for non-linear extensions of the linear feature extraction methods. In this case, the input vectors $\boldsymbol{x} \in \mathbb{R}^n$ are mapped into a new higher dimensional feature space $\mathcal{F}$ in which the linear methods are applied. This yields a non-linear (kernel) projection of data which has generally defined as

$$\boldsymbol{z} = \mathbf{A}^T \boldsymbol{k}(\boldsymbol{x}) + \boldsymbol{b}\,, \tag{3.3}$$

where $\mathbf{A}$ $[l \times m]$ is a parameter matrix and $\boldsymbol{b} \in \mathbb{R}^m$ a parameter vector. The vector $\boldsymbol{k}(\boldsymbol{x}) = [k(\boldsymbol{x}, \boldsymbol{x}_1), \ldots, k(\boldsymbol{x}, \boldsymbol{x}_l)]^T$ is a vector of kernel functions centered in the training data or their subset. The $\boldsymbol{x} \in \mathbb{R}^n$ stands for the input vector and $\boldsymbol{z} \in \mathbb{R}^m$ is the vector of extracted features. The extracted features are projections of $\boldsymbol{\phi}(\boldsymbol{x})$ onto m vectors (or functions) from the feature space $\mathcal{F}$. The Kernel PCA (Section 3.3) and the Generalized Discriminant Analysis (GDA) (Section 3.5) are representatives of the feature extraction methods yielding the kernel data projection (3.3). The kernel data projection is implemented in the function `kernelproj` (See examples in Section 3.3 and Section 3.5). The data type used to describe the kernel data projection is defined in Table 3. The summary of implemented methods for feature extraction is given in Table 3.1

Table 3.1: Implemented methods for feature extraction

`linproj`	Linear data projection.
`kerproj`	Kernel data projection.
`qmap`	Quadratic data mapping.
`lin2quad`	Merges linear rule and quadratic mapping.
`lin2svm`	Merges linear rule and kernel projection.
`lda`	Linear Discriminant Analysis.
`pca`	Principal Component Analysis.
`pcarec`	Computes reconstructed vector after PCA projection.
`gda`	Generalized Discriminant Analysis.
`greedykpca`	Greedy Kernel Principal Component Analysis.
`kpca`	Kernel Principal Component Analysis.
`kpcarec`	Reconstructs image after kernel PCA.
`demo_pcacomp`	Demo on image compression using PCA.

Table 3.2: Data-type used to describe linear data projection.

Linear data projection (structure array):	
`.W` $[n \times m]$	Projection matrix $\mathbf{W} = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m]$.
`.b` $[m \times 1]$	Bias vector b.
`.fun = 'linproj'`	Identifies function associated with this data type.

3.1 Principal Component Analysis

Let $\mathcal{T}_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ be a set of training vectors from the n-dimensional input space $\mathbb{R}^n$. The set of vectors $\mathcal{T}_Z = \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_l\}$ is a lower dimensional representation of the

Table 3.3: Data-type used to describe kernel data projection.

Kernel data projection (structure array):	
`.Alpha` $[l \times m]$	Parameter matrix $\mathbf{A}$ $[l \times m]$.
`.b` $[m \times 1]$	Bias vector b.
`.sv.X` $[n \times l]$	Training vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$.
`.options.ker` $[string]$	Kernel identifier.
`.options.arg` $[1 \times p]$	Kernel argument(s).
`.fun = 'kernelproj'`	Identifies function associated with this data type.

input training vectors $\mathcal{T}_X$ in the m-dimensional space $\mathbb{R}^m$. The vectors $\mathcal{T}_Z$ are obtained by the linear orthonormal projection

$$\boldsymbol{z} = \mathbf{W}^T \boldsymbol{x} + \boldsymbol{b} \,, \tag{3.4}$$

where the matrix $\mathbf{W}$ $[n \times m]$ and the vector $\boldsymbol{b}$ $[m \times 1]$ are parameters of the projection. The reconstructed vectors $\mathcal{T}_{\tilde{X}} = \{\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_l\}$ are computed by the linear back projection

$$\tilde{\boldsymbol{x}} = \mathbf{W}(\boldsymbol{z} - \boldsymbol{b}) \,, \tag{3.5}$$

obtained by inverting (3.4). The mean square reconstruction error

$$\varepsilon_{MS}(\mathbf{W}, \boldsymbol{b}) = \frac{1}{l} \sum_{i=1}^{l} \|\boldsymbol{x}_i - \tilde{\boldsymbol{x}}_i\|^2 \,, \tag{3.6}$$

is a function of the parameters of the linear projections (3.4) and (3.5). The Principal Component Analysis (PCA) is the linear orthonormal projection (3.4) which allows for the minimal mean square reconstruction error (3.6) of the training data $\mathcal{T}_X$. The parameters $(\mathbf{W}, \boldsymbol{b})$ of the linear projection are the solution of the optimization task

$$(\mathbf{W}, \boldsymbol{b}) = \operatorname*{argmin}_{\mathbf{W}', \boldsymbol{b}'} \varepsilon_{MS}(\mathbf{W}', \boldsymbol{b}') \,, \tag{3.7}$$

subject to

$$\langle \boldsymbol{w}_i \cdot \boldsymbol{w}_j \rangle = \delta(i,j) \,, \qquad \forall i, j \,,$$

where $\boldsymbol{w}_i$, $i = 1, \ldots, m$ are column vectors of the matrix $\mathbf{W} = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m]$ and $\delta(i,j)$ is the Kronecker delta function. The solution of the task (3.7) is the matrix $\mathbf{W} = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m]$ containing the m eigenvectors of the sample covariance matrix which have the largest eigen values. The vector $\boldsymbol{b}$ equals to $\mathbf{W}^T \boldsymbol{\mu}$, where μ is the sample mean of the training data. The PCA is implemented in the function `pca`. A demo showing the use of PCA for image compression is implemented in the script `demo_pcacomp`.

References: The PCA is treated throughout for instance in books [13, 3, 6].

Example: Principal Component Analysis

The input data are synthetically generated from the 2-dimensional Gaussian distribution. The PCA is applied to train the 1-dimensional subspace to approximate the input data with the minimal reconstruction error. The training data, the reconstructed data and the 1-dimensional subspace given by the first principal component are visualized (see Figure 3.1).

```
X = gsamp([5;5],[1 0.8;0.8 1],100);  % generate data
model = pca(X,1);                     % train PCA
Z = linproj(X,model);                 % lower dim. proj.
XR = pcarec(X,model);                 % reconstr. data

figure; hold on; axis equal;          % visualization
h1 = ppatterns(X,'kx');
h2 = ppatterns(XR,'bo');
[dummy,mn] = min(Z);
[dummy,mx] = max(Z);
h3 = plot([XR(1,mn) XR(1,mx)],[XR(2,mn) XR(2,mx)],'r');
legend([h1 h2 h3], ...
 'Input vectors','Reconstructed', 'PCA subspace');
```

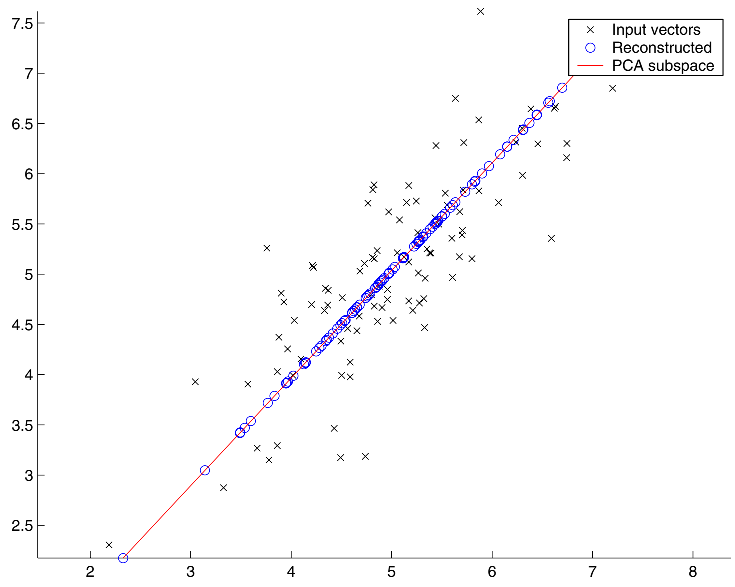


Figure 3.1: Example on the Principal Component Analysis.

3.2 Linear Discriminant Analysis

Let $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$, $\boldsymbol{x}_i \in \mathbb{R}^n$, $y \in \mathcal{Y} = \{1, 2, \ldots, c\}$ be labeled set of training vectors. The within-class $\mathbf{S_W}$ and between-class $\mathbf{S_B}$ scatter matrices are described as

$$\begin{aligned} \mathbf{S_W} &= \sum_{y \in \mathcal{Y}} \mathbf{S}_y \,, \qquad \mathbf{S}_y = \sum_{i \in \mathcal{I}_y} (\boldsymbol{x}_i - \boldsymbol{\mu}_i)(\boldsymbol{x}_i - \boldsymbol{\mu}_i)^T \,, \\ \mathbf{S_B} &= \sum_{y \in \mathcal{Y}} |\mathcal{I}_y| (\boldsymbol{\mu}_y - \boldsymbol{\mu})(\boldsymbol{\mu}_y - \boldsymbol{\mu})^T \,, \end{aligned} \tag{3.8}$$

where the total mean vector $\boldsymbol{\mu}$ and the class mean vectors $\boldsymbol{\mu}_y$, $y \in \mathcal{Y}$ are defined as

$$\boldsymbol{\mu} = \frac{1}{l} \sum_{i=1}^{l} \boldsymbol{x}_i \,, \qquad \boldsymbol{\mu}_y = \frac{1}{|\mathcal{I}_y|} \sum_{i \in \mathcal{I}_y} \boldsymbol{x}_i \,, y \in \mathcal{Y} \,.$$

The goal of the Linear Discriminant Analysis (LDA) is to train the linear data projection

$$\boldsymbol{z} = \mathbf{W}^T \boldsymbol{x} \,,$$

such that the class separability criterion

$$F(\mathbf{W}) = \frac{\det(\tilde{\mathbf{S}}_\mathbf{B})}{\det(\tilde{\mathbf{S}}_\mathbf{W})} = \frac{\det(\mathbf{S}_B)}{\det(\mathbf{S}_W)} \,,$$

is maximized. The $\tilde{\mathbf{S}}_\mathbf{B}$ is the between-class and $\tilde{\mathbf{S}}_\mathbf{W}$ is the within-class scatter matrix of projected data which is defined similarly to (3.8).
References: The LDA is treated for instance in books [3, 6].

Example 1: Linear Discriminant Analysis
The LDA is applied to extract 2 features from the Iris data set `iris.mat` which consists of the labeled 4-dimensional data. The data after the feature extraction step are visualized in Figure 3.2.

```
orig_data = load('iris');            % load input data
model = lda(orig_data,2);            % train LDA
ext_data = linproj(orig_data,model); % feature extraction
figure; ppatterns(ext_data);         % plot ext. data
```

Example 2: PCA versus LDA
This example shows why the LDA is superior to the PCA when features good for classification are to be extracted. The synthetical data are generated from the Gaussian mixture model. The LDA and PCA are trained on the generated data. The LDA and

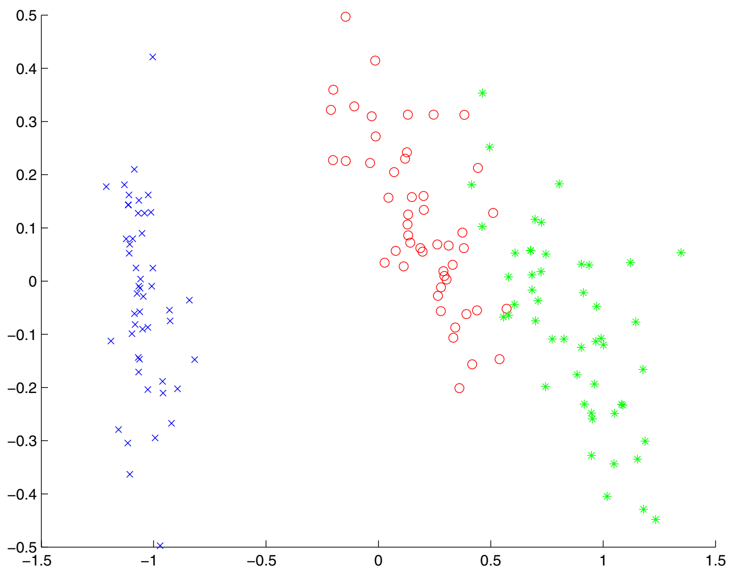


Figure 3.2: Example shows 2-dimensional data extracted from the originally 4-dimensional Iris data set using Linear Discriminant Analysis.

PCA directions are displayed as connection of the vectors projected to the trained LDA and PCA subspaces and re-projected back to the input space (see Figure 3.3a). The extracted data using the LDA and PCA model are displayed with the Gaussians fitted by the Maximum-Likelihood (see Figure 3.3b).

```
% Generate data
distrib.Mean = [[5;4] [4;5]];          % mean vectors
distrib.Cov(:,:,1) = [1 0.9; 0.9 1];   % 1st covariance
distrib.Cov(:,:,2) = [1 0.9; 0.9 1];   % 2nd covariance
distrib.Prior = [0.5 0.5];             % Gaussian weights
data = gmmsamp(distrib,250);           % sample data

lda_model = lda(data,1);               % train LDA
lda_rec = pcarec(data.X,lda_model);
lda_data = linproj(data,lda_model);

pca_model = pca(data.X,1);             % train PCA
pca_rec = pcarec(data.X,pca_model);
pca_data = linproj( data,pca_model);

figure; hold on; axis equal;           % visualization
ppatterns(data);
```

```
h1 = plot(lda_rec(1,:),lda_rec(2,:),'r');
h2 = plot(pca_rec(1,:),pca_rec(2,:),'b');
legend([h1 h2],'LDA direction','PCA direction');

figure; hold on;
subplot(2,1,1); title('LDA'); ppatterns(lda_data);
pgauss(mlcgmm(lda_data));
subplot(2,1,2); title('PCA'); ppatterns(pca_data);
pgauss(mlcgmm(pca_data));
```

3.3 Kernel Principal Component Analysis

The Kernel Principal Component Analysis (Kernel PCA) is the non-linear extension of the ordinary linear PCA. The input training vectors $\mathcal{T}_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$, $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ are mapped by $\boldsymbol{\phi}\colon \mathcal{X} \to \mathcal{F}$ to a high dimensional feature space $\mathcal{F}$. The linear PCA is applied on the mapped data $\mathcal{T}_\Phi = \{\boldsymbol{\phi}(\boldsymbol{x}_1), \ldots, \boldsymbol{\phi}(\boldsymbol{x}_l)\}$. The computation of the principal components and the projection on these components can be expressed in terms of dot products thus the kernel functions $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ can be employed. The kernel PCA trains the kernel data projection

$$\boldsymbol{z} = \mathbf{A}^T \boldsymbol{k}(\boldsymbol{x}) + \boldsymbol{b}\,, \tag{3.9}$$

such that the reconstruction error

$$\varepsilon_{KMS}(\mathbf{A}, \boldsymbol{b}) = \frac{1}{l} \sum_{i=1}^{l} \|\boldsymbol{\phi}(\boldsymbol{x}_i) - \tilde{\boldsymbol{\phi}}(\boldsymbol{x}_i)\|^2\,, \tag{3.10}$$

is minimized. The reconstructed vector $\tilde{\boldsymbol{\phi}}(\boldsymbol{x})$ is given as a linear combination of the mapped data $\mathcal{T}_\Phi$

$$\tilde{\boldsymbol{\phi}}(\boldsymbol{x}) = \sum_{i=1}^{l} \beta_i \boldsymbol{\phi}(\boldsymbol{x}_i)\,, \qquad \boldsymbol{\beta} = \mathbf{A}(\boldsymbol{z} - \boldsymbol{b})\,. \tag{3.11}$$

In contrast to the linear PCA, the explicit projection from the feature space $\mathcal{F}$ to the input space $\mathcal{X}$ usually do not exist. The problem is to find the vector $\boldsymbol{x} \in \mathcal{X}$ its image $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathcal{F}$ well approximates the reconstructed vector $\tilde{\boldsymbol{\phi}}(\boldsymbol{x}) \in \mathcal{F}$. This procedure is implemented in the function `kpcarec` for the case of the Radial Basis Function (RBF) kernel. The procedure consists of the following step:

I. Project input vector $\boldsymbol{x}_{in} \in \mathbb{R}^n$ onto its lower dimensional representation $\boldsymbol{z} \in \mathbb{R}^m$ using (3.9).

II. Computes vector $\boldsymbol{x}_{out} \in \mathbb{R}^n$ which is good pre-image of the reconstructed vector $\tilde{\boldsymbol{\phi}}(\boldsymbol{x}_{in})$, such that

$$\boldsymbol{x}_{out} = \underset{\boldsymbol{x}}{\operatorname{argmin}} \|\boldsymbol{\phi}(\boldsymbol{x}) - \tilde{\boldsymbol{\phi}}(\boldsymbol{x}_{in})\|^2 \,.$$

References: The kernel PCA is described at length in book [29, 30].

Example: Kernel Principal Component Analysis

The example shows using the kernel PCA for data denoising. The input data are synthetically generated 2-dimensional vectors which lie on a circle and are corrupted by the Gaussian noise. The kernel PCA model with RBF kernel is trained. The function `kpcarec` is used to find pre-images of the reconstructed data (3.11). The result is visualized in Figure 3.4.

```
X = gencircledata([1;1],5,250,1); % generate circle data
options.ker = 'rbf';               % use RBF kernel
options.arg = 4;                   % kernel argument
options.new_dim = 2;               % output dimension
model = kpca(X,options);           % compute kernel PCA
XR = kpcarec(X,model);             % compute reconstruced data
figure;                            % Visualization
h1 = ppatterns(X);
h2 = ppatterns(XR, '+r');
legend([h1 h2],'Input vectors','Reconstructed');
```

3.4 Greedy kernel PCA algorithm

The Greedy Kernel Principal Analysis (Greedy kernel PCA) is an efficient algorithm to compute the ordinary kernel PCA. Let $\mathcal{T}_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$, $\boldsymbol{x}_i \in \mathbb{R}^n$ be the set of input training vectors. The goal is to train the kernel data projection

$$\boldsymbol{z} = \mathbf{A}^T \boldsymbol{k}_S(\boldsymbol{x}) + \boldsymbol{b} \,, \tag{3.12}$$

where $\mathbf{A}$ $[d \times m]$ is the parameter matrix, $\boldsymbol{b}$ $[m \times 1]$ is the parameter vector and $\boldsymbol{k}_S = [k(\boldsymbol{x}, \boldsymbol{s}_1), \ldots, k(\boldsymbol{x}, \boldsymbol{s}_l)]^T$ are the kernel functions centered in the vectors $\mathcal{T}_S = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_d\}$. The vector set $\mathcal{T}_S$ is a subset of training data $\mathcal{T}_X$. In contrast to the ordinary kernel PCA, the subset $\mathcal{T}_S$ does not contain all the training vectors $\mathcal{T}_X$ thus the complexity of the projection (3.12) is reduced compared to (3.9). The objective of the Greedy kernel PCA is to minimize the reconstruction error (3.11) while the size d of the subset $\mathcal{T}_S$ is kept small. The Greedy kernel PCA algorithm is implemented in the function `greedykpca`.

References: The implemented Greedy kernel PCA algorithm is described in [11].

Example: Greedy Kernel Principal Component Analysis

The example shows using kernel PCA for data denoising. The input data are synthetically generated 2-dimensional vectors which lie on a circle and are corrupted with the Gaussian noise. The Greedy kernel PCA model with RBF kernel is trained. The number of vectors defining the kernel projection (3.12) is limited to 25. In contrast the ordinary kernel PCA (see experiment in Section 3.3) requires all the 250 training vectors. The function `kpcarec` is used to find pre-images of the reconstructed data obtained by the kernel PCA projection. The training and reconstructed vectors are visualized in Figure 3.5. The vector of the selected subset $\mathcal{T}_X$ are visualized as well.

```
X = gencircledata([1;1],5,250,1); % generate training data

options.ker = 'rbf';               % use RBF kernel
options.arg = 4;                   % kernel argument
options.new_dim = 2;               % output dimension
options.m = 25;                    % size of sel. subset
model = greedykpca(X,options);     % run greedy algorithm

XR = kpcarec(X,model);             % reconstructed vectors
figure;                            % visualization
h1 = ppatterns(X);
h2 = ppatterns(XR,'+r');
h3 = ppatterns(model.sv.X,'ob',12);
legend([h1 h2 h3], ...
 'Training set','Reconstructed set','Selected subset');
```

3.5 Generalized Discriminant Analysis

The Generalized Discriminant Analysis (GDA) is the non-linear extension of the ordinary Linear Discriminant Analysis (LDA). The input training data $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$, $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$, $y \in \mathcal{Y} = \{1, \ldots, c\}$ are mapped by $\boldsymbol{\phi}\colon \mathcal{X} \to \mathcal{F}$ to a high dimensional feature space $\mathcal{F}$. The ordinary LDA is applied on the mapped data $\mathcal{T}_{\Phi Y} = \{(\boldsymbol{\phi}(\boldsymbol{x}_1), y_1), \ldots, (\boldsymbol{\phi}(\boldsymbol{x}_l), y_l)\}$. The computation of the projection vectors and the projection on these vectors can be expressed in terms of dot products thus the kernel functions $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ can be employed. The resulting kernel data projection is defined as

$$\boldsymbol{z} = \mathbf{A}^T \boldsymbol{k}(\boldsymbol{x}) + \boldsymbol{b}\,,$$

where $\mathbf{A}$ $[l \times m]$ is the matrix and $\boldsymbol{b}$ $[m \times 1]$ the vector trained by the GDA. The parameters $(\mathbf{A}, \boldsymbol{b})$ are trained to increase the between-class scatter and decrease the within-class scatter of the extracted data $\mathcal{T}_{ZY} = \{(\boldsymbol{z}_1, y_1), \dots, (\boldsymbol{z}_l, y_l)\}$, $\boldsymbol{z}_i \in \mathbb{R}^m$. The GDA is implemented in the function `gda`.
References: The GDA was published in the paper [2].

Example: Generalized Discriminant Analysis

The GDA is trained on the Iris data set `iris.mat` which contains 3 classes of 4-dimensional data. The RBF kernel with kernel width $\sigma = 1$ is used. Notice, that setting σ too low leads to the perfect separability of the training data $\mathcal{T}_{XY}$ but the kernel projection is heavily over-fitted. The extracted data $\mathcal{Y}_{ZY}$ are visualized in Figure 3.6.

```
orig_data = load('iris');                 % load data
options.ker = 'rbf';                      % use RBF kernel
options.arg = 1;                          % kernel arg.
options.new_dim = 2;                      % output dimension
model = gda(orig_data,options);           % train GDA
ext_data = kernelproj(orig_data,model);   % feature ext.
figure; ppatterns(ext_data);              % visualization
```

3.6 Combining feature extraction and linear classifier

Let $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_l, y_l)\}$ be a training set of observations $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ and corresponding hidden states $y_i \in \mathcal{Y} = \{1, \dots, c\}$. Let $\boldsymbol{\phi}\colon \mathcal{X} \to \mathbb{R}^m$ be a mapping

$$\boldsymbol{\phi}(\boldsymbol{x}) = [\phi_1(\boldsymbol{x}), \dots, \phi_m(\boldsymbol{x})]^T$$

defining a non-linear feature extraction step (e.g., quadratic data mapping or kernel projection). The feature extraction step applied to the input data $\mathcal{T}_{XY}$ yields extracted data $\mathcal{T}_{\Phi Y} = \{(\boldsymbol{\phi}(\boldsymbol{x}_1), y_1), \dots, (\boldsymbol{\phi}(\boldsymbol{x}_l), y_l)\}$. Let $f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}) \rangle + b$ be a linear discriminant function found by an arbitrary training method on the extracted data $\mathcal{T}_{\Phi Y}$. The function f can be seen as a non-linear function of the input vectors $\boldsymbol{x} \in \mathcal{X}$, as

$$f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{\phi}(\boldsymbol{x}) \rangle + b = \sum_{i=1}^{m} w_i \phi_i(\boldsymbol{x}) + b \,.$$

In the case of the quadratic data mapping (3.2) used as the feature extraction step, the discriminant function can be written in the following form

$$f(\boldsymbol{x}) = \langle \boldsymbol{x} \cdot \mathbf{A}\boldsymbol{x} \rangle + \langle \boldsymbol{x} \cdot \boldsymbol{b} \rangle + c \,. \tag{3.13}$$

The discriminant function (3.13) defines the quadratic classifier described in Section 6.4. Therefore the quadratic classifier can be trained by (i) applying the quadratic data mapping `qmap`, (ii) training a linear rule on the mapped data (e.g., `perceptron`, `fld`) and (iii) combining the quadratic mapping and the linear rule. The combining is implement in function `lin2quad`.

In the case of the kernel projection (3.3) applied as the feature extraction step, the resulting discriminant function has the following form

$$f(\boldsymbol{x}) = \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}(\boldsymbol{x}) \rangle + b \,. \tag{3.14}$$

The discriminant function (3.14) defines the kernel (SVM type) classifier described in Section 5. This classifier can be trained by (i) applying the kernel projection `kernelproj`, (ii) training a linear rule on the extracted data and (iii) combing the kernel projection and the linear rule. The combining is implemented in function `lin2svm`.

Example: Quadratic classifier trained the Perceptron

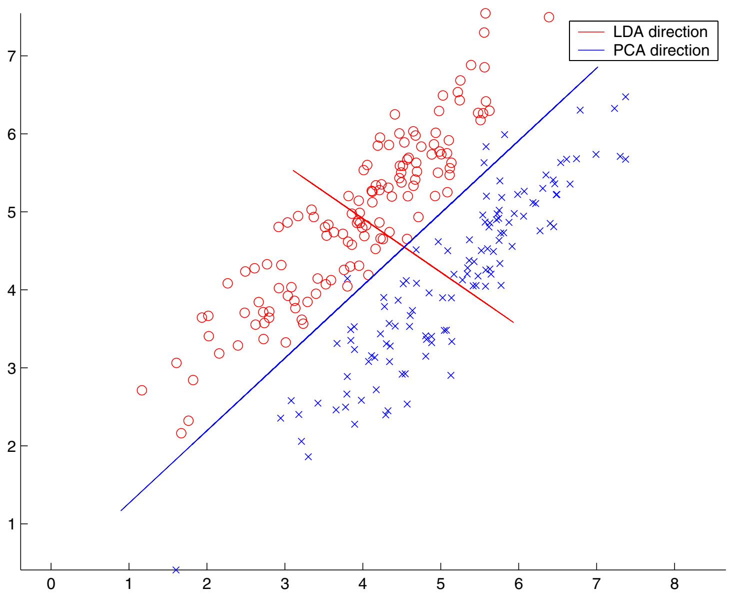
This example shows how to train the quadratic classifier using the Perceptron algorithm. The Perceptron algorithm produces the linear rule only. The input training data are linearly non-separable but become separable after the quadratic data mapping is applied. Function `lin2quad` is used to compute the parameters of the quadratic classifier explicitly based on the found linear rule in the feature space and the mapping. Figure 3.7 shows the decision boundary of the quadratic classifier.

```
% load training data living in the input space
input_data = load('vltava');

% map data to the feature space
map_data = qmap(input_data);

% train linear rule in the feature sapce
lin_model = perceptron(map_data);

% compute parameters of the quadratic classifier
quad_model = lin2quad(lin_model);

% visualize the quadratic classifier
figure; ppatterns(input_data);
pboundary(quad_model);
```

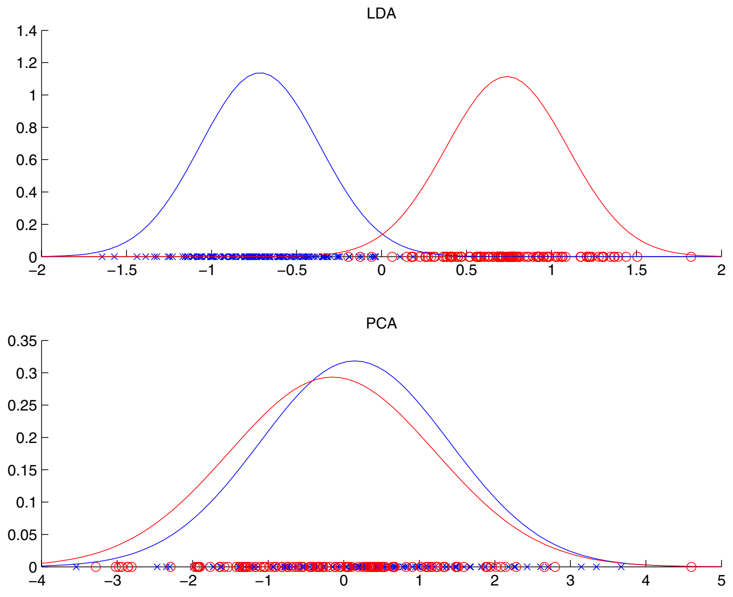
Example: Kernel classifier from the linear rule

This example shows how to train the kernel classifier using arbitrary training algorithm for linear rule. The Fisher Linear Discriminant was used in this example. The greedy kernel PCA algorithm was applied as the feature extraction step. The Riply's data set was used for training and evaluation. Figure 3.8 shows the found kernel classifier, the training data and vectors used in the kernel expansion of the discriminant function.

```
   % load input data
   trn = load('riply_trn');

   % train kernel PCA by greedy algorithm
   options = struct('ker','rbf','arg',1,'new_dim',10);
   kpca_model = greedykpca(trn.X,options);

   % project data
   map_trn = kernelproj(trn,kpca_model);

   % train linear classifier
   lin_model = fld(map_trn);

   % combine linear rule and kernel PCA
   kfd_model = lin2svm(kpca_model,lin_model);

   % visualization
   figure;
   ppatterns(trn); pboundary(kfd_model);
   ppatterns(kfd_model.sv.X,'ok',13);

   % evaluation on testing data
   tst = load('riply_tst');
   ypred = svmclass(tst.X,kfd_model);
   cerror(ypred,tst.y)

ans =

    0.0970
```



(a)



(b)

Figure 3.3: Comparison between PCA and LDA. Figure (a) shows input data and found LDA (red) and PCA (blue) directions onto which the data are projected. Figure (b) shows the class conditional Gaussians estimated from data projected data onto LDA and PCA directions.

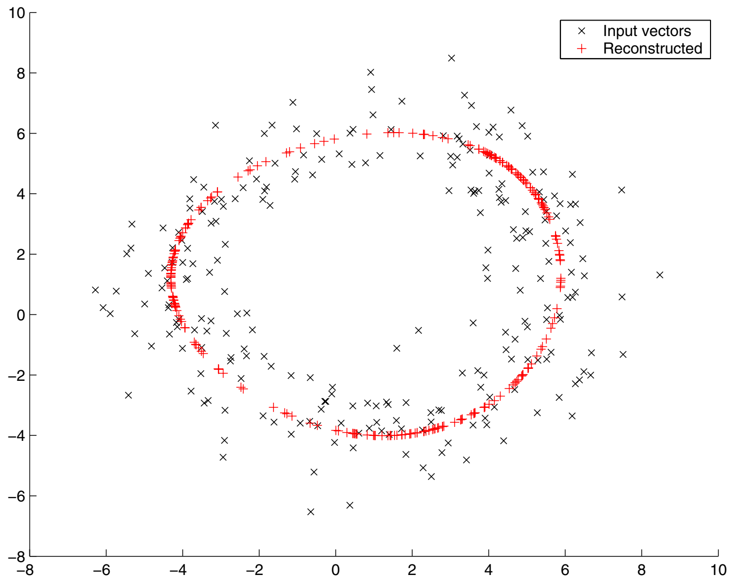


Figure 3.4: Example on the Kernel Principal Component Analysis.

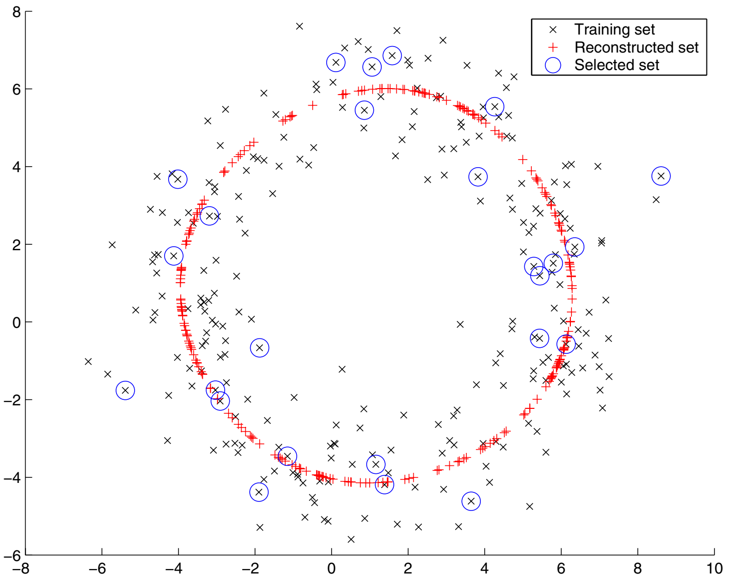


Figure 3.5: Example on the Greedy Kernel Principal Component Analysis.

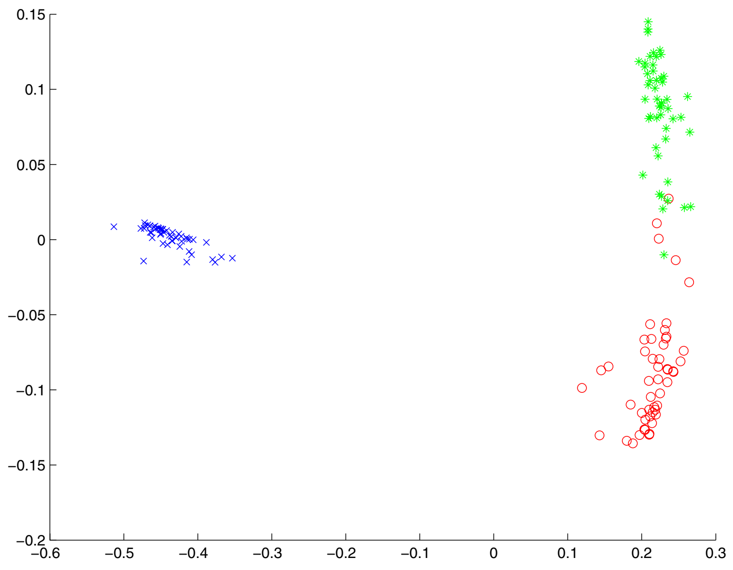


Figure 3.6: Example shows 2-dimensional data extracted from the originally 4-dimensional Iris data set using the Generalized Discriminant Analysis. In contrast to LDA (see Figure 3.2 for comparison), the class clusters are more compact but at the expense of a higher risk of over-fitting.

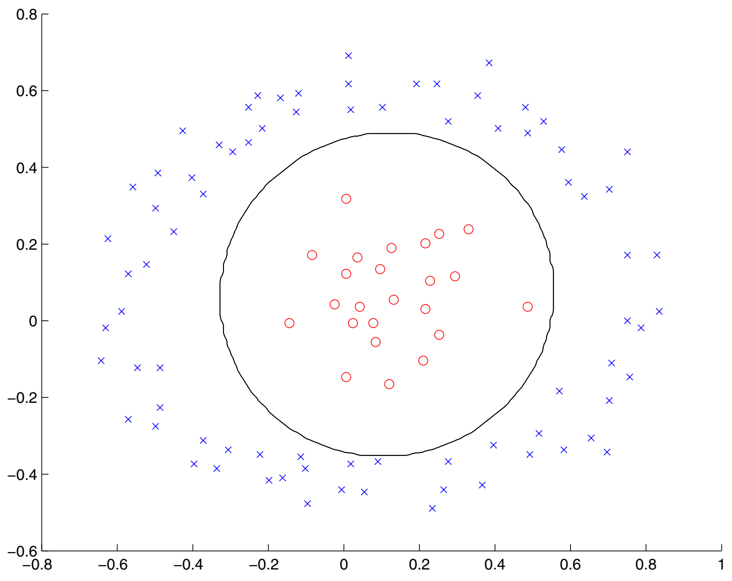


Figure 3.7: Example shows quadratic classifier found by the Perceptron algorithm on the data mapped to the feature by the quadratic mapping.

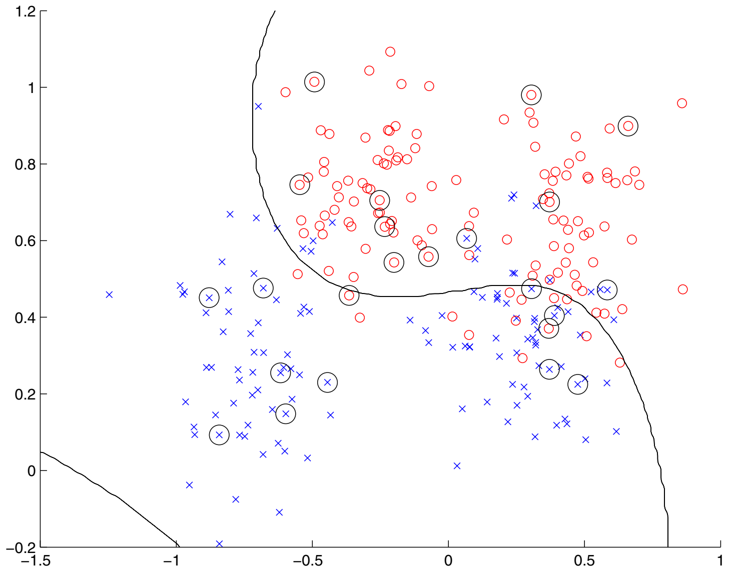


Figure 3.8: Example shows the kernel classifier found by the Fisher Linear Discriminant algorithm on the data mapped to the feature by the kernel PCA. The greedy kernel PCA was used therefore only a subset of training data was used to determine the kernel projection.

Chapter 4

Density estimation and clustering

The STPRtool represents a probability distribution function $P_X(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{R}^n$ as a structure array which must contain field `.fun` (see Table 4.2). The field `.fun` specifies function which is used to evaluate $P_X(\boldsymbol{x})$ for given set of realization $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ of a random vector. Let the structure `model` represent a given distribution function. The Matlab command

```
  y = feval(X,model.fun)
```

is used to evaluate $y_i = P_X(\boldsymbol{x}_i)$. The matrix `X` $[n \times l]$ contains column vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$. The vector `y` $[1 \times l]$ contains values of the distribution function. In particular, the Gaussian distribution and the Gaussian mixture models are described in Section 4.1. The summary of implemented methods for dealing with probability density estimation and clustering is given in Table 4.1

4.1 Gaussian distribution and Gaussian mixture model

The value of the multivariate Gaussian probability distribution function in the vector $\boldsymbol{x} \in \mathbb{R}^n$ is given by

$$P_X(\boldsymbol{x}) = \frac{1}{(2\pi)^{\frac{n}{2}}\sqrt{\det(\Sigma)}} \exp(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu}))\,, \tag{4.1}$$

where $\boldsymbol{\mu}$ $[n \times 1]$ is the mean vector and $\boldsymbol{\Sigma}$ $[n \times n]$ is the covariance matrix. The STPRtool uses specific data-type to describe a set of Gaussians which can be labeled, i.e., each pair $(\boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ is assigned to the class $y_i \in \mathcal{Y}$. The covariance matrix $\boldsymbol{\Sigma}$ is always represented as the square matrix $[n \times n]$, however, the shape of the matrix can be indicated in the optimal field `.cov_type` (see Table 4.3). The data-type is described in Table 4.4. The evaluation of the Gaussian distribution is implemented in the function

Table 4.1: Implemented methods: Density estimation and clustering

`gmmsamp`	Generates sample from Gaussian mixture model.
`gsamp`	Generates sample from Gaussian distribution.
`kmeans`	K-means clustering algorithm.
`pdfgauss`	Evaluates for multivariate Gaussian distribution.
`pdfgauss`	Evaluates Gaussian mixture model.
`sigmoid`	Evaluates sigmoid function.
`emgmm`	Expectation-Maximization Algorithm for Gaussian mixture model.
`melgmm`	Maximizes Expectation of Log-Likelihood for Gaussian mixture.
`mlcgmm`	Maximal Likelihood estimation of Gaussian mixture model.
`mlsigmoid`	Fitting a sigmoid function using ML estimation.
`mmgauss`	Minimax estimation of Gaussian distribution.
`rsde`	Reduced Set Density Estimator.
`demo_emgmm`	Demo on Expectation-Maximization (EM) algorithm.
`demo_mmgauss`	Demo on minimax estimation for Gaussian.
`demo_svmpout`	Fitting a posteriori probability to SVM output.

Table 4.2: Data-type used to represent probability distribution function.

Probability Distribution Function (structure array):	
`.fun` [*string*]	The name of a function which evaluates probability distribution `y = feval( X, model.fun )`.

`pdfgauss`. The random sampling from the Gaussian distribution is implemented in the function `gsamp`.

The Gaussian Mixture Model (GMM) is weighted sum of the Gaussian distributions

$$P_X(\boldsymbol{x}) = \sum_{y \in \mathcal{Y}} P_Y(y) P_{X|Y}(\boldsymbol{x}|y) \,, \tag{4.2}$$

where $P_Y(y)$, $y \in \mathcal{Y} = \{1, \ldots, c\}$ are weights and $P_{X|Y}(\boldsymbol{x}|y)$ is the Gaussian distribution (4.1) given by parameters $(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)$. The data-type used to represent the GMM is described in Table 4.5. The evaluation of the GMM is implemented in the function `pdfgmm`. The random sampling from the GMM is implemented in the function `gmmsamp`.

Example: Sampling from the Gaussian distribution

The example shows how to represent the Gaussian distribution and how to sample data it. The sampling from univariate and bivariate Gaussians is shown. The distribution function of the univariate Gaussian is visualized and compared to the histogram

Table 4.3: Shape of covariance matrix.

Parameter `cov_type`*:*	
`'full'`	Full covariance matrix.
`'diag'`	Diagonal covariance matrix.
`'spherical'`	Spherical covariance matrix.

Table 4.4: Data-type used to represent the labeled set of Gaussians.

Labeled set of Gaussians (structure array):	
`.Mean` $[n \times k]$	Mean vectors $\{\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k\}$.
`.Cov` $[n \times n \times k]$	Covariance matrices $\{\boldsymbol{\Sigma}_1, \dots, \boldsymbol{\Sigma}_k\}$.
`.y` $[1 \times k]$	Labels $\{y_1, \dots, y_k\}$ assigned to Gaussians.
`.cov_type` [*string*]	Optional field which indicates shape of the covariance matrix (see Table 4.3).
`.fun = 'pdfgauss'`	Identifies function associated with this data type.

of the sample data (see Figure 4.1a). The isoline of the distribution function of the bivariate Gaussian is visualized together with the sample data (see Figure 4.1b).

```
% univariate case
model = struct('Mean',1,'Cov',2);     % Gaussian parameters
figure; hold on;
% plot pdf. of the Gaussisan
plot([-4:0.1:5],pdfgauss([-4:0.1:5],model),'r');
[Y,X] = hist(gsamp(model,500),10);   % compute histogram
bar(X,Y/500);                        % plot histogram

% bi-variate case
model.Mean = [1;1];                  % Mean vector
model.Cov = [1 0.6; 0.6 1];          % Covariance matrix
figure; hold on;
ppatterns(gsamp(model,250));         % plot sampled data
pgauss(model);                       % plot shape
```

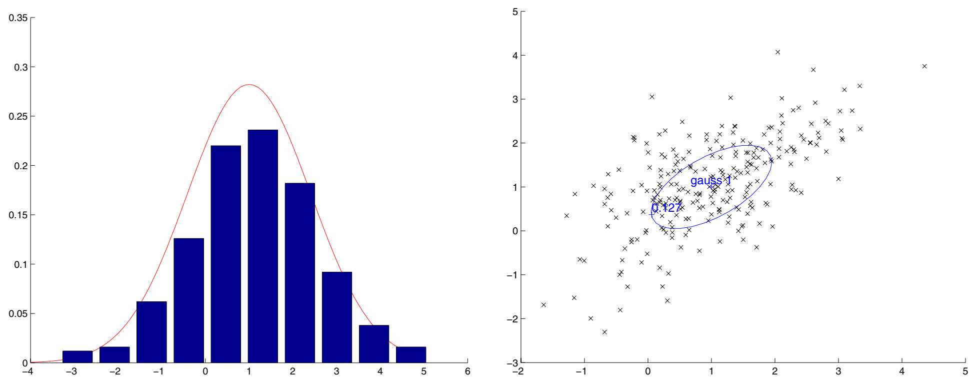
4.2 Maximum-Likelihood estimation of GMM

Let the probability distribution

$$P_{XY|\Theta}(\boldsymbol{x}, y|\boldsymbol{\theta}) = P_{X|Y\Theta}(\boldsymbol{x}|y, \boldsymbol{\theta}) P_{Y|\Theta}(y|\boldsymbol{\theta}) \,,$$

Table 4.5: Data-type used to represent the Gaussian Mixture Model.

Gaussian Mixture Model (structure array):	
`.Mean` $[n \times c]$	Mean vectors $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_c\}$.
`.Cov` $[n \times n \times c]$	Covariance matrices $\{\boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_c\}$.
`.Prior` $[c \times 1]$	Weights of Gaussians $\{P_K(1), \ldots, P_K(c)\}$.
`.fun = 'pdfgmm'`	Identifies function associated with this data type.



(a) (b)

Figure 4.1: Sampling from the univariate (a) and bivariate (b) Gaussian distribution.

be known up to parameters $\boldsymbol{\theta} \in \Theta$. The vector of observations $\boldsymbol{x} \in \mathbb{R}^n$ and the hidden state $y \in \mathcal{Y} = \{1, \ldots, c\}$ are assumed to be realizations of random variables which are independent and identically distributed according to the $P_{XY|\Theta}(\boldsymbol{x}, y|\boldsymbol{\theta})$. The conditional probability distribution $P_{X|Y\Theta}(\boldsymbol{x}|y, \boldsymbol{\theta})$ is assumed to be Gaussian distribution. The $\boldsymbol{\theta}$ involves parameters $(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)$, $y \in \mathcal{Y}$ of Gaussian components and the values of the discrete distribution $P_{Y|\Theta}(y|\boldsymbol{\theta})$, $y \in \mathcal{Y}$. The marginal probability $P_{X|\Theta}(\boldsymbol{x}|\boldsymbol{\theta})$ is the Gaussian mixture model (4.2). The Maximum-Likelihood estimation of the parameters $\boldsymbol{\theta}$ of the GMM for the case of complete and incomplete data is described in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1 Complete data

The input is a set $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ which contains both vectors of observations $\boldsymbol{x}_i \in \mathbb{R}^n$ and corresponding hidden states $y_i \in \mathcal{Y}$. The pairs $(\boldsymbol{x}_i, y_i)$ are assumed to samples from random variables which are independent and identically distributed (i.i.d.) according to $P_{XY|\Theta}(\boldsymbol{x}, y|\boldsymbol{\theta})$. The logarithm of probability $P(\mathcal{T}_{XY}|\boldsymbol{\theta})$ is referred to as the log-likelihood $L(\boldsymbol{\theta}|\mathcal{T}_{XY})$ of $\boldsymbol{\theta}$ with respect to $\mathcal{T}_{XY}$. Thanks to the i.i.d. assumption the log-likelihood can be factorized as

$$L(\boldsymbol{\theta}|\mathcal{T}_{XY}) = \log P(\mathcal{T}_{XY}|\boldsymbol{\theta}) = \sum_{i=1}^{l} \log P_{XY|\Theta}(\boldsymbol{x}_i, y_i|\boldsymbol{\theta}) \, .$$

The problem of Maximum-Likelihood (ML) estimation from the complete data $\mathcal{T}_{XY}$ is defined as

$$\boldsymbol{\theta}^* = \operatorname*{argmax}_{\boldsymbol{\theta}\in\Theta} L(\boldsymbol{\theta}|\mathcal{T}_{XY}) = \operatorname*{argmax}_{\boldsymbol{\theta}\in\Theta} \sum_{i=1}^{l} \log P_{XY|\Theta}(\boldsymbol{x}_i, y_i|\boldsymbol{\theta}) \, . \tag{4.3}$$

The ML estimate of the parameters $\boldsymbol{\theta}$ of the GMM from the complete data set $\mathcal{T}_{XY}$ has an analytical solution. The computation of the ML estimate is implemented in the function `mlcgmm`.

Example: Maximum-Likelihood estimate from complete data

The parameters of the Gaussian mixture model is estimated from the complete Riply's data set `riply_trn`. The data are binary labeled thus the GMM has two Gaussians components. The estimated GMM is visualized as: (i) shape of components of the GMM (see Figure 4.2a) and (ii) contours of the distribution function (see Figure 4.2b) .

```
data = load('riply_trn'); % load labeled (complete) data
model = mlcgmm(data);     % ML estimate of GMM

% visualization
figure; hold on;
ppatterns(data); pgauss(model);
figure; hold on;
ppatterns(data);
pgmm(model,struct('visual','contour'));
```

4.2.2 Incomplete data

The input is a set $\mathcal{T}_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ which contains only vectors of observations $\boldsymbol{x}_i \in \mathbb{R}^n$ without knowledge about the corresponding hidden states $y_i \in \mathcal{Y}$. The logarithm of probability $P(\mathcal{T}_X|\boldsymbol{\theta})$ is referred to as the log-likelihood $L(\boldsymbol{\theta}|\mathcal{T}_X)$ of $\boldsymbol{\theta}$ with respect to $\mathcal{T}_X$. Thanks to the i.i.d. assumption the log-likelihood can be factorized as

$$L(\boldsymbol{\theta}|\mathcal{T}_X) = \log P(\mathcal{T}_X|\boldsymbol{\theta}) = \sum_{i=1}^{l} \sum_{y\in\mathcal{Y}} P_{X|Y\Theta}(\boldsymbol{x}_i|y_i, \boldsymbol{\theta}) P_{Y|\Theta}(y_i|\boldsymbol{\theta}) \, .$$

The problem of Maximum-Likelihood (ML) estimation from the incomplete data $\mathcal{T}_X$ is defined as

$$\boldsymbol{\theta}^* = \operatorname*{argmax}_{\boldsymbol{\theta}\in\Theta} L(\boldsymbol{\theta}|\mathcal{T}_X) = \operatorname*{argmax}_{\boldsymbol{\theta}\in\Theta} \sum_{i=1}^{l} \sum_{y\in\mathcal{Y}} P_{X|Y\Theta}(\boldsymbol{x}_i|y_i,\boldsymbol{\theta}) P_{Y|\Theta}(y_i|\boldsymbol{\theta}) \,. \tag{4.4}$$

The ML estimate of the parameters $\boldsymbol{\theta}$ of the GMM from the incomplete data set $\mathcal{T}_{XY}$ does not have an analytical solution. The numerical optimization has to be used instead.

The Expectation-Maximization (EM) algorithm is an iterative procedure for ML estimation from the incomplete data. The EM algorithm builds a sequence of parameter estimates

$$\boldsymbol{\theta}^{(0)}, \boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(t)} \,,$$

such that the log-likelihood $L(\boldsymbol{\theta}^{(t)}|\mathcal{T}_X)$ monotonically increases, i.e.,

$$L(\boldsymbol{\theta}^{(0)}|\mathcal{T}_X) < L(\boldsymbol{\theta}^{(1)}|\mathcal{T}_X) < \ldots < L(\boldsymbol{\theta}^{(t)}|\mathcal{T}_X)$$

until a stationary point $L(\boldsymbol{\theta}^{(t-1)}|\mathcal{T}_X) = L(\boldsymbol{\theta}^{(t)}|\mathcal{T}_X)$ is achieved. The EM algorithm is a local optimization technique thus the estimated parameters depend on the initial guess $\boldsymbol{\theta}^{(0)}$. The random guess or K-means algorithm (see Section 4.5) are commonly used to select the initial $\boldsymbol{\theta}^{(0)}$. The EM algorithm for the GMM is implemented in the function `emgmm`. An interactive demo on the EM algorithm for the GMM is implemented in `demo_emgmm`.

References: The EM algorithm (including convergence proofs) is described in book [26]. A nice description can be found in book [3]. The monograph [18] is entirely devoted to the EM algorithm. The first papers describing the EM are [25, 5].

Example: Maximum-Likelihood estimate from incomplete data

The estimation of the parameters of the GMM is demonstrated on a synthetic data. The ground truth model is the GMM with two univariate Gaussian components. The ground truth model is sampled to obtain a training (incomplete) data. The EM algorithm is applied to estimate the GMM parameters. The random initialization is used by default. The ground truth model, sampled data and the estimate model are visualized in Figure 4.3a. The plot of the log-likelihood function $L(\boldsymbol{\theta}^{(t)}|\mathcal{T}_X)$ with respect to the number of iterations t is visualized in Figure 4.3b.

```
% ground truth Gaussian mixture model
true_model.Mean = [-2 2];
true_model.Cov = [1 0.5];
true_model.Prior = [0.4 0.6];
sample = gmmsamp(true_model, 250);
```

```
% ML estimation by EM
options.ncomp = 2;        % number of Gaussian components
options.verb = 1;         % display progress info
estimated_model = emgmm(sample.X,options);

% visualization
figure;
ppatterns(sample.X);
h1 = pgmm(true_model,struct('color','r'));
h2 = pgmm(estimated_model,struct('color','b'));
legend([h1(1) h2(1)],'Ground truth', 'ML estimation');
figure; hold on;
xlabel('iterations'); ylabel('log-likelihood');
plot( estimated_model.logL );
```

4.3 Minimax estimation

The input is a set $\mathcal{T}_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ contains vectors $\boldsymbol{x}_i \in \mathbb{R}^n$ which are realizations of random variable distributed according to $P_{X|\boldsymbol{\Theta}}(\boldsymbol{x}|\boldsymbol{\theta})$. The vectors $\mathcal{T}_X$ are assumed to be a realizations with high value of the distribution function $P_{X|\boldsymbol{\Theta}}(\boldsymbol{x}|\boldsymbol{\theta})$. The $P_{X|\boldsymbol{\Theta}}(\boldsymbol{x}|\boldsymbol{\theta})$ is known up to the parameter $\boldsymbol{\theta} \in \Theta$. The minimax estimation is defined as

$$\boldsymbol{\theta}^* = \operatorname*{argmax}_{\boldsymbol{\theta}\in\Theta} \min_{\boldsymbol{x}\in\mathcal{T}_X} \log P_{X|\Theta}(\boldsymbol{x}|\boldsymbol{\theta}) \,. \tag{4.5}$$

If a procedure for the Maximum-Likelihood estimate (global solution) of the distribution $P_{X|\Theta}(\boldsymbol{x}|\boldsymbol{\theta})$ exists then a numerical iterative algorithm which converges to the optimal estimate $\boldsymbol{\theta}^*$ can be constructed. The algorithm builds a series of estimates $\boldsymbol{\theta}^{(0)}, \boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(t)}$ which converges to the optimal estimate $\boldsymbol{\theta}^*$. The algorithm converges in a finite number of iterations to such an estimate $\boldsymbol{\theta}$ that the condition

$$\min_{\boldsymbol{x}\in\mathcal{T}_X} \log P_{X|\Theta}(\boldsymbol{x}|\boldsymbol{\theta}^*) - \min_{\boldsymbol{x}\in\mathcal{T}_X} \log P_{X|\Theta}(\boldsymbol{x}|\boldsymbol{\theta}) < \varepsilon \,, \tag{4.6}$$

holds for $\varepsilon > 0$ which defines closeness to the optimal solution $\boldsymbol{\theta}^*$. The inequality (4.6) can be evaluated efficiently thus it is a natural choice for the stopping condition of the algorithm. The STPRtool contains an implementation of the minimax algorithm for the Gaussian distribution (4.1). The algorithm is implemented in the function `mmgauss`. An interactive demo on the minimax estimation for the Gaussian distribution is implemented in the function `demo_mmgauss`.
References: The minimax algorithm for probability estimation is described and analyzed in Chapter 8 of the book [26].

Example: Minimax estimation of the Gaussian distribution

The bivariate Gaussian distribution is described by three samples which are known to have high value of the distribution function. The minimax algorithm is applied to estimate the parameters of the Gaussian. In this particular case, the minimax problem is equivalent to searching for the minimal enclosing ellipsoid. The found solution is visualized as the isoline of the distribution function at the point $P_{X|\Theta}(\boldsymbol{x}|\boldsymbol{\theta})$ (see Figure 4.4).

```
X = [[0;0] [1;0] [0;1]];    % points with high p.d.f.
model = mmgauss(X);         % run minimax algorithm

% visualization
figure; ppatterns(X,'xr',13);
pgauss(model, struct('p',exp(model.lower_bound)));
```

4.4 Probabilistic output of classifier

In general, the discriminant function of an arbitrary classifier does not have meaning of probability (e.g., SVM, Perceptron). However, the probabilistic output of the classifier can help in post-processing, for instance, in combining more classifiers together. Fitting a sigmoid function to the classifier output is a way how to solve this problem.

Let $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ is the training set composed of the vectors $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ and the corresponding binary hidden states $y_i \in \mathcal{Y} = \{1, 2\}$. The training set $\mathcal{T}_{XY}$ is assumed to be identically and independently distributed from underlying distribution. Let $f\colon \mathcal{X} \subseteq \mathbb{R}^n \to \mathbb{R}$ be a discriminant function trained from the data $\mathcal{T}_{XY}$ by an arbitrary learning method. The aim is to estimate parameters of a posteriori distribution $P_{Y|F\Theta}(y|f(\boldsymbol{x}), \boldsymbol{\theta})$ of the hidden state y given the value of the discriminant function $f(\boldsymbol{x})$. The distribution is modeled by a sigmoid function

$$P_{Y|F\Theta}(1|f(\boldsymbol{x}), \boldsymbol{\theta}) = \frac{1}{1 + \exp(a_1 f(\boldsymbol{x}) + a_2)} ,$$

which is determined by parameters $\boldsymbol{\theta} = [a_1, a_2]^T$. The log-likelihood function is defined as

$$L(\boldsymbol{\theta}|\mathcal{T}_{XY}) = \sum_{i=1}^{l} \log P_{Y|F\Theta}(y_i|f(\boldsymbol{x}_i), \boldsymbol{\theta}) .$$

The parameters $\boldsymbol{\theta} = [a_1, a_2]^T$ can be estimated by the maximum-likelihood method

$$\boldsymbol{\theta} = \operatorname*{argmax}_{\boldsymbol{\theta}'} L(\boldsymbol{\theta}'|\mathcal{T}_{XY}) = \operatorname*{argmax}_{\boldsymbol{\theta}'} \sum_{i=1}^{l} \log P_{Y|F\Theta}(y_i|f(\boldsymbol{x}_i), \boldsymbol{\theta}') . \qquad (4.7)$$

The STPRtool provides an implementation `mlsigmoid` which uses the Matlab Optimization toolbox function `fminunc` to solve the task (4.7). The function produces the parameters $\boldsymbol{\theta} = [a_1, a_2]^T$ which are stored in the data-type describing the sigmoid (see Table 4.6). The evaluation of the sigmoid is implemented in function `sigmoid`.

Table 4.6: Data-type used to describe sigmoid function.

Sigmoid function (structure array):	
`.A` $[2 \times 1]$	Parameters $\boldsymbol{\theta} = [a_1, a_2]^T$ of the sigmoid function.
`.fun = 'sigmoid'`	Identifies function associated with this data type.

References: The method of fitting the sigmoid function is described in [22].

Example: Probabilistic output for SVM

This example is implemented in the script `demo_svmpout`. The example shows how to train the probabilistic output for the SVM classifier. The Riply's data set `riply_trn` is used for training the SVM and the ML estimation of the sigmoid. The ML estimated of Gaussian mixture model (GMM) of the SVM output is used for comparison. The GMM allows to compute the a posteriori probability for comparison to the sigmoid estimate. The example is implemented in the script `demo_svmpout`. Figure 4.5a shows found probabilistic models of a posteriori probability of the SVM output. Figure 4.5b shows the decision boundary of SVM classifier for illustration.

```
% load training data
data = load('riply_trn');

% train SVM
options = struct('ker','rbf','arg',1,'C',10);
svm_model = smo(data,options);

% compute SVM output
[dummy,svm_output.X] = svmclass(data.X,svm_model);
svm_output.y = data.y;

% fit sigmoid function to svm output
sigmoid_model = mlsigmoid(svm_output);

% ML estimation of GMM model of SVM output
gmm_model = mlcgmm(svm_output);

% visulization
figure; hold on;
```

```
xlabel('svm output f(x)'); ylabel('p(y=1|f(x))');

% sigmoid model
fx = linspace(min(svm_output.X), max(svm_output.X), 200);
sigmoid_apost = sigmoid(fx,sigmoid_model);
hsigmoid = plot(fx,sigmoid_apost,'k');
ppatterns(svm_output);

% GMM model
pcond = pdfgauss( fx, gmm_model);
gmm_apost = (pcond(1,:)*gmm_model.Prior(1))./...
  (pcond(1,:)*gmm_model.Prior(1)+(pcond(2,:)*gmm_model.Prior(2)));
hgmm = plot(fx,gmm_apost,'g');
hcomp = pgauss(gmm_model);

legend([hsigmoid,hgmm,hcomp],'P(y=1|f(x)) ML-Sigmoid',...
 'P(y=1|f(x)) ML-GMM','P(f(x)|y=1) ML-GMM','P(f(x)|y=2) ML-GMM');

% SVM decision boundary
figure; ppatterns(data); psvm(svm_model);
```

4.5 K-means clustering

The input is a set $\mathcal{T}_X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_l\}$ which contains vectors $\boldsymbol{x}_i \in \mathbb{R}^n$. Let $\boldsymbol{\theta} = \{\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_c\}$, $\boldsymbol{\mu}_i \in \mathbb{R}^n$ be a set of unknown cluster centers. Let the objective function $F(\boldsymbol{\theta})$ be defined as

$$F(\boldsymbol{\theta}) = \frac{1}{l} \sum_{i=1}^{l} \min_{y=1,\dots,c} \|\boldsymbol{\mu}_y - \boldsymbol{x}_i\|^2 .$$

The small value of $F(\boldsymbol{\theta})$ indicates that the centers $\boldsymbol{\theta}$ well describe the input set $\mathcal{T}_X$. The task is to find such centers $\boldsymbol{\theta}$ which describe the set $\mathcal{T}_X$ best, i.e., the task is to solve

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) = \operatorname*{argmin}_{\boldsymbol{\theta}} \frac{1}{l} \sum_{i=1}^{l} \min_{y=1,\dots,c} \|\boldsymbol{\mu}_y - \boldsymbol{x}_i\|^2 .$$

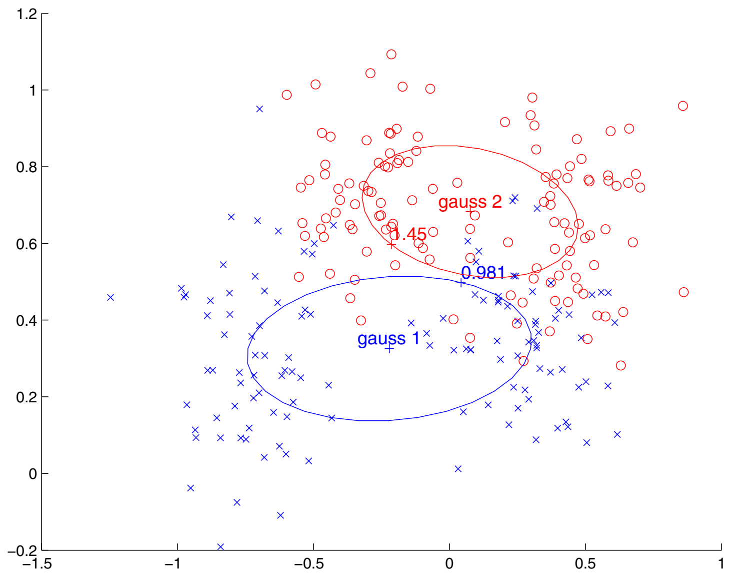
The K-means[1] is an iterative procedure which iteratively builds a series $\boldsymbol{\theta}^{(0)}, \boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(t)}$. The K-means algorithm converges in a finite number of steps to the local optimum of the objective function $F(\boldsymbol{\theta})$. The found solution depends on the initial guess $\boldsymbol{\theta}^{(0)}$ which

[1]The K in the name K-means stands for the number of centers which is here denoted by c.

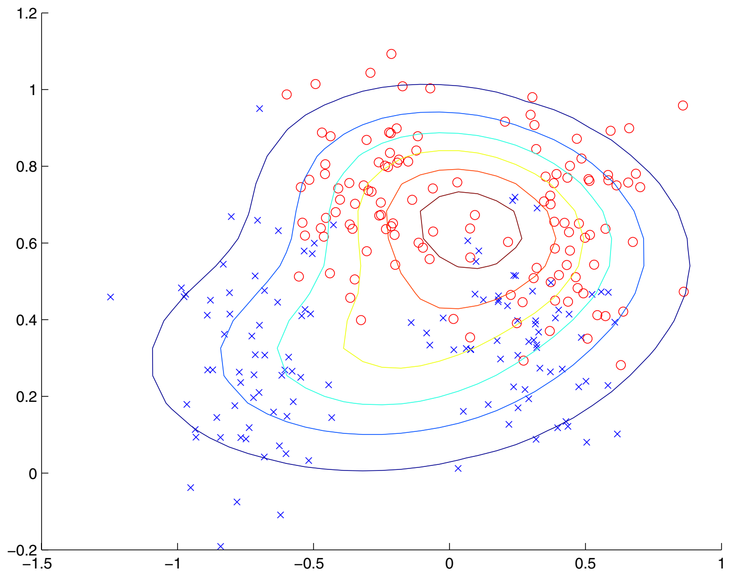
is usually selected randomly. The K-means algorithm is implemented in the function `kmeans`.
References: The K-means algorithm is described in the books [26, 6, 3].

Example: K-means clustering
The K-means algorithm is used to find 4 clusters in the Riply's training set. The solution is visualized as the cluster centers. The vectors classified to the corresponding centers are distinguished by color (see Figure 4.6a). The plot of the objective function $F(\boldsymbol{\theta}^{(t)})$ with respect to the number of iterations is shown in Figure 4.6b.

```
data = load('riply_trn');              % load data
[model,data.y] = kmeans(data.X, 4 ); % run k-means

% visualization
figure; ppatterns(data);
ppatterns(model.X,'sk',14);
pboundary( model );
figure; hold on;
xlabel('t'); ylabel('F');
plot(model.MsErr);
```

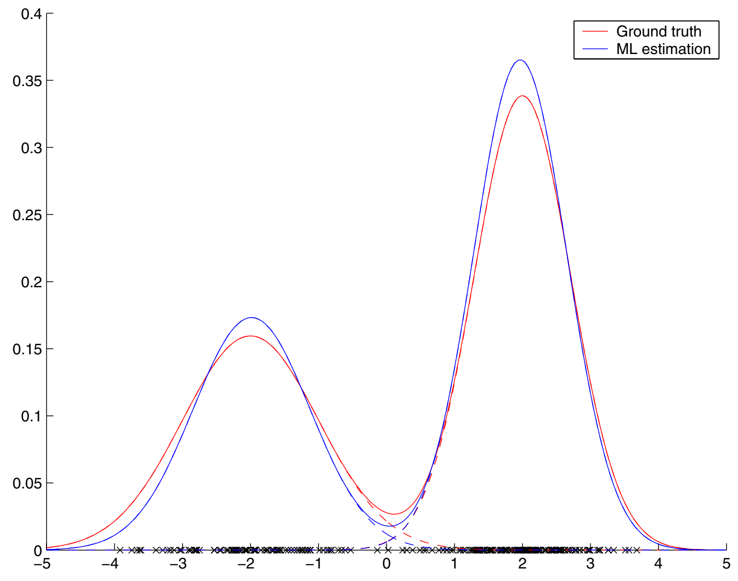


(a)

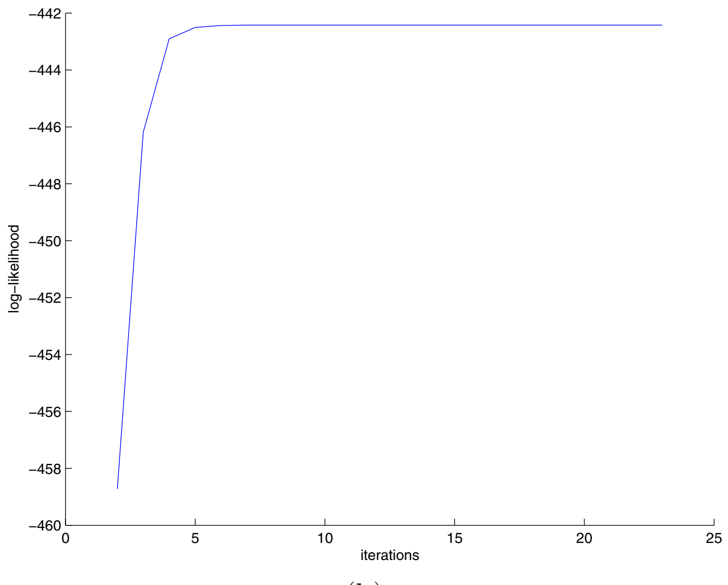


(b)

Figure 4.2: Example of the Maximum-Likelihood of the Gaussian Mixture Model from the incomplete data. Figure (a) shows components of the GMM and Figure (b) shows contour plot of the distribution function.



(a)



(b)

Figure 4.3: Example of using the EM algorithm for ML estimation of the Gaussian mixture model. Figure (a) shows ground truth and the estimated GMM and Figure(b) shows the log-likelihood with respect to the number of iterations of the EM.

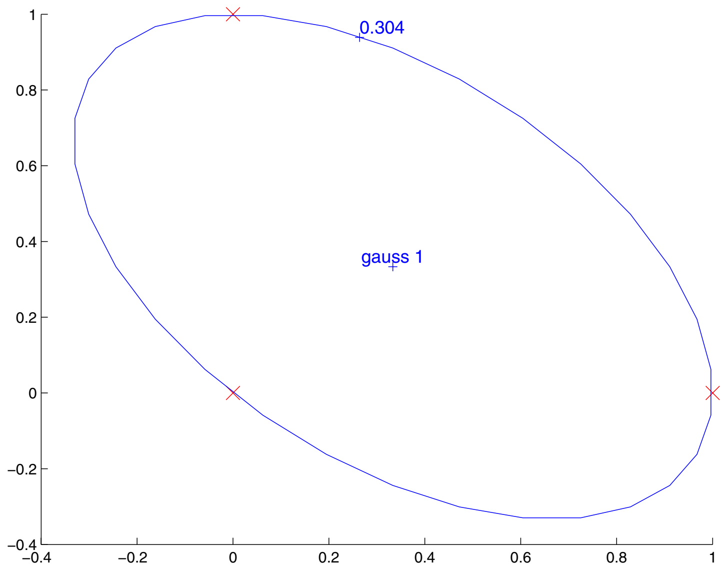
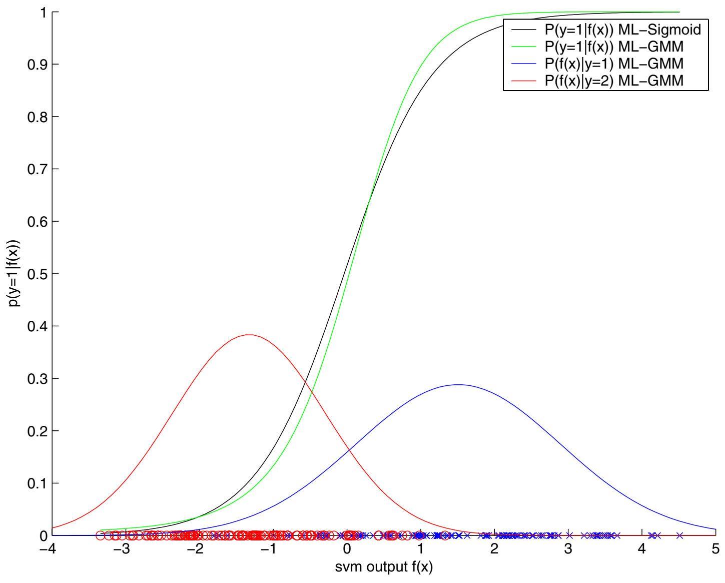
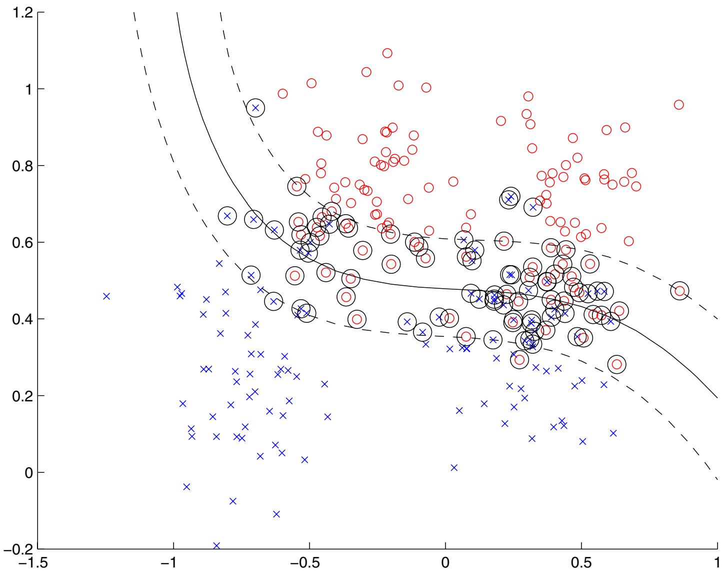


Figure 4.4: Example of the Minimax estimation of the Gaussian distribution.

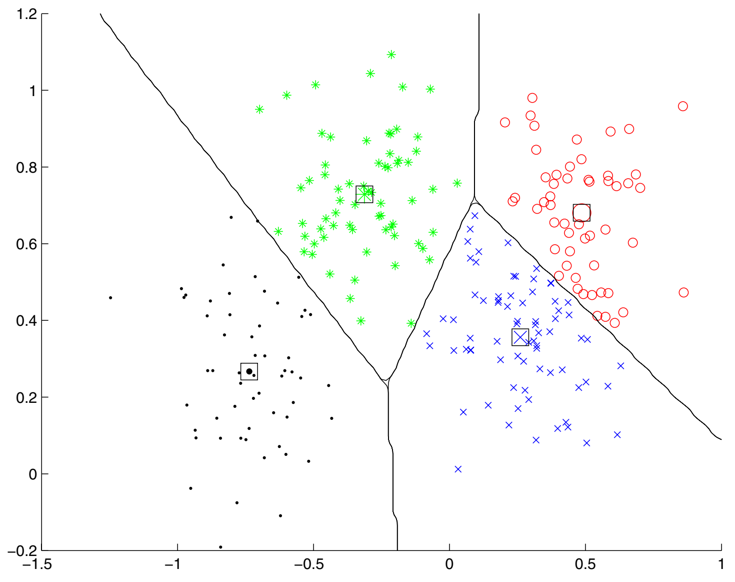


(a)

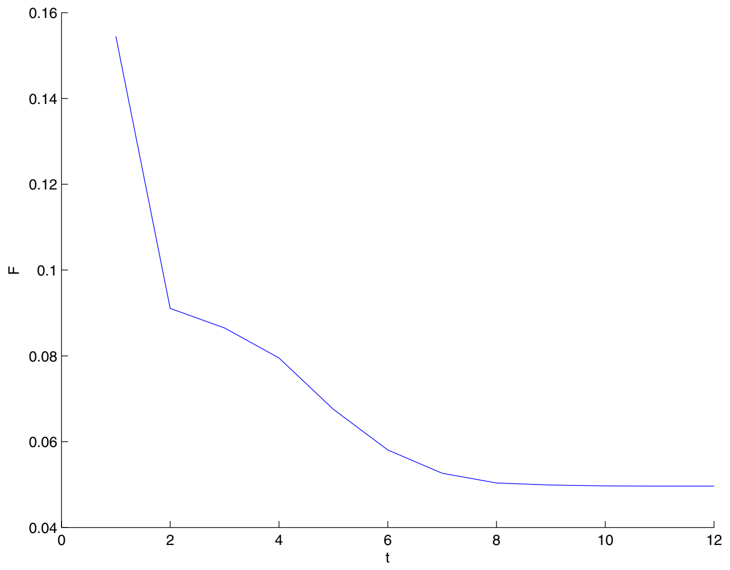


(b)

Figure 4.5: Example of fitting a posteriori probability to the SVM output. Figure (a) shows the sigmoid model and GMM model both fitted by ML estimated. Figure (b) shows the corresponding SVM classifier.



(a)



(b)

Figure 4.6: Example of using the K-means algorithm to cluster the Riply's data set. Figure (a) shows found clusters and Figure (b) contains the plot of the objective function $F(\boldsymbol{\theta}^{(t)})$ with respect to the number of iterations.

Chapter 5

Support vector and other kernel machines

5.1 Support vector classifiers

The binary support vector classifier uses the discriminant function $f \colon \mathcal{X} \subseteq \mathbb{R}^n \to \mathbb{R}$ of the following form

$$f(\boldsymbol{x}) = \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}_S(\boldsymbol{x}) \rangle + b \,. \tag{5.1}$$

The $\boldsymbol{k}_S(\boldsymbol{x}) = [k(\boldsymbol{x}, \boldsymbol{s}_1), \dots, k(\boldsymbol{x}, \boldsymbol{s}_d)]^T$ is the vector of evaluations of kernel functions centered at the support vectors $\mathcal{S} = \{\boldsymbol{s}_1, \dots, \boldsymbol{s}_d\}$, $\boldsymbol{s}_i \in \mathbb{R}^n$ which are usually subset of the training data. The $\boldsymbol{\alpha} \in \mathbb{R}^l$ is a weight vector and $b \in \mathbb{R}$ is a bias. The binary classification rule $q \colon \mathcal{X} \to \mathcal{Y} = \{1, 2\}$ is defined as

$$q(\boldsymbol{x}) = \left\{ \begin{array}{lll} 1 & \text{for} & f(\boldsymbol{x}) \geq 0 \,, \\ 2 & \text{for} & f(\boldsymbol{x}) < 0 \,. \end{array} \right. \tag{5.2}$$

The data-type used by the STPRtool to represent the binary SVM classifier is described in Table 5.2.

The multi-class generalization involves a set of discriminant functions $f_y \colon \mathcal{X} \subseteq \mathbb{R}^n \to \mathbb{R}$, $y \in \mathcal{Y} = \{1, 2, \dots, c\}$ defined as

$$f_y(\boldsymbol{x}) = \langle \boldsymbol{\alpha}_y \cdot \boldsymbol{k}_S(\boldsymbol{x}) \rangle + b_y \,, \qquad y \in \mathcal{Y} \,.$$

Let the matrix $\mathbf{A} = [\boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_c]$ be composed of all weight vectors and $\boldsymbol{b} = [b_1, \dots, b_c]^T$ be a vector of all biases. The multi-class classification rule $q \colon \mathcal{X} \to \mathcal{Y} = \{1, 2, \dots, c\}$ is defined as

$$q(\boldsymbol{x}) = \operatorname*{argmax}_{y \in \mathcal{Y}} f_y(\boldsymbol{x}) \,. \tag{5.3}$$

The data-type used by the STPRtool to represent the multi-class SVM classifier is described in Table 5.3. Both the binary and the multi-class SVM classifiers are implemented in the function `svmclass`.

The majority voting strategy is other commonly used method to implement the multi-class SVM classifier. Let $q_j \colon \mathcal{X} \subseteq \mathbb{R}^n \rightarrow \{y_j^1, y_j^2\}$, $j = 1, \ldots, g$ be a set of g binary SVM rules (5.2). The j-th rule $q_j(\boldsymbol{x})$ classifies the inputs $\boldsymbol{x}$ into class $y_j^1 \in \mathcal{Y}$ or $y_j^2 \in \mathcal{Y}$. Let $\boldsymbol{v}(\boldsymbol{x})$ be a vector $[c \times 1]$ of votes for classes $\mathcal{Y}$ when the input $\boldsymbol{x}$ is to be classified. The vector $\boldsymbol{v}(\boldsymbol{x}) = [v_1(\boldsymbol{x}), \ldots, v_c(\boldsymbol{x})]^T$ is computed as:

```
Set $v_y = 0$, $\forall y \in \mathcal{Y}$.
For $j = 1, \ldots, g$ do
    $v_y(\boldsymbol{x}) = v_y(\boldsymbol{x}) + 1$ where $y = q_j(\boldsymbol{x})$.
end.
```

The majority votes based multi-class classifier assigns the input $\boldsymbol{x}$ into such class $y \in \mathcal{Y}$ having the majority of votes

$$y = \operatorname*{argmax}_{y'=1,\ldots,g} v_{y'}(\boldsymbol{x}) \,. \tag{5.4}$$

The data-type used to represent the majority voting strategy for the multi-class SVM classifier is described in Table 5.4. The strategy (5.4) is implemented in the function `mvsvmclass`. The implemented methods to deal with the SVM and related kernel classifiers are enlisted in Table 5.1
References: Books describing Support Vector Machines are for example [31, 4, 30]

5.2 Binary Support Vector Machines

The input is a set $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ of training vectors $\boldsymbol{x}_i \in \mathbb{R}^n$ and corresponding hidden states $y_i \in \mathcal{Y} = \{1, 2\}$. The linear SVM aims to train the linear discriminant function $f(\boldsymbol{x}) = \langle \boldsymbol{w} \cdot \boldsymbol{x} \rangle + b$ of the binary classifier

$$q(\boldsymbol{x}) = \begin{cases} 1 & \text{for} \quad f(\boldsymbol{x}) \geq 0 \,, \\ 2 & \text{for} \quad f(\boldsymbol{x}) < 0 \,. \end{cases}$$

The training of the optimal parameters $(\boldsymbol{w}^*, b^*)$ is transformed to the following quadratic programming task

$$(\boldsymbol{w}^*, b^*) = \operatorname*{argmin}_{\boldsymbol{w}, b} \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{l} \xi_i^p \,, \tag{5.5}$$

$$\begin{aligned} \langle \boldsymbol{w} \cdot \boldsymbol{x}_i \rangle + b &\geq +1 - \xi_i \,, \qquad i \in \mathcal{I}_1 \,, \\ \langle \boldsymbol{w} \cdot \boldsymbol{x}_i \rangle + b &\leq -1 + \xi_i \,, \qquad i \in \mathcal{I}_2 \,, \\ \xi_i &\geq 0 \,, \qquad i \in \mathcal{I}_1 \cup \mathcal{I}_2 \,. \end{aligned}$$

Table 5.1: Implemented methods: Support vector and other kernel machines.

`svmclass`	Support Vector Machines Classifier.
`mvsvmclass`	Majority voting multi-class SVM classifier.
`evalsvm`	Trains and evaluates Support Vector Machines classifier.
`bsvm2`	Multi-class BSVM with L2-soft margin.
`oaasvm`	Multi-class SVM using One-Against-All decomposition.
`oaosvm`	Multi-class SVM using One-Against-One decomposition.
`smo`	Sequential Minimal Optimization for binary SVM with L1-soft margin.
`svmlight`	Interface to SVM^{light} software.
`svmquadprog`	SVM trained by Matlab Optimization Toolbox.
`diagker`	Returns diagonal of kernel matrix of given data.
`kdist`	Computes distance between vectors in kernel space.
`kernel`	Evaluates kernel function.
`kfd`	Kernel Fisher Discriminant.
`kperceptr`	Kernel Perceptron.
`minball`	Minimal enclosing ball in kernel feature space.
`rsrbf`	Reduced Set Method for RBF kernel expansion.
`rbfpreimg`	RBF pre-image by Schoelkopf's fixed-point algorithm.
`rbfpreimg2`	RBF pre-image problem by Gradient optimization.
`rbfpreimg3`	RBF pre-image problem by Kwok-Tsang's algorithm.
`demo_svm`	Demo on Support Vector Machines.

The $\mathcal{I}_1 = \{i\colon y_i = 1\}$ and $\mathcal{I}_2 = \{i\colon y_i = 2\}$ are sets of indices. The $C > 0$ stands for the regularization constant. The $\xi_i \geq 0$, $i \in \mathcal{I}_1 \cup \mathcal{I}_2$ are slack variables used to relax the inequalities for the case of non-separable data. The linear $p = 1$ and quadratic $p = 2$ term for the slack variables ξ_i is used. In the case of the parameter $p = 1$, the $L1$-soft margin SVM is trained and for $p = 2$ it is denoted as the $L2$-soft margin SVM.

The training of the non-linear (kernel) SVM with $L1$-soft margin corresponds to solving the following QP task

$$\boldsymbol{\beta}^* = \operatorname*{argmax}_{\boldsymbol{\beta}} \langle \boldsymbol{\beta} \cdot \mathbf{1} \rangle - \frac{1}{2} \langle \boldsymbol{\beta} \cdot \mathbf{H} \boldsymbol{\beta} \rangle \,, \tag{5.6}$$

subject to

$$\begin{array}{rcl} \langle \boldsymbol{\beta} \cdot \boldsymbol{\gamma} \rangle & = & 1 \,, \\ \boldsymbol{\beta} & \geq & \mathbf{0} \,, \\ \boldsymbol{\beta} & \leq & \mathbf{1} C \,. \end{array}$$

Table 5.2: Data-type used to describe binary SVM classifier.

Binary SVM classifier (structure array):	
`.Alpha` $[d \times 1]$	Weight vector $\boldsymbol{\alpha}$.
`.b` $[1 \times 1]$	Bias b.
`.sv.X` $[n \times d]$	Support vectors $\mathcal{S} = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_d\}$.
`.options.ker` $[string]$	Kernel identifier.
`.options.arg` $[1 \times p]$	Kernel argument(s).
`.fun = 'svmclass'`	Identifies function associated with this data type.

Table 5.3: Data-type used to describe multi-class SVM classifier.

Multi-class SVM classifier (structure array):	
`.Alpha` $[d \times c]$	Weight vectors $\mathbf{A} = [\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_c]$.
`.b` $[c \times 1]$	Vector of biased $\boldsymbol{b} = [b_1, \ldots, b_c]^T$.
`.sv.X` $[n \times d]$	Support vectors $\mathcal{S} = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_d\}$.
`.options.ker` $[string]$	Kernel identifier.
`.options.arg` $[1 \times p]$	Kernel argument(s).
`.fun = 'svmclass'`	Identifies function associated with this data type.

The $\mathbf{0}$ $[l \times 1]$ is vector of zeros and the $\mathbf{1}$ $[l \times 1]$ is vector of ones. The vector $\boldsymbol{\gamma}$ $[l \times 1]$ and the matrix $\mathbf{H}$ $[l \times l]$ are constructed as follows

$$\gamma_i = \left\{ \begin{array}{rl} 1 & \text{if} \quad y_i = 1 \,, \\ -1 & \text{if} \quad y_i = 2 \,, \end{array} \right. \qquad H_{i,j} = \left\{ \begin{array}{rl} k(\boldsymbol{x}_i, \boldsymbol{x}_j) & \text{if} \quad y_i = y_j \,, \\ -k(\boldsymbol{x}_i, \boldsymbol{x}_j) & \text{if} \quad y_i \neq y_j \,, \end{array} \right.$$

where $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is selected kernel function. The discriminant function (5.1) is determined by the weight vector $\boldsymbol{\alpha}$, bias b and the set of support vectors $\mathcal{S}$. The set of support vectors is denoted as $\mathcal{S} = \{\boldsymbol{x}_i\colon i \in \mathcal{I}_{SV}\}$, where the set of indices is $\mathcal{I}_{SV} = \{i\colon \beta_i > 0\}$. The weight vectors are compute as

$$\alpha_i = \left\{ \begin{array}{rl} \beta_i & \text{if} \quad y_i = 1 \,, \\ -\beta_i & \text{if} \quad y_i = 2 \,, \end{array} \right. \qquad \text{for} \quad i \in \mathcal{I}_{SV} \,. \tag{5.7}$$

The bias b can be computed from the Karush-Kuhn-Tucker (KKT) conditions as the following constrains

$$\begin{array}{rcl} f(\boldsymbol{x}_i) = \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}_S(\boldsymbol{x}_i) \rangle + b & = & +1 \quad \text{for} \quad i \in \{j\colon y_j = 1, 0 < \beta_j < C\} \,, \\ f(\boldsymbol{x}_i) = \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}_S(\boldsymbol{x}_i) \rangle + b & = & -1 \quad \text{for} \quad i \in \{j\colon y_j = 2, 0 < \beta_j < C\} \,, \end{array}$$

must hold for the optimal solution. The bias b is computed as an average over all the constrains such that

$$b = \frac{1}{|\mathcal{I}_b|} \sum_{i \in \mathcal{I}_b} \gamma_i - \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}_S(\boldsymbol{x}_i) \rangle \,.$$

Table 5.4: Data-type used to describe the majority voting strategy for multi-class SVM classifier.

Majority voting SVM classifier (structure array):	
`.Alpha` $[d \times g]$	Weight vectors $\mathbf{A} = [\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_g]$.
`.b` $[g \times 1]$	Vector of biased $\boldsymbol{b} = [b_1, \ldots, b_g]^T$.
`.bin_y` $[2 \times g]$	Targets for the binary rules. The vector `bin_y(1,:)` contains $[y_1^1, y_2^1, \ldots, y_g^1]$ and the `bin_y(2,:)` contains $[y_1^2, y_2^2, \ldots, y_g^2]$.
`.sv.X` $[n \times d]$	Support vectors $\mathcal{S} = \{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_d\}$.
`.options.ker` $[string]$	Kernel identifier.
`.options.arg` $[1 \times p]$	Kernel argument(s).
`.fun = 'mvsvmclass'`	Identifies function associated with this data type.

where $\mathcal{I}_b = \{i \colon 0 < \beta_i < C\}$ are the indices of the vectors on the boundary.

The training of the non-linear (kernel) SVM with $L2$-soft margin corresponds to solving the following QP task

$$\boldsymbol{\beta}^* = \operatorname*{argmax}_{\boldsymbol{\beta}} \langle \boldsymbol{\beta} \cdot \mathbf{1} \rangle - \frac{1}{2} \langle \boldsymbol{\beta} \cdot (\mathbf{H} + \frac{1}{2C} \mathbf{E}) \boldsymbol{\beta} \rangle \,, \tag{5.8}$$

subject to

$$\begin{aligned} \langle \boldsymbol{\beta} \cdot \boldsymbol{\gamma} \rangle &= 1 \,, \\ \boldsymbol{\beta} &\geq \mathbf{0} \,, \end{aligned}$$

where $\mathbf{E}$ is the identity matrix. The set of support vectors is set to $\mathcal{S} = \{\boldsymbol{x}_i \colon i \in \mathcal{I}_{SV}\}$ where the set of indices is $\mathcal{I}_{SV} = \{i \colon \beta_i > 0\}$. The weight vector $\boldsymbol{\alpha}$ is given by (5.7). The bias b can be computed from the KKT conditions as the following constrains

$$\begin{aligned} f(\boldsymbol{x}_i) = \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}_S(\boldsymbol{x}_i) \rangle + b &= +1 - \frac{\alpha_i}{2C} \quad \text{for} \quad i \in \{i \colon y_j = 1, \beta_j > 0\} \,, \\ f(\boldsymbol{x}_i) = \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}_S(\boldsymbol{x}_i) \rangle + b &= -1 + \frac{\alpha_i}{2C} \quad \text{for} \quad i \in \{i \colon y_j = 2, \beta_j > 0\} \,, \end{aligned}$$

must hold at the optimal solution. The bias b is computed as an average over all the constrains thus

$$b = \frac{1}{|\mathcal{I}_{SV}|} \sum_{i \in \mathcal{I}_{SV}} \gamma_i (1 - \frac{\alpha_i}{2C}) - \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}_S(\boldsymbol{x}_i) \rangle \,.$$

The binary SVM solvers implemented in the STPRtool are enlisted in Table 5.5. An interactive demo on the binary SVM is implemented in `demo_svm`.
References: The Sequential Minimal Optimizer (SMO) is described in the books [23, 27, 30]. The SVM^{light} is described for instance in the book [27].

Example: Training binary SVM.

Table 5.5: Implemented binary SVM solvers.

Function	description
`smo`	Implementation of the Sequential Minimal Optimizer (SMO) used to train the binary SVM with L1-soft margin. It can handle moderate size problems.
`svmlight`	Matlab interface to SVM^{light} software (Version: 5.00) which trains the binary SVM with L1-soft margin. The executable file `svm_learn` for the Linux OS must be installed. It can be downloaded from `http://svmlight.joachims.org/`. The SVM^{light} is very powerful optimizer which can handle large problems.
`svmquadprog`	SVM solver based on the QP solver `quadprog` of the Matlab optimization toolbox. It trains both L1-soft and L2-soft margin binary SVM. The function `quadprog` is a part of the Matlab Optimization toolbox. It useful for small problems only.

The binary SVM is trained on the Riply's training set `riply_trn`. The SMO solver is used in the example but other solvers (`svmlight`, `svmquadprog`) can by used as well. The trained SVM classifier is evaluated on the testing data `riply_tst`. The decision boundary is visualized in Figure 5.1.

```
  trn = load('riply_trn');       % load training data
  options.ker = 'rbf';           % use RBF kernel
  options.arg = 1;               % kernel argument
  options.C = 10;                % regularization constant

  % train SVM classifier
  model = smo(trn,options);
% model = svmlight(trn,options);
% model = svmquadprog(trn,options);

  % visualization
  figure;
  ppatterns(trn); psvm(model);

  tst = load('riply_tst');        % load testing data
  ypred = svmclass(tst.X,model);  % classify data
  cerror(ypred,tst.y)             % compute error
```

```
ans =

    0.0920
```

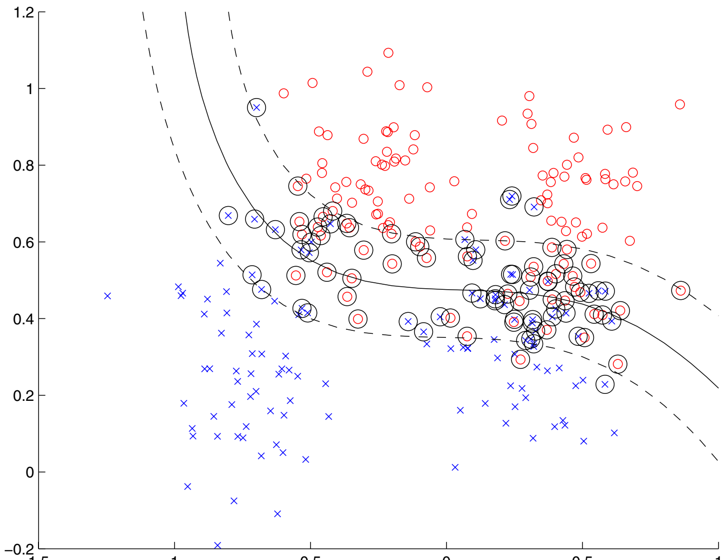


Figure 5.1: Example showing the binary SVM classifier trained on the Riply's data.

5.3 One-Against-All decomposition for SVM

The input is a set $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ of training vectors $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ and corresponding hidden states $y_i \in \mathcal{Y} = \{1, 2, \ldots, c\}$. The goal is to train the multi-class rule $q \colon \mathcal{X} \subseteq \mathbb{R}^n \to \mathcal{Y}$ defined by (5.3). The problem can be solved by the One-Against-All (OAA) decomposition. The OAA decomposition transforms the multi-class problem into a series of c binary subtasks that can be trained by the binary SVM. Let the training set $\mathcal{T}^y_{XY} = \{(\boldsymbol{x}_1, y'_1), \ldots, (\boldsymbol{x}_l, y'_l)\}$ contain the modified hidden states defined as

$$y'_i = \begin{cases} 1 & \text{for} \quad y = y_i \,, \\ 2 & \text{for} \quad y \neq y_i \,. \end{cases}$$

The discriminant functions

$$f_y(\boldsymbol{x}) = \langle \boldsymbol{\alpha}_y \cdot \boldsymbol{k}_S(\boldsymbol{x}) \rangle + b_y \,, \qquad y \in \mathcal{Y} \,,$$

are trained by the binary SVM solver from the set $\mathcal{T}^y_{XY}$, $y \in \mathcal{Y}$. The OAA decomposition is implemented in the function `oaasvm`. The function allows to specify any binary SVM

solver (see Section 5.2) used to solve the binary problems. The result is the multi-class SVM classifier (5.3) represented by the data-type described in Table 5.3.
References: The OAA decomposition to train the multi-class SVM and comparison to other methods is described for instance in the paper [12].

Example: Multi-class SVM using the OAA decomposition
The example shows the training of the multi-class SVM using the OAA decomposition. The SMO binary solver is used to train the binary SVM subtasks. The training data and the decision boundary is visualized in Figure 5.2.

```
data = load('pentagon');       % load data
options.solver = 'smo';        % use SMO solver
options.ker = 'rbf';           % use RBF kernel
options.arg = 1;               % kernel argument
options.C = 10;                % regularization constant
model = oaasvm(data,options ); % training

% visualization
figure;
ppatterns(data);
ppatterns(model.sv.X,'ko',12);
pboundary(model);
```

5.4 One-Against-One decomposition for SVM

The input is a set $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ of training vectors $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ and corresponding hidden states $y_i \in \mathcal{Y} = \{1, 2, \ldots, c\}$. The goal is to train the multi-class rule $q\colon \mathcal{X} \subseteq \mathbb{R}^n \rightarrow \mathcal{Y}$ based on the majority voting strategy defined by (5.4). The one-against-one (OAO) decomposition strategy can be used train such a classifier. The OAO decomposition transforms the multi-class problem into a series of $g = c(c-1)/2$ binary subtasks that can be trained by the binary SVM. Let the training set $\mathcal{T}^j_{XY} = \{(\boldsymbol{x}'_1, y'_1), \ldots, (\boldsymbol{x}'_{l_j}, y'_{l_j})\}$ contain the training vectors $\boldsymbol{x}_i \in \mathcal{I}^j = \{i\colon y_i = y^1 \bigvee y_i = y^2\}$ and the modified the hidden states defined as

$$y'_i = \left\{ \begin{array}{ll} 1 & \text{for} \quad y^1_j = y_i \,, \\ 2 & \text{for} \quad y^2_j = y_i \,, \end{array} \right. \qquad i \in \mathcal{I}^j \,.$$

The training set $\mathcal{T}^j_{XY}$, $j = 1, \ldots, g$ is constructed for all $g = c(c-1)/2$ combinations of classes $y^1_j \in \mathcal{Y}$ and $y^2_j \in \mathcal{Y} \setminus \{y^1_j\}$. The binary SVM rules q_j, $j = 1, \ldots, g$ are trained on the data $\mathcal{T}^j_{XY}$. The OAO decomposition for multi-class SVM is implemented in function

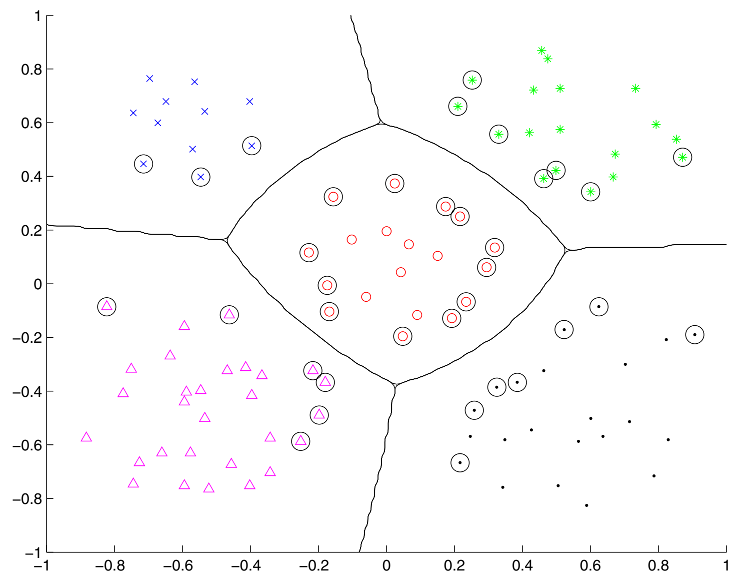


Figure 5.2: Example showing the multi-class SVM classifier build by one-against-all decomposition.

`oaosvm`. The function `oaosvm` allows to specify an arbitrary binary SVM solver to train the subtasks.
References: The OAO decomposition to train the multi-class SVM and comparison to other methods is described for instance in the paper [12].

Example: Multi-class SVM using the OAO decomposition

The example shows the training of the multi-class SVM using the OAO decomposition. The SMO binary solver is used to train the binary SVM subtasks. The training data and the decision boundary is visualized in Figure 5.3.

```
data = load('pentagon');       % load data
options.solver = 'smo';        % use SMO solver
options.ker = 'rbf';           % use RBF kernel
options.arg = 1;               % kernel argument
options.C = 10;                % regularization constant
model = oaosvm(data,options); % training

% visualization
figure;
ppatterns(data);
ppatterns(model.sv.X,'ko',12);
pboundary(model);
```

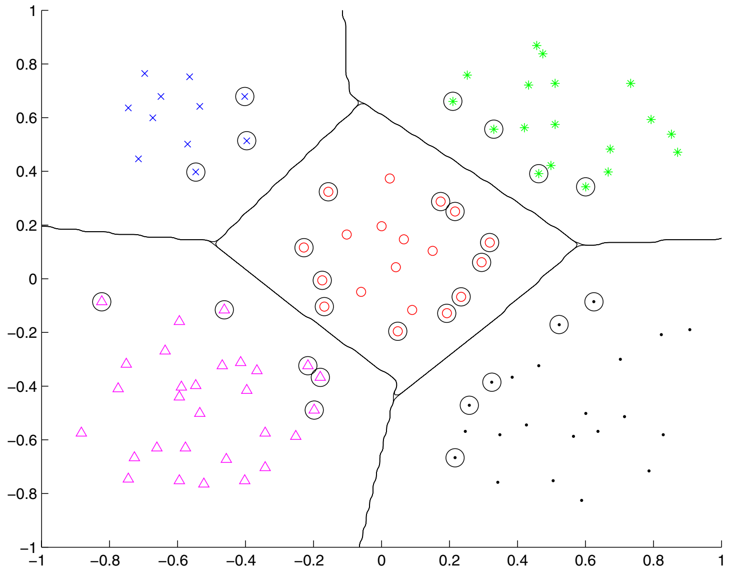


Figure 5.3: Example showing the multi-class SVM classifier build by one-against-one decomposition.

5.5 Multi-class BSVM formulation

The input is a set $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ of training vectors $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ and corresponding hidden states $y_i \in \mathcal{Y} = \{1, 2, \ldots, c\}$. The goal is to train the multi-class rule $q\colon \mathcal{X} \subseteq \mathbb{R}^n \to \mathcal{Y}$ defined by (5.3). In the linear case, the parameters of the multi-class rule can be found by solving the multi-class BSVM formulation which is defined as

$$(\mathbf{W}^*, \boldsymbol{b}^*, \boldsymbol{\xi}^*) = \operatorname*{argmin}_{\mathbf{W}, \boldsymbol{b}} \underbrace{\frac{1}{2} \sum_{y \in \mathcal{Y}} (\|\boldsymbol{w}_y\|^2 + b_y^2) + C \sum_{i \in \mathcal{I}} \sum_{y \in \mathcal{Y} \setminus \{y\}} (\xi_i)^p}_{F(\mathbf{W}, \boldsymbol{b}, \boldsymbol{\xi})}, \tag{5.9}$$

subject to

$$\begin{aligned} \langle \boldsymbol{w}_{y_i} \cdot \boldsymbol{x}_i \rangle + b_{y_i} - (\langle \boldsymbol{w}_y \cdot \boldsymbol{x}_i \rangle + b_y) &\geq 1 - \xi_i^y, \quad & i \in \mathcal{I}, y \in \mathcal{Y} \setminus \{y_i\}, \\ \xi_i^y &\geq 0, \quad & i \in \mathcal{I}, y \in \mathcal{Y} \setminus \{y_i\}, \end{aligned}$$

where $\mathbf{W} = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_c]$ is a matrix of normal vectors, $\boldsymbol{b} = [b_1, \ldots, b_c]^T$ is vector of biases, $\boldsymbol{\xi}$ is a vector of slack variables and $\mathcal{I} = [1, \ldots, l]$ is set of indices. The $L1$-soft margin is used for $p = 1$ and $L2$-soft margin for $p = 2$. The letter B in BSVM stands for the bias term added to the objective (5.9).

In the case of the $L2$-soft margin $p = 2$, the optimization problem (5.9) can be expressed in its dual form

$$\mathbf{A}^* = \operatorname*{argmax}_{\mathbf{A}} \sum_{i\in\mathcal{I}} \sum_{y\in\mathcal{Y}\setminus\{y_i\}} \alpha_i^y - \frac{1}{2} \sum_{i\in\mathcal{I}} \sum_{y\in\mathcal{Y}\setminus\{y_i\}} \sum_{j\in\mathcal{I}} \sum_{u\in\mathcal{Y}\setminus\{y_j\}} \alpha_i^y \alpha_j^u h(y,u,i,j)\,, \qquad (5.10)$$

subject to

$$\alpha_i^y \geq 0\,, \quad i \in \mathcal{I}\,, y \in \mathcal{Y}\setminus\{y_i\}\,,$$

where $\mathbf{A} = [\boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^c]$ are optimized weight vectors. The function $h\colon \mathcal{Y}\times\mathcal{Y}\times\mathcal{I}\times\mathcal{I} \rightarrow \mathbb{R}$ is defined as

$$h(y,u,i,j) = (\langle \boldsymbol{x}_i \cdot \boldsymbol{x}_j \rangle + 1)(\delta(y_i,y_j) + \delta(y,u) - \delta(y_i,u) - \delta(y_j,y)) + \frac{\delta(y,u)\delta(i,j)}{2C}\,. \quad (5.11)$$

The criterion (5.10) corresponds to the single-class SVM criterion which is simple to optimize. The multi-class rule (5.3) is composed of the set of discriminant functions $f_y\colon \mathcal{X} \rightarrow \mathbb{R}$ which are computed as

$$f_y(\boldsymbol{x}) = \sum_{i\in\mathcal{I}} \langle \boldsymbol{x}_i \cdot \boldsymbol{x} \rangle \sum_{u\in\mathcal{Y}\setminus\{y_i\}} \alpha_i^u (\delta(u,y_i) - \delta(u,y)) + b_y\,, \qquad y \in \mathcal{Y}\,, \qquad (5.12)$$

where the bias b_y, $y \in \mathcal{Y}$ is given by

$$b_y = \sum_{i\in\mathcal{I}} \sum_{y\in\mathcal{Y}\setminus\{y_i\}} \alpha_i^u (\delta(y,y_i) - \delta(y,u))\,, \qquad y \in \mathcal{Y}\,.$$

The non-linear (kernel) classifier is obtained by substituting the selected kernel $k\colon \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ for the dot products $\langle \boldsymbol{x} \cdot \boldsymbol{x}' \rangle$ to (5.11) and (5.12). The training of the multi-class BSVM classifier with $L2$-soft margin is implemented in the function `bsvm2`. The optimization task (5.10) can be solved by most the optimizers. The optimizers implemented in the function `bsvm2` are enlisted bellow:

Mitchell-Demyanov-Malozemov (`solver = 'mdm'`).

Kozinec's algorithm (`solver = 'kozinec'`).

Nearest Point Algorithm (`solver = 'npa'`).

The above mentioned algorithms are of iterative nature and converge to the optimal solution of the criterion (5.9). The upper bound F_{UB} and the lower bound F_{LB} on the optimal value $F(\mathbf{W}^*, \boldsymbol{b}^*, \boldsymbol{\xi}^*)$ of the criterion (5.9) can be computed. The bounds are used in the algorithms to define the following two stopping conditions:

Relative tolerance: $\dfrac{F_{UB} - F_{LB}}{F_{LB} + 1} \leq \varepsilon_{rel}$.

Absolute tolerance: $F_{UB} \leq \varepsilon_{abs}$.

References: The multi-class BSVM formulation is described in paper [12]. The transformation of the multi-class BSVM criterion to the single-class criterion (5.10) implemented in the STPRtool is published in paper [9]. The Mitchell-Demyanov-Malozemov and Nearest Point Algorithm are described in paper [14]. The Kozinec's algorithm based SVM solver is described in paper [10].

Example: Multi-class BSVM with $L2$-soft margin.

The example shows the training of the multi-class BSVM using the NPA solver. The other solvers can be used instead by setting the variable `options.solver` (other options are `'kozinec', 'mdm'`). The training data and the decision boundary is visualized in Figure 5.4.

```
data = load('pentagon');         % load data
options.solver = 'npa';          % use NPA solver
options.ker = 'rbf';             % use RBF kernel
options.arg = 1;                 % kernel argument
options.C = 10;                  % regularization constant
model = bsvm2(data,options);     % training

% visualization
figure;
ppatterns(data);
ppatterns(model.sv.X,'ko',12);
pboundary(model);
```

5.6 Kernel Fisher Discriminant

The input is a set $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_l, y_l)\}$ of training vectors $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ and corresponding values of binary state $y_i \in \mathcal{Y} = \{1, 2\}$. The Kernel Fisher Discriminant is the non-linear extension of the linear FLD (see Section 2.3). The input training vectors are assumed to be mapped into a new feature space $\mathcal{F}$ by a mapping function $\boldsymbol{\phi}\colon \mathcal{X} \to \mathcal{F}$. The ordinary FLD is applied on the mapped data training data $\mathcal{T}_{\Phi Y} = \{(\boldsymbol{\phi}(\boldsymbol{x}_1), y_1), \dots, (\boldsymbol{\phi}(\boldsymbol{x}_l), y_l)\}$. The mapping is performed efficiently by means of kernel functions $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ being the dot products of the mapped vectors $k(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{\phi}(\boldsymbol{x}) \cdot \boldsymbol{\phi}(\boldsymbol{x}') \rangle$. The aim is to find a direction $\boldsymbol{\psi} = \sum_{i=1}^{l} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i)$ in the feature space $\mathcal{F}$ given by weights $\boldsymbol{\alpha} = [\alpha_1, \dots, \alpha_l]^T$ such that the criterion

$$F(\boldsymbol{\alpha}) = \frac{\langle \boldsymbol{\alpha} \cdot \mathbf{M} \boldsymbol{\alpha} \rangle}{\langle \boldsymbol{\alpha} \cdot (\mathbf{N} + \mu \mathbf{E}) \boldsymbol{\alpha} \rangle} = \frac{\langle \boldsymbol{\alpha} \cdot \boldsymbol{m} \rangle^2}{\langle \boldsymbol{\alpha} \cdot (\mathbf{N} + \mu \mathbf{E}) \boldsymbol{\alpha} \rangle}\,, \tag{5.13}$$

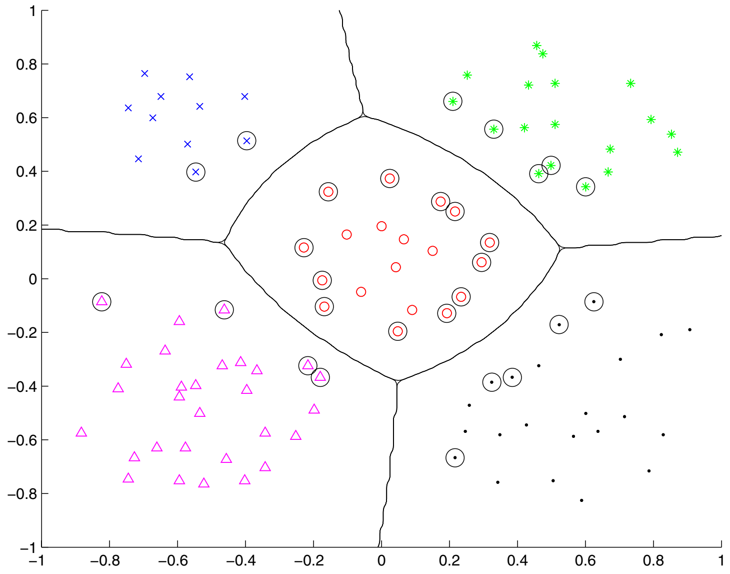


Figure 5.4: Example showing the multi-class BSVM classifier with $L2$-soft margin.

is maximized. The criterion (5.13) is the feature space counterpart of the criterion (2.13) defined in the input space $\mathcal{X}$. The vector $\boldsymbol{m}$ $[l \times 1]$, matrices $\mathbf{N}$ $[l \times l]$ $\mathbf{M}$ $[l \times l]$ are constructed as

$$
\begin{aligned}
\boldsymbol{m}_y &= \frac{1}{|\mathcal{I}_y|}\mathbf{K}\mathbf{1}_y\,, \quad y \in \mathcal{Y}\,, & \boldsymbol{m} &= \boldsymbol{m}_1 - \boldsymbol{m}_2\,, \\
\mathbf{N} &= \mathbf{K}\mathbf{K}^T - \sum_{y\in\mathcal{Y}} |\mathcal{I}_y| \boldsymbol{m}_y \boldsymbol{m}_y^T\,, & \mathbf{M} &= (\boldsymbol{m}_1 - \boldsymbol{m}_2)(\boldsymbol{m}_1 - \boldsymbol{m}_2)^T\,,
\end{aligned}
$$

where the vector $\mathbf{1}_y$ has entries $i \in \mathcal{I}_y$ equal to and remaining entries zeros. The kernel matrix $\mathbf{K}$ $[l \times l]$ contains kernel evaluations $K_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. The diagonal matrix $\mu\mathbf{E}$ in the denominator of the criterion (5.13) serves as the regularization term. The discriminant function $f(\boldsymbol{x})$ of the binary classifier (5.2) is given by the found vector $\boldsymbol{\alpha}$ as

$$
f(\boldsymbol{x}) = \langle \boldsymbol{\psi} \cdot \boldsymbol{\phi}(\boldsymbol{x}) \rangle + b = \langle \boldsymbol{\alpha} \cdot \boldsymbol{k}(\boldsymbol{x}) \rangle + b\,, \tag{5.14}
$$

where the $\boldsymbol{k}(\boldsymbol{x}) = [k(\boldsymbol{x}_1, \boldsymbol{x}), \ldots, k(\boldsymbol{x}_l, \boldsymbol{x})]^T$ is vector of evaluations of the kernel. The discriminant function (5.14) is known up to the bias b which is determined by the linear SVM with $L1$-soft margin is applied on the projected data $\mathcal{T}_{ZY} = \{(z_1, y_1), \ldots, (z_l, y_l)\}$. The projections $\{z_1, \ldots, z_l\}$ are computed as

$$
z_i = \langle \alpha \cdot \boldsymbol{k}(\boldsymbol{x}_i) \rangle\,.
$$

The KFD training algorithm is implemented in the function `kfd`.

References: The KFD formulation is described in paper [19]. The detailed analysis can be found in book [30].

Example: Kernel Fisher Discriminant

The binary classifier is trained by the KFD on the Riply's training set `riply_trn.mat`. The found classifier is evaluated on the testing data `riply_tst.mat`. The training data and the found kernel classifier is visualized in Figure 5.5. The classifier is visualized as the SVM classifier, i.e., the isolines with $f(\boldsymbol{x}) = +1$ and $f(\boldsymbol{x}) = -1$ are visualized and the support vectors (whole training set) are marked as well.

```
trn = load('riply_trn');  % load training data
options.ker = 'rbf';      % use RBF kernel
options.arg = 1;          % kernel argument
options.C = 10;           % regularization of linear SVM
options.mu = 0.001;       % regularization of KFD criterion
model = kfd(trn, options) % KFD training

% visualization
figure;
ppatterns(trn);
psvm(model);

% evaluation on testing data
tst = load('riply_tst');
ypred = svmclass( tst.X, model );
cerror( ypred, tst.y )
```

```
ans =

    0.0960
```

5.7 Pre-image problem

Let $\mathcal{T}_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ be a set of vector from the input space $\mathcal{X} \subseteq \mathbb{R}^n$. The kernel methods works with the data non-linearly mapped $\boldsymbol{\phi}\colon \mathcal{X} \to \mathcal{F}$ into a new feature space $\mathcal{F}$. The mapping is performed implicitly by using the kernel functions $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which are dot products of the input data mapped into the feature space $\mathcal{F}$ such that $k(\boldsymbol{x}_a, \boldsymbol{x}_b) = \langle \boldsymbol{\phi}(\boldsymbol{x}_a) \cdot \boldsymbol{\phi}(\boldsymbol{x}_b) \rangle$. Let $\boldsymbol{\psi} \in \mathcal{F}$ be given by the following kernel expansion

$$\boldsymbol{\psi} = \sum_{i=1}^{l} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i) , \tag{5.15}$$

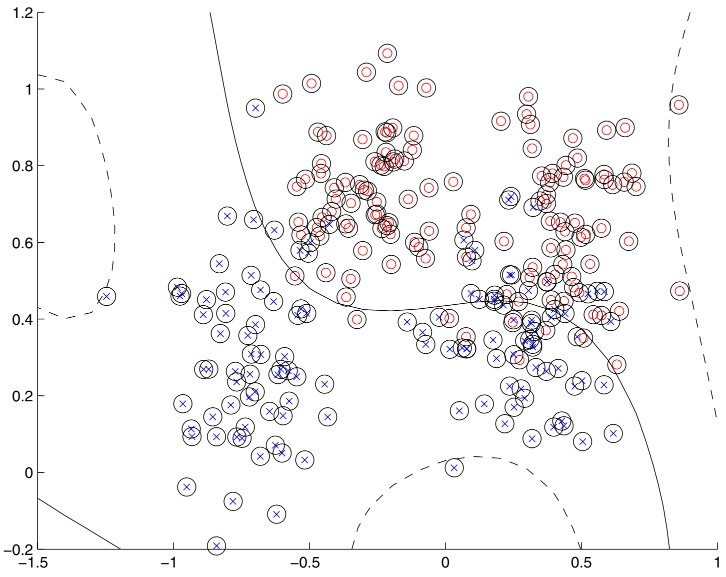


Figure 5.5: Example shows the binary classifier trained based on the Kernel Fisher Discriminant.

where $\boldsymbol{\alpha} = [\alpha_1, \dots, \alpha_l]^T$ is a weight vector of real coefficients. The pre-image problem aims to find an input vector $\boldsymbol{x} \in \mathcal{X}$, its feature space image $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathcal{F}$ would well approximate the expansion point $\Psi \in \mathcal{F}$ given by (5.15). In other words, the pre-image problem solves mapping from the feature space $\mathcal{F}$ back to the input space $\mathcal{X}$. The pre-image problem can be stated as the following optimization task

$$\begin{aligned}
\boldsymbol{x} &= \operatorname*{argmin}_{\boldsymbol{x}'} \|\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\psi}\|^2 \\
&= \operatorname*{argmin}_{\boldsymbol{x}'} \|\boldsymbol{\phi}(\boldsymbol{x}') - \sum_{i=1}^{l} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i)\|^2 \\
&= \operatorname*{argmin}_{\boldsymbol{x}'} k(\boldsymbol{x}', \boldsymbol{x}') - 2\sum_{i=1}^{l} \alpha_i k(\boldsymbol{x}', \boldsymbol{x}_i) + \sum_{i=1}^{l}\sum_{j=1}^{l} \alpha_i \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \,.
\end{aligned} \tag{5.16}$$

Let the Radial Basis Function (RBF) $k(\boldsymbol{x}_a, \boldsymbol{x}_b) = \exp(-0.5\|\boldsymbol{x}_a - \boldsymbol{x}_b\|^2/\sigma^2)$ be assumed. The pre-image problem (5.16) for the particular case of the RBF kernel leads to the following task

$$\boldsymbol{x} = \operatorname*{argmax}_{\boldsymbol{x}'} \sum_{i=1}^{l} \alpha_i \exp(-0.5\|\boldsymbol{x}' - \boldsymbol{x}_i\|^2/\sigma^2) \,. \tag{5.17}$$

The STPRtool provides three optimization algorithms to solve the RBF pre-image problem (5.17) which are enlisted in Table 5.6. However, all the methods are guaranteed to find only a local optimum of the task (5.17).

Table 5.6: Methods to solve the RBF pre-image problem implemented in the STPRtool.

`rbfpreimg`	An iterative fixed point algorithm proposed in [28].
`rbfpreimg2`	A standard gradient optimization method using the Matlab optimization toolbox for $1D$ search along the gradient.
`rbfpreimg3`	A method exploiting the correspondence between distances in the input space and the feature space proposed in [16].

References: The pre-image problem and the implemented algorithms for its solution are described in [28, 16].

Example: Pre-image problem

The example shows how to visualize a pre-image of the data mean in the feature induced by the RBF kernel. The result is visualized in Figure 5.6.

```
% load input data
data = load('riply_trn');

% define the mean in the feature space
num_data = size(data,2);
expansion.Alpha = ones(num_data,1)/num_data;
expansion.sv.X = data.X;
expansion.options.ker = 'rbf';
expansion.options.arg = 1;

% compute pre-image problem
x = rbfpreimg(expansion);

% visualization
figure;
ppatterns(data.X);
ppatterns(x,'or',13);
```

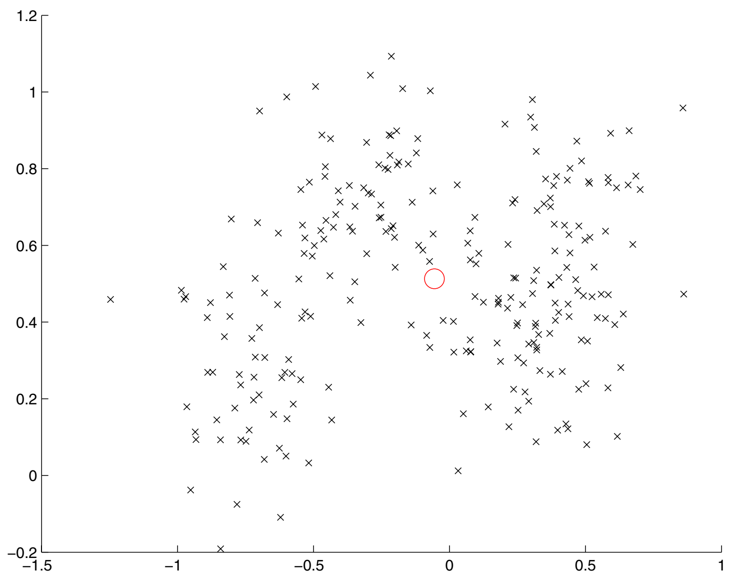


Figure 5.6: Example shows training data and the pre-image of their mean in the feature space induced by the RBF kernel.

5.8 Reduced set method

The reduced set method is useful to simplify the kernel classifier trained by the SVM or other kernel machines. Let the function $f\colon \mathcal{X} \subseteq \mathbb{R}^n \to \mathbb{R}$ be given as

$$f(\boldsymbol{x}) = \sum_{i=1}^{l} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}) + b = \langle \boldsymbol{\psi} \cdot \boldsymbol{\phi}(\boldsymbol{x}) \rangle + b \,, \tag{5.18}$$

where $\mathcal{T}_X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_l\}$ are vectors from $\mathbb{R}^n$, $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel function, $\boldsymbol{\alpha} = [\alpha_1, \dots, \alpha_l]^T$ is a weight vector of real coefficients and $b \in \mathbb{R}$ is a scalar. Let $\boldsymbol{\phi}\colon \mathcal{X} \to \mathcal{F}$ be a mapping from the input space $\mathcal{X}$ to the feature space $\mathcal{F}$ associated with kernel function such that $k(\boldsymbol{x}_a, \boldsymbol{x}_b) = \langle \boldsymbol{\phi}(\boldsymbol{x}_a) \cdot \boldsymbol{\phi}(\boldsymbol{x}_b) \rangle$. The function $f(\boldsymbol{x})$ can be expressed in the feature space $\mathcal{F}$ as the linear function $f(\boldsymbol{x}) = \langle \boldsymbol{\psi} \cdot \boldsymbol{\phi}(\boldsymbol{x}) \rangle + b$. The reduced set method aims to find a new function

$$\tilde{f}(\boldsymbol{x}) = \sum_{i=1}^{m} \beta_i k(\boldsymbol{s}_i, \boldsymbol{x}) + b = \langle \boldsymbol{\phi}(\boldsymbol{x}) \cdot \underbrace{\sum_{i=1}^{m} \beta_i \boldsymbol{\phi}(\boldsymbol{s}_i)}_{\tilde{\boldsymbol{\psi}}} \rangle + b \,, \tag{5.19}$$

such that the expansion (5.19) is shorter, i.e., $m < l$, and well approximates the original one (5.18). The weight vector $\boldsymbol{\beta} = [\beta_1, \dots, \beta_m]^T$ and the vectors $\mathcal{T}_S = \{\boldsymbol{s}_1, \dots, \boldsymbol{s}_m\}$, $\boldsymbol{s}_i \in \mathbb{R}^n$ determine the reduced kernel expansion (5.19). The problem of finding the reduced kernel expansion (5.19) can be stated as the optimization task

$$(\boldsymbol{\beta}, \mathcal{T}_S) = \operatorname*{argmin}_{\boldsymbol{\beta}', \mathcal{T}_S'} \|\tilde{\boldsymbol{\psi}} - \boldsymbol{\psi}\|^2 = \operatorname*{argmin}_{\boldsymbol{\beta}', \mathcal{T}_S'} \left\| \sum_{i=1}^{m} \beta_i' \boldsymbol{\phi}(\boldsymbol{s}_i') - \sum_{i=1}^{l} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}_i) \right\|^2 . \qquad (5.20)$$

Let the Radial Basis Function (RBF) $k(\boldsymbol{x}_a, \boldsymbol{x}_b) = \exp(-0.5\|\boldsymbol{x}_a - \boldsymbol{x}_b\|^2/\sigma^2)$ be assumed. In this case, the optimization task (5.20) can be solved by an iterative greedy algorithm. The algorithm converts the task (5.20) into a series of m pre-image tasks (see Section 5.7). The reduced set method for the RBF kernel is implemented in function `rsrbf`.
References: The implemented reduced set method is described in [28].

Example: Reduced set method for SVM classifier

The example shows how to use the reduced set method to simplify the SVM classifier. The SVM classifier is trained from the Riply's training set `riply_trn/mat`. The trained kernel expansion (discriminant function) is composed of 94 support vectors. The reduced expansion with 10 support vectors is found. Decision boundaries of the original and the reduced SVM are visualized in Figure 5.7. The original and reduced SVM are evaluated on the testing data `riply_tst.mat`.

```
% load training data
trn = load('riply_trn');

% train SVM classifier
model = smo(trn,struct('ker','rbf','arg',1,'C',10));

% compute the reduced rule with 10 support vectors
red_model = rsrbf(model,struct('nsv',10));

% visualize decision boundaries
figure; ppatterns(trn);
h1 = pboundary(model,struct('line_style','r'));
h2 = pboundary(red_model,struct('line_style','b'));
legend([h1(1) h2(1)],'Original SVM','Reduced SVM');

% evaluate SVM classifiers
tst = load('riply_tst');
ypred = svmclass(tst.X,model);
err = cerror(ypred,tst.y);

red_ypred = svmclass(tst.X,model);
```

```
  red_err = cerror(red_ypred,tst.y);

  fprintf(['Original SVM: nsv = %d, err = %.4f\n' ...
           'Reduced SVM: nsv = %d, err = %.4f\n'], ...
            model.nsv, err, red_model.nsv, red_err);

Original SVM: nsv = 94, err = 0.0960
Reduced SVM: nsv = 10, err = 0.0960
```

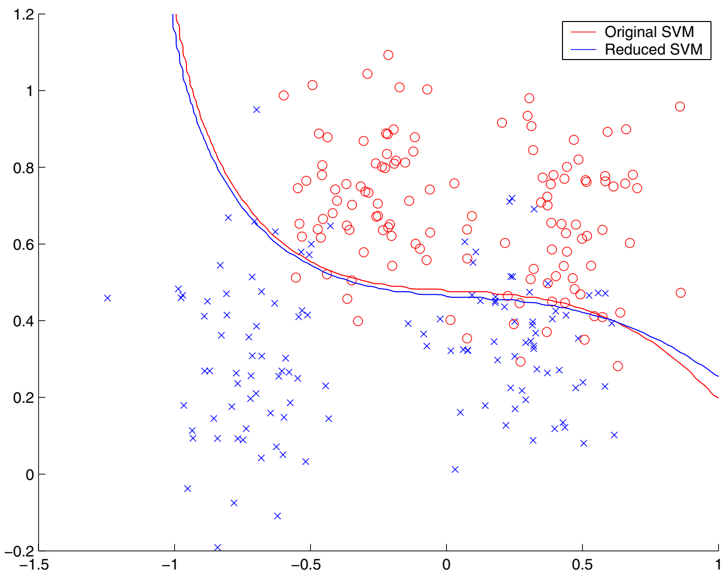


Figure 5.7: Example shows the decision boundaries of the SVM classifier trained by SMO and the approximated SVM classifier found by the reduced set method. The original decision rule involves 94 support vectors while the reduced one only 10 support vectors.

Chapter 6

Miscellaneous

6.1 Data sets

The STPRtool provides several datasets and functions to generate synthetic data. The datasets stored in the Matlab data file are enlisted in Table 6.1. The functions which generate synthetic data and scripts for converting external datasets to the Matlab are given in Table 6.2.

Table 6.1: Datasets stored in `/data/` directory of the STPRtool.

`/riply_trn.mat`	Training part of the Riply's data set [24].
`/riply_tst.mat`	Testing part of the Riply's data set [24].
`/iris.mat`	Well known Fisher's data set.
`/anderson_task/*.mat`	Input data for the demo on the Generalized Anderson's task `demo_anderson`.
`/binary_separable/*.mat`	Input data for the demo on training linear classifiers from finite vector sets `demo_linclass`.
`/gmm_samles/*.mat`	Input data for the demo on the Expectation-Maximization algorithm for the Gaussian mixture model `demo_emgmm`.
`/multi_separable/*.mat`	Linearly separable multi-class data in $2D$.
`/ocr_data/*.mat`	Examples of hand-written numerals. A description is given in Section 7.1.
`/svm_samples/*.mat`	Input data for the demo on the Support Vector Machines `demo_svm`.

Table 6.2: Data generation.

`createdata`	GUI which allows to create interactively labeled vectors sets and labeled sets of Gaussian distributions. It operates in 2-dimensional space only.
`genlsdata`	Generates linearly separable binary data with prescribed margin.
`gmmsamp`	Generates data from the Gaussian mixture model.
`gsamp`	Generates data from the multivariate Gaussian distribution.
`gencircledata`	Generates data on circle corrupted by the Gaussian noise.
`usps2mat`	Converts U.S. Postal Service (USPS) database of hand-written numerals to the Matlab file.

6.2 Visualization

The STPRtool provides tools for visualization of common models and data. The list of functions for visualization is given in Table 6.3. Each function contains an example of using.

Table 6.3: Functions for visualization.

`pandr`	Visualizes solution of the Generalized Anderson's task.
`pboundary`	Plots decision boundary of given classifier in 2D.
`pgauss`	Visualizes set of bivariate Gaussians.
`pgmm`	Visualizes the bivariate Gaussian mixture model.
`pkernelproj`	Plots isolines of kernel projection.
`plane3`	Plots plane in 3D.
`pline`	Plots line in 2D.
`ppatterns`	Visualizes patterns as points in the feature space. It works for 1D, 2D and 3D.
`psvm`	Plots decision boundary of binary SVM classifier in 2D.
`showim`	Displays images entered as column vectors.

6.3 Bayesian classifier

The object under study is assumed to be described by a vector of observations $\boldsymbol{x} \in \mathcal{X}$ and a hidden state $y \in \mathcal{Y}$. The $\boldsymbol{x}$ and y are realizations of random variables with joint probability distribution $P_{XY}(\boldsymbol{x}, y)$. A decision rule $q\colon \mathcal{X} \to \mathcal{D}$ takes a decision $d \in \mathcal{D}$ based on the observation $\boldsymbol{x} \in \mathcal{X}$. Let $W\colon \mathcal{D} \times \mathcal{Y} \to \mathbb{R}$ be a loss function which penalizes the decision $q(\boldsymbol{x}) \in \mathcal{D}$ when the true hidden state is $y \in \mathcal{Y}$. Let $\mathcal{X} \subseteq \mathbb{R}^n$ and the sets $\mathcal{Y}$ and $\mathcal{D}$ be finite. The Bayesian risk $R(q)$ is an expectation of the value of the loss function W when the decision rule q is applied, i.e.,

$$R(q) = \int_{\mathcal{X}} \sum_{y \in \mathcal{Y}} P_{XY}(\boldsymbol{x}, y) W(q(\boldsymbol{x}), y)\, \boldsymbol{dx}\, . \tag{6.1}$$

The optimal rule q^* which minimizes the Bayesian risk (6.1) is referred to as the Bayesian rule

$$q^*(\boldsymbol{x}) = \operatorname*{argmin}_{y} \sum_{y \in \mathcal{Y}} P_{XY}(\boldsymbol{x}, y) W(q(\boldsymbol{x}), y)\, , \qquad \forall \boldsymbol{x} \in \mathcal{X}\, .$$

The STPRtool implements the Bayesian rule for two particular cases:

Minimization of misclassification: The set of decisions $\mathcal{D}$ coincides to the set of hidden states $\mathcal{Y} = \{1, \ldots, c\}$. The 0/1-loss function

$$W_{0/1}(q(\boldsymbol{x}), y) = \begin{cases} 0 & \text{for} \quad q(\boldsymbol{x}) = y\, , \\ 1 & \text{for} \quad q(\boldsymbol{x}) \neq y\, . \end{cases} \tag{6.2}$$

is used. The Bayesian risk (6.1) with the 0/1-loss function corresponds to the expectation of misclassification. The rule $q\colon \mathcal{X} \to \mathcal{Y}$ which minimizes the expectation of misclassification is defined as

$$\begin{aligned} q(\boldsymbol{x}) &= \operatorname*{argmax}_{y \in \mathcal{Y}} P_{Y|X}(y|\boldsymbol{x})\, , \\ &= \operatorname*{argmax}_{y \in \mathcal{Y}} P_{X|Y}(\boldsymbol{x}|y) P_Y(y)\, . \end{aligned} \tag{6.3}$$

Classification with reject-option: The set of decisions $\mathcal{D}$ is assumed to be $\mathcal{D} = \mathcal{Y} \cup \{\text{dont\_know}\}$. The loss function is defined as

$$W_\varepsilon(q(\boldsymbol{x}), y) = \begin{cases} 0 & \text{for} \quad q(\boldsymbol{x}) = y\, , \\ 1 & \text{for} \quad q(\boldsymbol{x}) \neq y\, , \\ \varepsilon & \text{for} \quad q(\boldsymbol{x}) = \text{dont\_know}\, , \end{cases} \tag{6.4}$$

where ε is penalty for the decision dont_know. The rule $q\colon \mathcal{X} \to \mathcal{Y}$ which minimizes the Bayesian risk with the loss function (6.4) is defined as

$$q(\boldsymbol{x}) = \begin{cases} \operatorname*{argmax}_{y \in \mathcal{Y}} P_{X|Y}(\boldsymbol{x}|y) P_Y(y) & \text{if} \quad 1 - \max_{y \in \mathcal{Y}} P_{Y|X}(y|\boldsymbol{x}) < \varepsilon\, , \\ \text{dont\_know} & \text{if} \quad 1 - \max_{y \in \mathcal{Y}} P_{Y|X}(y|\boldsymbol{x}) \geq \varepsilon\, . \end{cases} \tag{6.5}$$

To apply the optimal classification rules one has to know the class-conditional distributions $P_{X|Y}$ and a priory distribution P_Y (or their estimates). The data-type used by the STPRtool is described in Table 6.4. The Bayesian classifiers (6.3) and (6.5) are implemented in function `bayescls`.

Table 6.4: Data-type used to describe the Bayesian classifier.

Bayesian classifier (structure array):	
`.Pclass` [*cell* $1 \times c$]	Class conditional distributions $P_{X\|Y}$. The distribution $P_{X\|Y}(\boldsymbol{x}\|y)$ is given by `Pclassy` which uses the data-type described in Table 4.2.
`.Prior` [$1 \times c$]	Discrete distribution $P_Y(y)$, $y \in \mathcal{Y}$.
`.fun = 'bayescls'`	Identifies function associated with this data type.

References: The Bayesian decision making with applications to PR is treated in details in book [26].

Example: Bayesian classifier

The example shows how to design the plug-in Bayesian classifier for the Riply's training set `riply_trn.mat`. The class-conditional distributions $P_{X|Y}$ are modeled by the Gaussian mixture models (GMM) with two components. The GMM are estimated by the EM algorithm `emgmm`. The a priori probabilities $P_Y(y)$, $y \in \{1, 2\}$ are estimated by the relative occurrences (it corresponds to the ML estimate). The designed Bayesian classifier is evaluated on the testing data `riply_tst.mat`.

```
% load input training data
trn = load('riply_trn');
inx1 = find(trn.y==1);
inx2 = find(trn.y==2);

% Estimation of class-conditional distributions by EM
bayes_model.Pclass{1} = emgmm(trn.X(:,inx1),struct('ncomp',2));
bayes_model.Pclass{2} = emgmm(trn.X(:,inx2),struct('ncomp',2));

% Estimation of priors
n1 = length(inx1); n2 = length(inx2);
bayes_model.Prior = [n1 n2]/(n1+n2);

%  Evaluation on testing data
tst = load('riply_tst');
ypred = bayescls(tst.X,bayes_model);
cerror(ypred,tst.y)
```

```
ans =

   0.0900
```

The following example shows how to visualize the decision boundary of the found Bayesian classifier minimizing the misclassification (solid black line). The Bayesian classifier splits the feature space into two regions corresponding to the first class and the second class. The decision boundary of the reject-options rule (dashed line) is displayed for comparison. The reject-option rule splits the feature space into three regions corresponding to the first class, the second class and the region in between corresponding to the dont_know decision. The visualization is given in Figure 6.1.

```
% Visualization
figure; hold on; ppatterns(trn);
bayes_model.fun = 'bayescls';
pboundary(bayes_model);

% Penalization for don't know decision
reject_model = bayes_model;
reject_model.eps = 0.1;

% Vislualization of rejet-option rule
pboundary(reject_model,struct('line_style','k--'));
```

6.4 Quadratic classifier

The quadratic classification rule $q: \mathcal{X} \subseteq \mathbb{R}^n \rightarrow \mathcal{Y} = \{1, 2, \ldots, c\}$ is composed of a set of discriminant functions

$$f_y(\boldsymbol{x}) = \langle \boldsymbol{x} \cdot \mathbf{A}_y \boldsymbol{x} \rangle + \langle \boldsymbol{b}_y \cdot \boldsymbol{x} \rangle + c_y \,, \qquad \forall y \in \mathcal{Y} \,,$$

which are quadratic with respect to the input vector $\boldsymbol{x} \in \mathbb{R}^n$. The quadratic discriminant function f_y is determined by a matrix $\mathbf{A}_y$ $[n \times n]$, a vector $\boldsymbol{b}_y$ $[n \times 1]$ and a scalar c_y $[1 \times 1]$. The input vector $\boldsymbol{x} \in \mathbb{R}^n$ is assigned to the class $y \in \mathcal{Y}$ its corresponding discriminant function f_y attains maximal value

$$y = \operatorname*{argmax}_{y' \in \mathcal{Y}} f_{y'}(\boldsymbol{x}) = \operatorname*{argmax}_{y' \in \mathcal{Y}} \left(\langle \boldsymbol{x} \cdot \mathbf{A}_y \boldsymbol{x} \rangle + \langle \boldsymbol{b}_y \cdot \boldsymbol{x} \rangle + c_y \right) \,. \tag{6.6}$$

In the particular binary case $\mathcal{Y} = \{1, 2\}$, the quadratic classifier is represented by a single discriminant function

$$f(\boldsymbol{x}) = \langle \boldsymbol{x} \cdot \mathbf{A} \boldsymbol{x} \rangle + \langle \boldsymbol{b} \cdot \boldsymbol{x} \rangle + c \,,$$

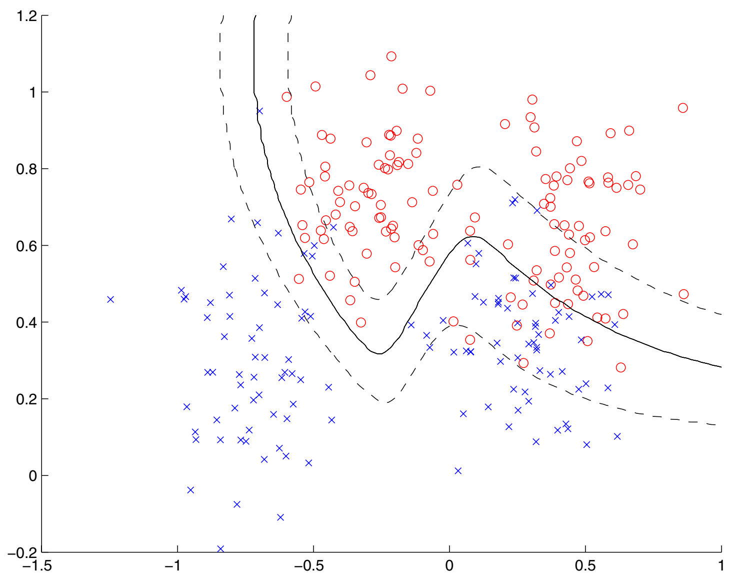


Figure 6.1: Figure shows the decision boundary of the Bayesian classifier (solid line) and the decision boundary of the reject-option rule with $\varepsilon = 0.1$ (dashed line). The class-conditional probabilities are modeled by the Gaussian mixture models with two components estimated by the EM algorithm.

given by a matrix $\mathbf{A}$ $[n \times n]$, a vector $\boldsymbol{b}$ $[n \times 1]$ and a scalar c $[1 \times 1]$. The input vector $\boldsymbol{x} \in \mathbb{R}^n$ is assigned to class $y \in \{1, 2\}$ as follows

$$q(\boldsymbol{x}) = \begin{cases} 1 & \text{if} \quad f(\boldsymbol{x}) = \langle \boldsymbol{x} \cdot \mathbf{A}\boldsymbol{x} \rangle + \langle \boldsymbol{b} \cdot \boldsymbol{x} \rangle + c \geq 0 \,, \\ 2 & \text{if} \quad f(\boldsymbol{x}) = \langle \boldsymbol{x} \cdot \mathbf{A}\boldsymbol{x} \rangle + \langle \boldsymbol{b} \cdot \boldsymbol{x} \rangle + c < 0 \,. \end{cases} \tag{6.7}$$

The STPRtool uses a specific structure array to describe the binary (see Table 6.5) and the multi-class (see Table 6.6) quadratic classifier. The implementation of the quadratic classifier itself provides function `quadclass`.

Table 6.5: Data-type used to describe binary quadratic classifier.

Binary linear quadratic (structure array):	
`.A` $[n \times n]$	Parameter matrix $\mathbf{A}$.
`.b` $[n \times 1]$	Parameter vector $\boldsymbol{b}$.
`.c` $[1 \times 1]$	Parameter scalar c.
`.fun = 'quadclass'`	Identifies function associated with this data type.

Example: Quadratic classifier

The example shows how to train the quadratic classifier using the Fisher Linear Discriminant and the quadratic data mapping. The classifier is trained from the Riply's

Table 6.6: Data-type used to describe multi-class quadratic classifier.

Multi-class quadratic classifier (structure array):	
`.A` $[n \times n \times c]$	Parameter matrices $\mathbf{A}_y$, $y \in \mathcal{Y}$.
`.b` $[n \times c]$	Parameter vectors b_y, $y \in \mathcal{Y}$.
`.c` $[1 \times c]$	Parameter scalars c_y, $y \in \mathcal{Y}$.
`.fun = 'quadclass'`	Identifies function associated with this data type.

training set `riply_trn.mat`. The found quadratic classifier is evaluated on the testing data `riply_trn.mat`. The boundary of the quadratic classifier is visualized (see Figure 6.2).

```
  trn = load('riply_trn');          % load input data
  quad_data = qmap(trn);            % quadratic mapping
  lin_model = fld(quad_data);       % train FLD

  % make quadratic classifier
  quad_model = lin2quad(lin_model);

  % visualization
  figure;
  ppatterns(trn); pboundary(quad_model);

  % evaluation
  tst = load('riply_tst');
  ypred = quadclass(tst.X,quad_model);
  cerror(ypred,tst.y)

ans =

    0.0940
```

6.5 K-nearest neighbors classifier

Let $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ be a set of prototype vectors $\boldsymbol{x}_i \in \mathcal{X} \subseteq \mathbb{R}^n$ and corresponding hidden states $y_i \in \mathcal{Y} = \{1, \ldots, c\}$. Let $\mathbb{R}^n(\boldsymbol{x}) = \{\boldsymbol{x}' \colon \|\boldsymbol{x} - \boldsymbol{x}'\| \leq r^2\}$ be a ball centered in the vector $\boldsymbol{x}$ in which lie K prototype vectors $\boldsymbol{x}_i$, $i \in \{1, \ldots, l\}$, i.e., $|\{\boldsymbol{x}_i \colon \boldsymbol{x}_i \in \mathbb{R}^n(\boldsymbol{x})\}| = K$. The K-nearest neighbor (KNN) classification rule $q \colon \mathcal{X} \to \mathcal{Y}$ is defined as

$$q(\boldsymbol{x}) = \operatorname*{argmax}_{y \in \mathcal{Y}} v(\boldsymbol{x}, y) \,, \tag{6.8}$$

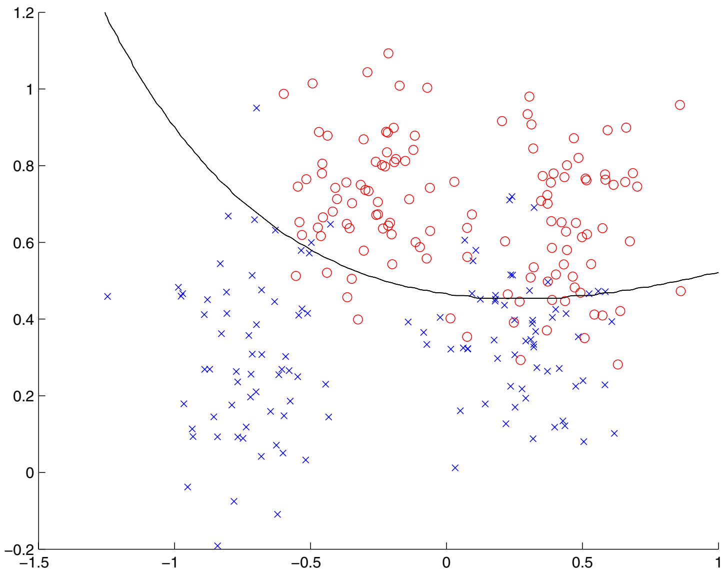


Figure 6.2: Figure shows the decision boundary of the quadratic classifier.

where $v(\boldsymbol{x}, y)$ is number of prototype vectors $\boldsymbol{x}_i$ with hidden state $y_i = y$ which lie in the ball $\boldsymbol{x}_i \in \mathbb{R}^n(\boldsymbol{x})$. The classification rule (6.8) is computationally demanding if the set of prototype vectors is large. The data-type used by the STPRtool to represent the K-nearest neighbor classifier is described in Table 6.7.

Table 6.7: Data-type used to describe the K-nearest neighbor classifier.

K-NN classifier (structure array):	
`.X` $[n \times l]$	Prototype vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$.
`.y` $[1 \times l]$	Labels $\{y_1, \ldots, y_l\}$.
`.fun = 'knnclass'`	Identifies function associated with this data type.

References: The K-nearest neighbors classifier is described in most books on PR, for instance, in book [6].

Example: K-nearest neighbor classifier

The example shows how to create the ($K = 8$)-NN classification rule from the Riply's training data `riply_trn.mat`. The classifier is evaluated on the testing data `riply_tst.mat`. The decision boundary is visualized in Figure 6.3.

```
% load training data and setup 8-NN rule
trn = load('riply_trn');
model = knnrule(trn,8);
```

```
   % visualize decision boundary and training data
   figure; ppatterns(trn); pboundary(model);

   % evaluate classifier
   tst = load('riply_tst');
   ypred = knnclass(tst.X,model);
   cerror(ypred,tst.y)

ans =

    0.1160
```

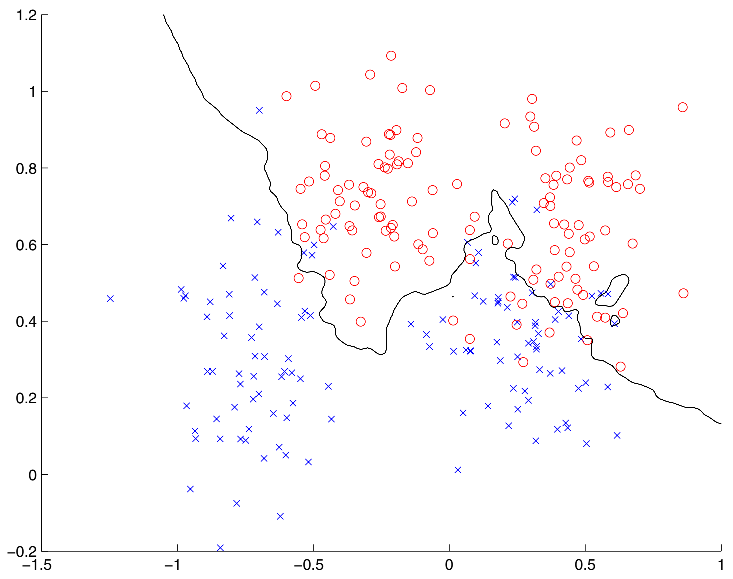


Figure 6.3: Figure shows the decision boundary of the $(K = 8)$-nearest neighbor classifier.

Chapter 7

Examples of applications

We intend to demonstrate how methods implemented in the STPRtool can be used to solve a more complex applications. The applications selected are the optical character recognition (OCR) system based on the multi-class SVM (Section 7.1) and image denoising system using the kernel PCA (Section 7.2).

7.1 Optical Character Recognition system

The use of the STPRtool is demonstrated on a simple Optical Character Recognition (OCR) system for handwritten numerals from 0 to 9. The STPRtool provides GUI which allows to use standard computer mouse as an input device to draw images of numerals. The GUI is intended just for demonstration purposes and the scanner or other device would be used in practice instead. The multi-class SVM are used as the base classifier of the numerals. A numeral to be classified is represented as a vector $\boldsymbol{x}$ containing pixel values taken from the numeral image. The training stage of the OCR system involves (i) collection of training examples of numerals and (ii) training the multi-class SVM. To see the trained OCR system run the demo script `demo_ocr`.

The GUI for drawing numerals is implemented in a function `mpaper`. The function `mpaper` creates a form composed of 50 cells to which the numerals can be drawn (see Figure 7.1a). The drawn numerals are normalized to size 16×16 pixels (different size can be specified). The function `mpaper` is designed to invoke a specified function whenever the middle mouse button is pressed. In particular, the functions `save_chars` and `ocr_fun` are called from the `mpaper` to process drawn numerals. The function `save_chars` saves drawn numerals in the stage of collecting training examples and the function `ocr_fun` is used to recognize the numerals if the trained OCR is applied.

The set of training examples has to be collected before the OCR is trained. The function `collect_chars` invokes the form for drawing numerals and allows to save the numerals to a specified file. The STPRtool contains examples of numerals collected from 6 different persons. Each person drew 50 examples of each of numerals from `'0'` to `'9'`. The numerals are stored in the directory `/data/ocr_numerals/`. The file name format `name_xx.mat` (the character `_` can be omitted from the name) is used. The string `name` is name of the person who drew the particular numeral and `xx` stands for a label assigned to the numeral. The labels from $y = 1$ to $y = 9$ corresponds to the numerals from `'1'` to `'9'` and the label $y = 10$ is used for numeral `'0'`. For example, to collect examples of numeral `'1'` from person `honza_cech`

```
collect_chars('honza_cech1')
```

was called. Figure 7.1a shows the form filled with the examples of the numeral `'1'`. The Figure 7.1b shows numerals after normalization which are saved to the specified file `honza_cech.mat`. The images of normalized numerals are stored as vectors to a matrix $\mathbf{X}$ of size $[256 \times 50]$. The vector dimension $n = 256$ corresponds to 16×16 pixels of each image numeral and $l = 50$ is the maximal number of numerals drawn to a single form. If the file name format `name_xx.mat` is met then the function `mergesets(Directory,DataFile)` can be used to merge all the file to a single one. The resulting file contains a matrix `X` of all examples and a vector `y` containing labels are assigned to examples according the number `xx`. The string `Directory` specifies the directory where the files `name_xx.mat` are stored and the string `FileName` is a file name to which the resulting training set is stored.

Having the labeled training set $\mathcal{T}_{XY} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l)\}$ created the multi-class SVM classifier can be trained. The methods to train the multi-class SVM are described in Chapter 5. In particular, the One-Against-One (OAO) decomposition was used as it requires the shortest training time. However, if a speed of classification is the major objective the multi-class BSVM formulation is often a better option as it produce less number of support vectors. The SVM has two free parameters which have to be tuned: the regularization constant C and the kernel function $k(\boldsymbol{x}_a, \boldsymbol{x}_b)$ with its argument. The RBF kernel $k(\boldsymbol{x}_a, \boldsymbol{x}_b) = \exp(-0.5\|\boldsymbol{x}_a - \boldsymbol{x}_b\|^2/\sigma^2)$ determined by the parameter σ was used in the experiment. Therefore, the free SVM parameters are C and σ. A common method to tune the free parameters is to use the cross-validation to select the best parameters from a pre-selected set $\Theta = \{(C_1, \sigma_1), \ldots, (C_k, \sigma_k)\}$. The STPRtool contains a function `evalsvm` which evaluates the set of SVM parameters Θ based on the cross-validation estimation of the classification error. The function `evalsvm` returns the SVM model its parameters (C^*, σ^*) turned out to be have the lowest cross-validation error over the given set Θ. The parameter tuning is implemented in the script `tune_ocr`. The parameters $C = \infty$ (data are separable) and $\sigma = 5$ were found to be the best.

After the parameters (C^*, σ^*) are tuned the SVM classifier is trained using all training data available. The training procedure for OCR classifier is implemented in the function `train_ocr`. The directory `/demo/ocr/models/` contains the SVM model trained by OAO decomposition strategy (function `oaosvm`) and model trained based on the multi-class BSVM formulation (function `bsvm2`).

The particular model used by OCR is specified in the function `ocr_fun` which implicitly uses the file `ocrmodel.mat` stored in the directory `/demo/ocr/`. To run the resulting OCR type `demo_ocr` or call

```
mpaper(struct('fun','ocr_fun'))
```

If one intends to modify the classifier or the way how the classification results are displayed then modify the function `ocr_fun`. A snapshot of running OCR is shown in Figure 7.2.

7.2 Image denoising

The aim is to train a model of a given class of images in order to restore images corrupted by noise. The U.S. Postal Service (USPS) database of images of handwritten numerals [17] was used as the class of images to model. The additive Gaussian noise was assumed as the source of image degradation. The kernel PCA was used to model the image class. This method was proposed in [20] and further developed in [15].

The STPRtool contains a script which converts the USPS database to the Matlab data file. It is implemented in the function `usps2mat`. The corruption of the USPS images by the Gaussian noise is implemented in script `make_noisy_data`. The script `make_noisy_data` produces the Matlab file `usps_noisy.mat` its contains is summarized in Table 7.1.

The aim is to train a model of images of all numerals at ones. Thus the unlabeled training data set $\mathcal{T}_X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\}$ is used. The model should describe the non-linear subspace $\mathcal{X}_{img} \subseteq \mathbb{R}^n$ where the images live. It is assumed that the non-linear subspace $\mathcal{X}_{img}$ forms a linear subspace $\mathcal{F}_{img}$ of a new feature space $\mathcal{F}$. A link between the input space $\mathcal{X}$ and the feature space $\mathcal{F}$ is given by a mapping $\boldsymbol{\phi}\colon \mathcal{X} \rightarrow \mathcal{F}$ which is implicitly determined by a selected kernel function $k\colon \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$. The kernel function corresponds to the dot product in the feature space, i.e., $k(\boldsymbol{x}_a, \boldsymbol{x}_b) = \langle \boldsymbol{\phi}(\boldsymbol{x}_a) \cdot \boldsymbol{\phi}(\boldsymbol{x}_b) \rangle$. The kernel PCA can be used to find the basis vectors of the subspace $\mathcal{F}_{img}$ from the input training data $\mathcal{T}_X$. Having the kernel PCA model of the subspace $\mathcal{F}_{img}$ it can be used to map the images $\boldsymbol{x} \in \mathcal{X}$ into the subspace $\mathcal{X}_{img}$. A noisy image $\hat{\boldsymbol{x}}$ is assumed to lie outside the subspace $\mathcal{X}_{img}$. The denoising procedure is based on mapping the noisy image $\hat{\boldsymbol{x}}$ into the subspace $\mathcal{X}_{img}$. The idea of using the kernel PCA for image denoising is demonstrated in Figure 7.3. The example which produced the figure is implemented in `demo_kpcadenois`.

The subspace $\mathcal{F}_{img}$ is modeled by the kernel PCA trained from the set $\mathcal{T}_X$ mapped into the feature space $\mathcal{F}$ by a function $\boldsymbol{\phi}\colon \mathcal{X} \rightarrow \mathcal{F}$. Let $\mathcal{T}_\Phi = \{\boldsymbol{\phi}(\boldsymbol{x}_1), \ldots, \boldsymbol{\phi}(\boldsymbol{x}_l)\}$ denote the feature space representation of the training images $\mathcal{T}_X$. The kernel PCA computes the lower dimensional representation $\mathcal{T}_Z = \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_l\}$, $\boldsymbol{z}_i \in \mathbb{R}^m$ of the mapped images

Table 7.1: The content of the file `usps_noisy.mat` produced by the script `make_noisy_data`.

Structure array `trn` containing the training data:	
`gnd_X` [256×7291]	Training images of numerals represented as column vectors. The input are 7291 gray-scale images of size 16×16.
`X` [256×7291]	Training images corrupted by the Gaussian noise with signal-to-noise ration SNR $= 1$ (it can be changed).
`y` [1×7291]	Labels $y \in \mathcal{Y} = \{1, \ldots, 10\}$ of training images.

Structure array `tst` containing the testing data:	
`gnd_X` [256×2007]	Testing images of numerals represented as column vectors. The input are 2007 gray-scale images of size 16×16.
`X` [256×2007]	Training images corrupted by the Gaussian noise with signal-to-noise ration SNR $= 1$ (it can be changed).
`y` [1×2007]	Labels $y \in \mathcal{Y} = \{1, \ldots, 10\}$ of training images.

$\mathcal{T}_\Phi$ to minimize the mean square reconstruction error

$$\varepsilon_{KMS} = \sum_{i=1}^{l} \|\boldsymbol{\phi}(\boldsymbol{x}_i) - \tilde{\boldsymbol{\phi}}(\boldsymbol{x}_i)\|^2 \,,$$

where

$$\tilde{\boldsymbol{\phi}}(\boldsymbol{x}) = \sum_{i=1}^{l} \beta_i \boldsymbol{\phi}(\boldsymbol{x}) \,, \quad \boldsymbol{\beta} = \mathbf{A}(\boldsymbol{z} - \boldsymbol{b}) \,, \quad \boldsymbol{z} = \mathbf{A}^T \boldsymbol{k}(\boldsymbol{x}) + \boldsymbol{b} \,. \tag{7.1}$$

The vector $k(\boldsymbol{x}) = [k(\boldsymbol{x}_1, \boldsymbol{x}), \ldots, k(\boldsymbol{x}_l, \boldsymbol{x})]^T$ contains kernel functions evaluated at the training data $\mathcal{T}_X$. The matrix $\mathbf{A}$ $[l \times m]$ and the vector $\boldsymbol{b}$ $[m \times 1]$ are trained by the kernel PCA algorithm. Given a projection $\tilde{\boldsymbol{\phi}}(\boldsymbol{x})$ onto the kernel PCA subspace an image $\boldsymbol{x}' \in \mathbb{R}^n$ of the input space can be recovered by solving the pre-image problem

$$\boldsymbol{x}' = \operatorname*{argmin}_{\boldsymbol{x}} \|\boldsymbol{\phi}(\boldsymbol{x}) - \tilde{\boldsymbol{\phi}}(\boldsymbol{x})\|^2 \,. \tag{7.2}$$

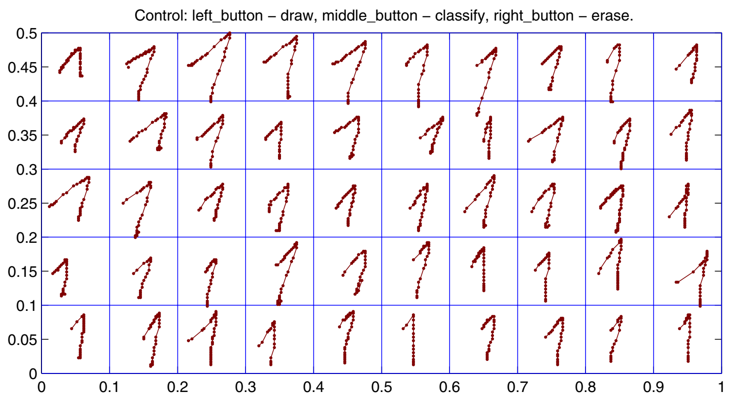
The Radial Basis Function (RBF) kernel $k(\boldsymbol{x}_a, \boldsymbol{x}_b) = \exp(-0.5\|\boldsymbol{x}_a - \boldsymbol{x}_b\|^2/\sigma^2)$ was used here as the pre-image problem (7.2) can be well solved for this case. The whole procedure of reconstructing the corrupted image $\hat{\boldsymbol{x}} \in \mathbb{R}^n$ involves (i) using (7.1) to compute $\boldsymbol{z}$ which determines $\tilde{\boldsymbol{\phi}}(\boldsymbol{x})$ and (ii) solving the pre-image problem (7.2) which yields the resulting $\boldsymbol{x}'$. This procedure is implemented in function `kpcarec`.

In contrast to the linear PCA, the kernel PCA allows to model the non-linearity which is very likely in the case of images. However, the non-linearity hidden in the parameter σ of the used RBF kernel has to be tuned properly to prevent over-fitting. Another parameter is the dimension m of the subspace kernel PCA subspace to be trained. The best parameters (σ^*, m^*) was selected out of a prescribed set $\Theta = \{(\sigma_1, m_1), \dots, (\sigma_k, m_k)\}$ such that the cross-validation estimate of the mean square error

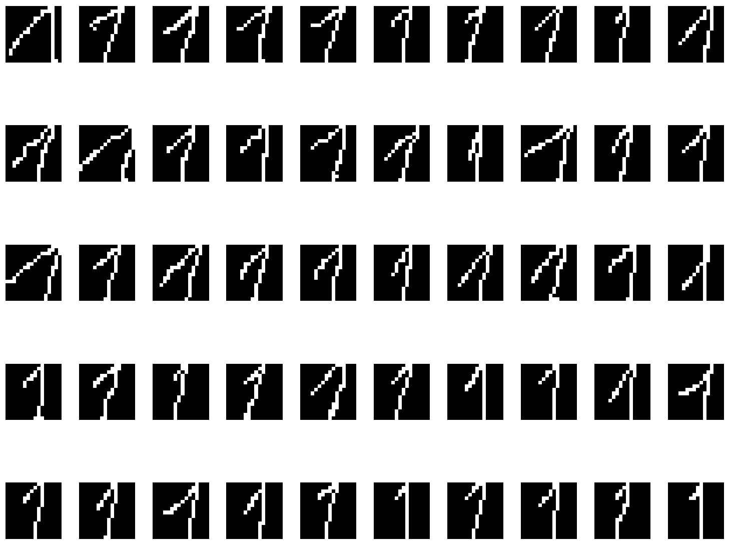
$$\varepsilon_{MS}(\sigma, m) = \int_{\mathcal{X}} P_X(\boldsymbol{x}) \left(\|\boldsymbol{x} - \boldsymbol{x}'\|^2\right) \boldsymbol{dx} \,,$$

was minimal. The $\boldsymbol{x}$ stands for the ground truth image and $\boldsymbol{x}'$ denotes the image restored from noisy one $\hat{\boldsymbol{x}}$ using (7.1) and (7.2). The images are assumed to be generated from distribution $P_X(\boldsymbol{x})$. Figure 7.4 shows the idea of tuning the parameters of the kernel PCA. The linear PCA was used to solve the same task for comparison. The same idea was used to tune the output dimension of the linear PCA. Having the parameters tuned the resulting kernel PCA and linear PCA are trained on the whole training set. The whole training procedure of the kernel PCA is implemented in the script `train_kpca_denois`. The training for linear PCA is implemented in the script `train_lin_denois`. The greedy kernel PCA Algorithm 3.4 is used for training the model. It produces sparse model and can deal with large data sets. The results of tuning the parameters are displayed in Figure 7.5. The figure was produced by a script `show_denois_tuning`.

The function `kpcarec` is used to denoise images using the kernel PCA model. In the case of the linear PCA, the function `pcarec` was used. An example of denoising of the testing data of USPS is implemented in script `show_denoising`. Figure 7.6 shows examples of ground truth images, noisy images, images restored by the linear PCA and kernel PCA.

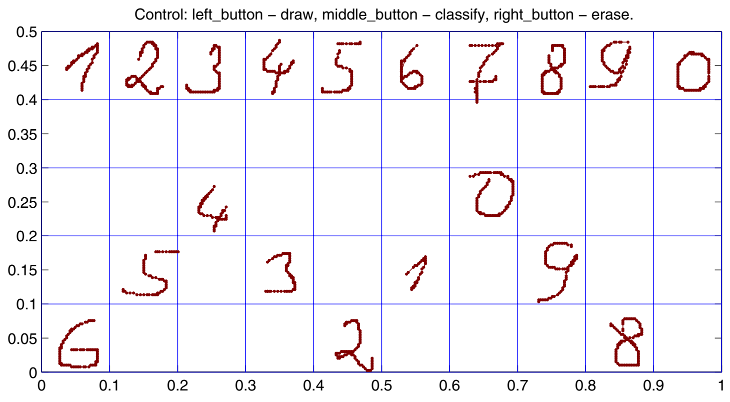


(a)

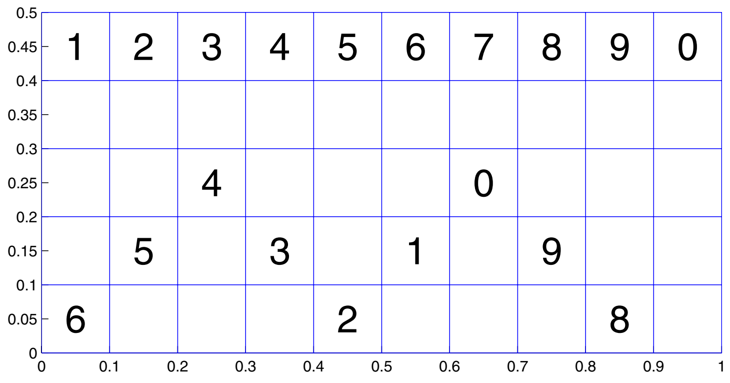


(b)

Figure 7.1: Figure (a) shows hand-written training examples of numeral `'1'` and Figure (b) shows corresponding normalized images of size 16 $\times$ 16 pixels.


Control: left_button – draw, middle_button – classify, right_button – erase.

(a)



(b)

Figure 7.2: A snapshot of running OCR system. Figure (a) shows hand-written numerals and Figure (b) shows recognized numerals.

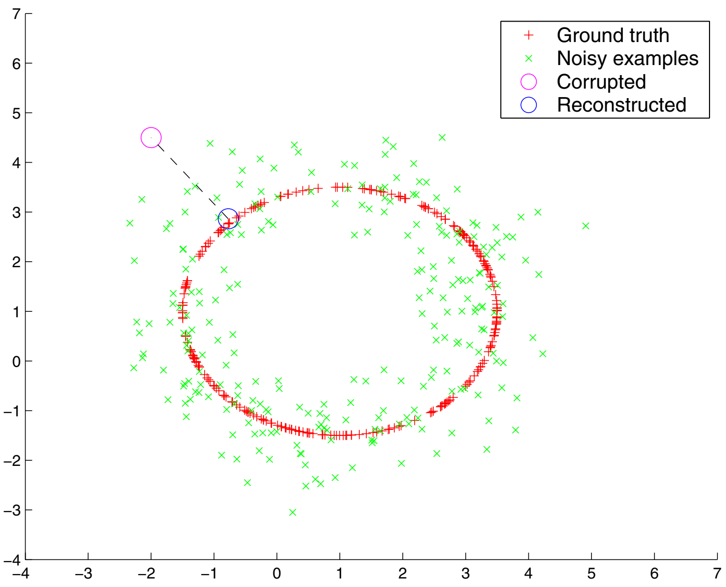


Figure 7.3: Demonstration of idea of image denoising by kernel PCA.

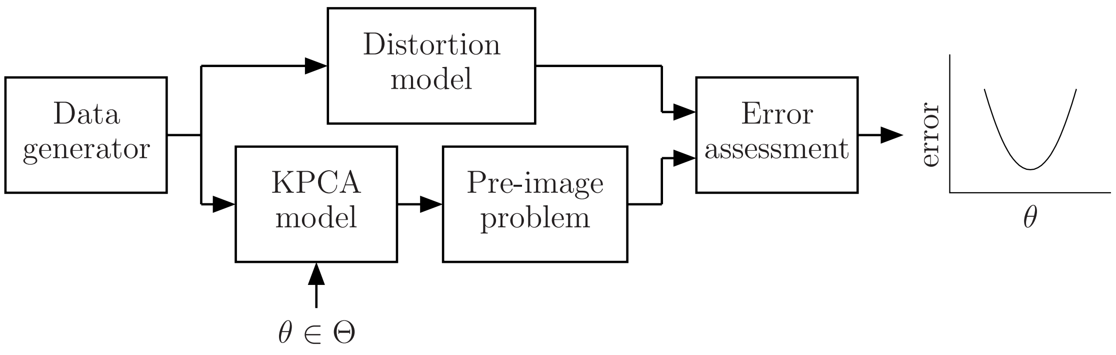
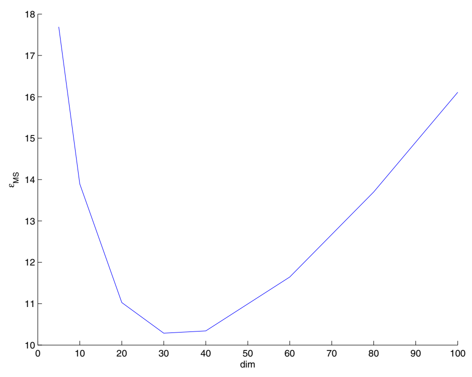
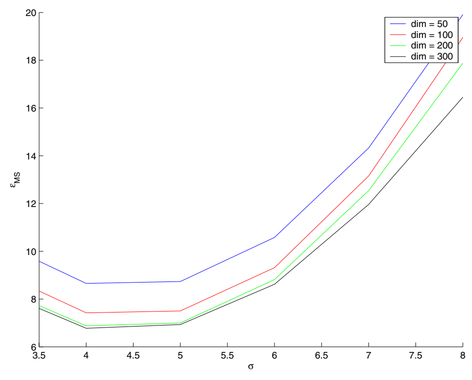


Figure 7.4: Schema of tuning parameters of kernel PCA model used for image restoration.

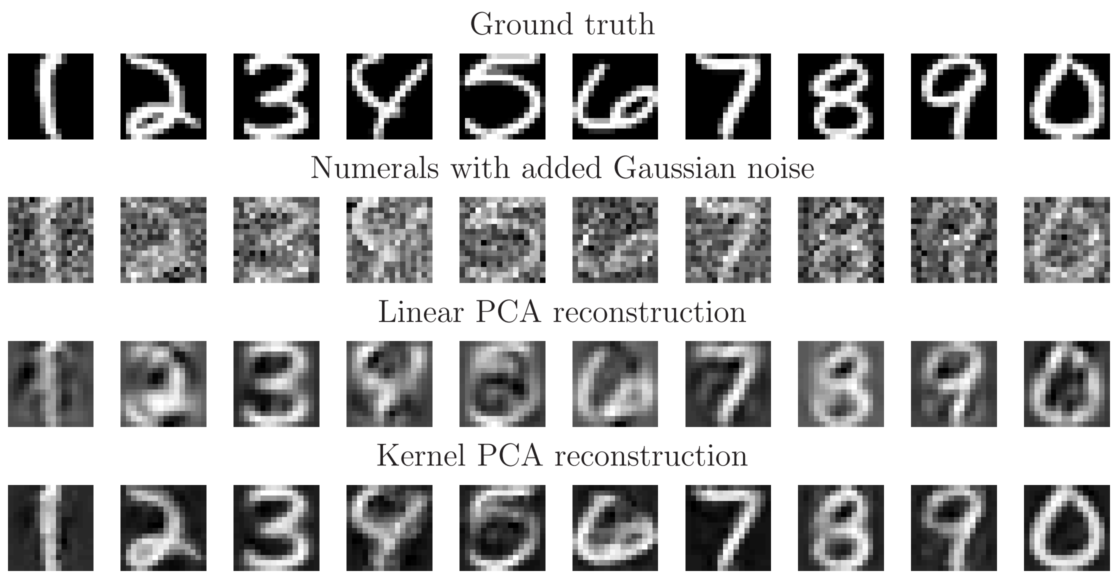


(a)



(b)

Figure 7.5: Figure (a) shows plot of ε_{MS} with respect to output dimension of linear PCA. Figure(b) shows plot of ε_{MS} with respect to output dimension and kernel argument of the kernel PCA.



Ground truth
Numerals with added Gaussian noise
Linear PCA reconstruction
Kernel PCA reconstruction

Figure 7.6: The results of denosing the handwritten images of USPS database.

Chapter 8

Conclusions and future planes

The STPRtool as well as its documentation is being updating. Therefore we are grateful for any comment, suggestion or other feedback which would help in the development of the STPRtool.

There are many relevant methods and proposals which have not been implemented yet. We welcome anyone who likes to contribute to the STPRtool in any way. Please send us an email and your potential contribution.

The future planes involve the extensions of the toolbox by the following methods:

- Feature selection methods.
- AdaBoost and related learning methods.
- Efficient optimizers to for large scale Support Vector Machines learning problems.
- Model selection methods for SVM.
- Probabilistic Principal Component Analysis (PPCA) and mixture of PPCA estimated by the Expectation-Maximization (EM) algorithm.
- Discrete probability models estimated by the EM algorithm.

Bibliography

[1] T.W. Anderson and R.R. Bahadur. Classification into two multivariate normal distributions with differrentia covariance matrices. *Anals of Mathematical Statistics*, 33:420–431, June 1962.

[2] G. Baudat and F. Anouar. Generalized discriminant analysis using a kernel approach. *Neural Computation*, 12(10):2385–2404, 2000.

[3] C.M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, Great Britain, 3th edition, 1997.

[4] N. Cristianini and J. Shawe-Taylor. *Support Vector Machines*. Cambridge University Press, 2000.

[5] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39:185–197, 1977.

[6] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. John Wiley & Sons, 2nd. edition, 2001.

[7] R.P.W. Duin. Prtools: A matlab toolbox for pattern recognition, 2000.

[8] V. Franc. Programové nástroje pro rozpoznávání (Pattern Recognition Programming Tools, In Czech). Master's thesis, České vysoké učení technické, Fakulta elektrotechnická, Katedra kybernetiky, February 2000.

[9] V. Franc and V. Hlaváč. Multi-class support vector machine. In R. Kasturi, D. Laurendeau, and Suen C., editors, *16th International Conference on Pattern Recognition*, volume 2, pages 236–239. IEEE Computer Society, 2002.

[10] V. Franc and V. Hlaváč. An iterative algorithm learning the maximal margin classifier. *Pattern Recognition*, 36(9):1985–1996, 2003.

[11] V. Franc and V. Hlaváč. Greedy algorithm for a training set reduction in the kernel methods. In N. Petkov and M.A. Westenberg, editors, *Computer Analysis of Images and Patterns*, pages 426–433, Berlin, Germany, 2003. Springer.

[12] C.W. Hsu and C.J. Lin. A comparison of methods for multiclass support vector machins. *IEEE Transactions on Neural Networks*, 13(2), March 2002.

[13] I.T. Jollife. *Principal Component Analysis*. Springer-Verlag, New York, 1986.

[14] S.S. Keerthi, S.K. Shevade, C. Bhattacharya, and K.R.K. Murthy. A fast iterative nearest point algorithm for support vector machine classifier design. *IEEE Transactions on Neural Networks*, 11(1):124–136, January 2000.

[15] Kim Kwang, In, Franz Matthias, O., and Schölkopf Bernhard. Kernel hebbian algorithm for single-fram super-resolution. In Leonardis Aleš and Bischof Horst, editors, *Statisical Learning in Computer Vision, ECCV Workshop*. Springer, May 2004.

[16] J.T. Kwok and I.W. Tsang. The pre-image problem in kernel methods. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, pages 408–415, Washington, D.C., USA, August 2003.

[17] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.J Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[18] G. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. John Wiley & Sons, New York, 1997.

[19] S. Mika, G. Rätsch, J. Weston, B. Schölkpf, and K. Müller. Fisher discriminant analysis with kernel. In Y.H. Hu, J. Larsen, and S. Wilson, E. Douglas, editors, *Neural Networks for Signal Processing*, pages 41–48. IEEE, 1999.

[20] S. Mika, B. Schölkopf, A. Smola, K.R. Müller, M. Scholz, and G. Rätsch. Kernel pca and de-noising in feature spaces. In M.S. Kearns, S.A. Solla, and D.A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 536 – 542, Cambridge, MA, 1999. MIT Press.

[21] I.T. Nabney. *NETLAB : algorithms for pattern recognition*. Advances in pattern recognition. Springer, London, 2002.

[22] J. Platt. Probabilities for sv machines. In A.J. Smola, P.J. Bartlett, B. Scholkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers (Neural Information Processing Series)*. MIT Press, 2000.

[23] J.C. Platt. Sequential minimal optimizer: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, Redmond, 1998.

[24] B.D. Riply. Neural networks and related methods for classification (with discusion). *J. Royal Statistical Soc. Series B*, 56:409–456, 1994.

[25] M.I. Schlesinger. A connection between learning and self-learning in the pattern recognition (in Russian). *Kibernetika*, 2:81–88, 1968.

[26] M.I. Schlesinger and V. Hlaváč. *Ten lectures on statistical and structural pattern recognition.* Kluwer Academic Publishers, 2002.

[27] B. Schölkopf, C. Burges, and A.J. Smola. *Advances in Kernel Methods – Support Vector Learning.* MIT Press, Cambridge, 1999.

[28] B. Schölkopf, P. Knirsch, and C. Smola, A. Burges. Fast approximation of support vector kernel expansions, and an interpretation of clustering as approximation in feature spaces. In P. Levi, M. Schanz, R.J. Ahler, and F. May, editors, *Mustererkennung 1998-20. DAGM.*, pages 124–132, Berlin, Germany, 1998. Springer-Verlag.

[29] B. Scholkopf, A. Smola, and K.R. Muller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.

[30] B. Schölkopf and A.J. Smola. *Learning with Kernels.* The MIT Press, MA, 2002.

[31] V. Vapnik. *The nature of statistical learning theory.* Springer Verlag, 1995.