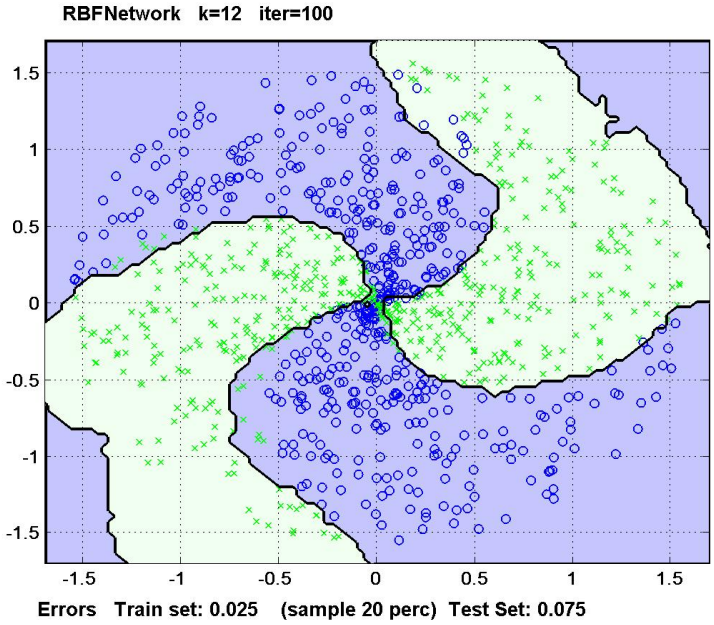
Anna Bartkowiak

NEURAL NETWORKS and PATTERN RECOGNITION

Lecture notes Fall 2004



Institute of Computer Science, University of Wrocław

Contents

{ plik roadmap.tex January 26, 2005}

Neural Networks and Pattern Recognition

Neural Networks and Pattern Recognition – Program

I 4.11.04. Elements of a Pattern Recognition System. Example: Fish: Salmon and Sea bass. Slides DHS Ch.1, Figures Ch.1.

II 11.10.04. Probability distributions when data vectors are coming from c classes $\omega_1, \ldots, \omega_c$: a priori $p(\omega_i)$, conditional on class ω_i, $p(\mathbf{x}|\omega_i)$, total $p(\mathbf{x})$, posterior $p(\omega_i|\mathbf{x})$.
Constructing decision bounds for classification into two classes.
Figs 2.1, 2.6, 2.7, 2.8, 2.9–2.14 from Duda [1]. Slides Ch.2. Bayesian Decision Theory. Part 1, no. 2–10 (priors and posteriors), Part 2, no. 13–15 (equations for normal p.d.f.), Part 3, no. 1–13.

III K–means. Illustration in slides by A. Moore.
Regression and classification by neural networks, slides by A. Moore

IV 25.X.04. Bayesian decision theory – continuous features. The concepts of Decision rule, decision boundary or decision surface. Bayes rule for minimum error, the reject (withhold option) – presentation based on the book by Webb [2].
Slides from the workshop 'Pattern Recognition' by Marchette and Solka [4]: Pattern recognition flowchart, pattern recognition steps, defining and extracting 'features'. Examples of problems: SALAD (ship ...), Artificial Nose, Network anomaly detection, Medical example (Pima). Bayesian decision theory for two classes, Probability densities, Bayesian risk. Definition of a classifier, evaluating and comparing classifiers (Re-substitution, Cross validation, Test sets).
Simple parametric classifiers, discriminant functions for normal densities.

V 8.XI.04. 22. XI.04. Nonparametric techniques I. Parzen windows. Density estimation by kernel methods. Probabilistic Neural Networks.

VI 22. XI.04. Nonparametric techniques II. Estimation using k-nearest-neighbor rule (k_n-N-N) and nearest-neighbor rule (N-N). Estimation of probabilities á posteriori for given $\mathbf{x}$. Choice of metric. Reducing computational complexity by computing partial distances, prestructuring (search tree), editing (pruning). The RCE (Reduced Coulomb Energy) network.

VII 29.11.04. Linear discriminant functions I. Ordinary and generalized linear discriminant functions based on polynomial and generally Phi functions. Example of bad discrimination in the observed data space and a perfect discrimination in the outer generalized data space. Augmented feature and weight vectors. Advantages when using the augmented feature and weight terminology. An example of the solution region for linear separable samples.

VIII 06.12.04. Linear discriminant functions II. Linearly separable case. Seeking for the weight vector: Basic gradient descent methods. The perceptron criterion function, error correcting procedures. Minimum squared-error procedures. Linear programming methods, the simplex algorithm.

IX 13.12.04. Support Vector Machines I. The separable case.

X 20.12.04. Support Vector Machines II. The case of non-separable samples. The SVM function from the MatLab 'Patt' toolbox [3]

XI 03.01.05. Software for SVM by Steve Gunn [5], Mixture models.

XII 10.01.05. Combining information from several classifiers, Mc Nemars' test, Bagging, Boosting.

XIII 17.01.05. Comparing and combining classifiers. AdaBoost, Mixture of Experts.

XIV 24.01.05. Graphical visualization of multivariate data. The proposal of Liao as extension of Kohonens' SOM. Curvilinear component analysis - as originated by Herault. Problems with high-dimensional data: the empty space phenomenon.

XV A survey of neural network algorithms and their implementation in the Classification Toolbox (PATT) by Stork and Elad Yom–Tov [3].

References

[1] Richard O. Duda, P.E. Hart, David G. Stork: Pattern Classification, 2nd Edition, Wiley 2001.

[2] Andrew Webb, Statistical Pattern Recognition, 2nd Edition. Wiley 2002, Reprint September 2004.

[3] David G. Stork and Elad Yom-Tov, Computer Manual in MATLAB to accompany Pattern Classification. Wiley Interscience, 2004, 136 pages. ISBN: 0-471-42977-5.

[4] David J. Marchette and Jeffrey L. Solks, Pattern Recognition. Naval Surface Warfare Center. Slides presented during the Interface 2002 at the Mason University.

[5] Steve Gunn, Support Vector Machines for Classification and Regression. ISIS Technical Report. 14 May 1998. Image, Speech and Intelligent Systems Group, Univ. of Southampton.

[6] Francesco Camastra. Kernel Methods for Computer Vision. Theory and Applications. citeseer.ist.psu.edu/483015.html

[7] Alessandro Verri, Support Vector Machines for classification. Slides 1–32.

[8] Massimiliano Pontil and Alessandro Verri, Support Vector Machines for 3-D Object Reconition. IEEE Trans. on Pattern Analysis and Machine Intelligence, V. 20, Nb. 6, 637–646, 1998. url citeseer.ist.psu.edu/pontil98support.html

[9] Corinna Cortes, Vladimir Vapnik, Support-vector networks.
Machine Learning (Kluwer) 20: 1–25, 1995.

[10] Netlab by Ian Nabney: Netlab neural network software, Neural Computing Research Group, Division of Electric Engineering and Computer Science, Aston University, Birmingham UK, http://www.ncrg.aston.ac.uk/

2 Bayesian Decision theory

2.1 Setting Decision Boundaries

Ω – measurement space in R^d
A decision rule partitions the measurement space into c disjoint regions: $\Omega = \Omega_1 \dots \Omega_c$.

If an observation vector is in Ω_i, then it is assumed as belonging to class ω_i, $(i = 1, \dots, c)$.

Each region Ω_i may be multiply connected – that is, it may be made up of several disjoint regions.

The boundaries between the regions Ω_i are the **decision boundaries** or **decision surfaces**.

Generally, it is in regions close to these boundaries that the highest proportion of misclassification occurs.

Our decision – whether a pattern belongs to class ω_i or not, is 'yes' or 'not'. However, when the classified pattern is near the classification boundary, we may *withhold* the decision until further information is available so that classification may be made later. If the option of 'withholding' is respected, then we have $c + 1$ outcomes.

2.2 Bayesian Decision Theory – continuous features

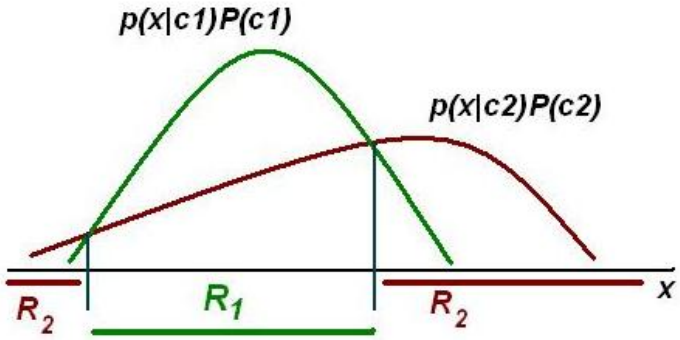


Figure 2.1: Decision regions for two classes build from the likelihood ratio. Region $\mathcal{R}_1$ is designated by those $\mathbf{x}$'s, for which the posteriors satisfy $p(\mathbf{x}|\omega_1)P(\omega_1) > p(\mathbf{x}|\omega_2)P(\omega_2)$. Class ω_1 is denoted by *c1*, class ω_2 by *c2*. File dis2.jpg

Making decision means assigning a given $\mathbf{x}$ to one of 2 classes. We may do it

- Using only *the prior* information:
  Decide ω_1 if $P(\omega_1) > P(\omega_2)$, otherwise decide ω_2.
- Using *class–conditional* information $p(\mathbf{x}|\omega_1)$ and $p(\mathbf{x}|\omega_2)$:
  Evaluate *posterior* evidence $j = 1, 2$
  $p(\mathbf{x}) = \sum_{j=1}^{2} p(\mathbf{x}|\omega_j)P(\omega_j)$
  $P(\omega_j|\mathbf{x}) = \frac{p(\mathbf{x}|\omega_j)P(\omega_j)}{p(\mathbf{x})}, \quad j = 1, 2$

Then look at the *likelihood ratio*. The inequality

$$l_r(\mathbf{x}) = \frac{P(\mathbf{x}|\omega_1)}{P(\mathbf{x}|\omega_2)} > \frac{P(\omega_2)}{P(\omega_1)}$$

implies that $\mathbf{x} \in$ class ω_1.

2.3 Bayes' rule for minimum error for c classes, the ROC curve

Consider $p(\mathbf{x}|\omega_j)P(\omega_j) > p(\mathbf{x}|\omega_k)P(\omega_k)$, $k = 1, \ldots, c, \quad k \neq j$

The probability of making an error, $P(error)$ may be expressed as

$$P(error) = \sum_{i=1}^{c} P(error|\omega_i) \cdot P(\omega_i), \tag{2.1}$$

with $P(error|\omega_i)$ denoting the probability of misclassifying patterns from class i. This probability can be expressed as

$$P(error|\omega_i) = \int_{\Omega - \Omega_i} p(\mathbf{x}|\omega_i) d\mathbf{x}. \tag{2.2}$$

Taking this into account, we obtain

$$\begin{aligned} P(error) &= \sum_{i=1}^{c} P(\omega_i) \int_{\Omega - \Omega_i} p(\mathbf{x}|\omega_i) d\mathbf{x} \\ &= \sum_{i=1}^{c} P(\omega_i) \Big(1 - \int_{\Omega_i} p(\mathbf{x}|\omega_i) d\mathbf{x}\Big) \\ &= 1 - \sum_{i=1}^{c} P(\omega_i) \int_{\Omega_i} p(\mathbf{x}|\omega_i) d\mathbf{x}, \end{aligned}$$

from which we see that minimizing the probability of making an error is equivalent to maximizing

$$\sum_{i=1}^{c} p(\omega_i) \int_{\Omega_i} P(\mathbf{x}|\omega_i) d\mathbf{x},$$

i.e. the probability of correct classification.

This is achieved by selecting Ω_i to be the region for which $P(\omega_i)p(\mathbf{x}|\omega_i)$ is the largest over all classes. Then the probability of correct classification, c, is

$$\mathsf{c} = \int_{\Omega} \max_i P(\omega_i) p(\mathbf{x}|\omega_i) d\mathbf{x},$$

and the Bayesian error, e_B, is

$$\mathsf{e}_B = 1 - \int_{\Omega} \max_i P(\omega_i) p(\mathbf{x}|\omega_i) d\mathbf{x}.$$

This is the smallest error we can get.

Establishing the decision boundary in a different way results in a larger error. An example of such behavior is considered below and shown in Fig. 2.2.

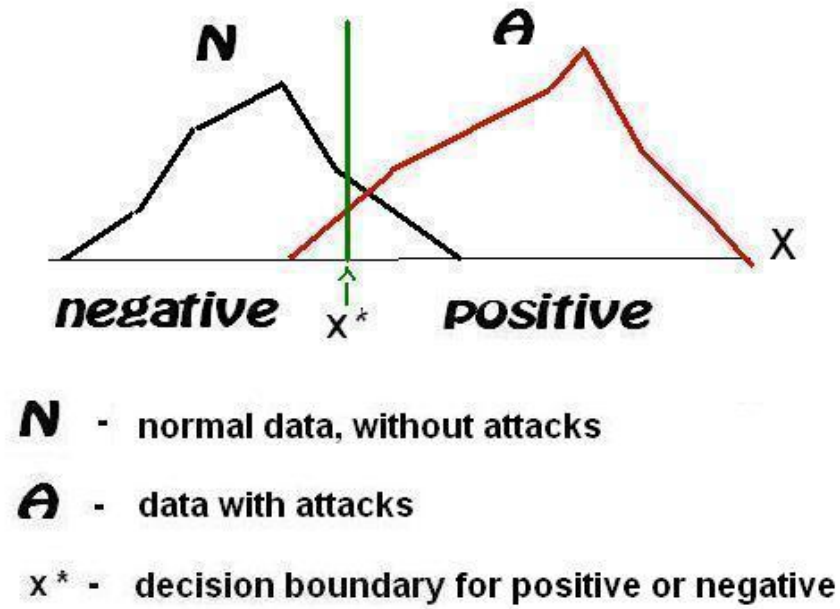


Figure 2.2: (Possible errors arising when fixing the decision boundary as the point $x*$. Some part of 'negatives' will appear as 'positive' (false positive'); some part of 'positives' will not be detected. The overall error could be reduced using the Bayesian formula. File roc2.jpg

Our goal might be focused on recognizing – in first place – events belonging to one class. Say, class 1 (ω_1) denotes computer attacks, or a specific dangerous disease that should be diagnosis. Then, in first place, we should properly recognize events from the 1st class and do not misclassify too much of them. Such a situation is presented in figure 2.2. Presence of computer attacks is indicated by an index x based on occurrence of some amplified calls of some functions working in the computer system. Generally speaking, the calls occur more frequently during attacks; thus the index x for the class 'attacks' has higher value. How to set the decision boundary x^* for deciding: do the given observed $\mathbf{x}$ represents an attack or just a normal functioning of the device?

Let us define the rate of **false positives** and **false negatives**.

We may draw also a *Relative Operating Characteristic (ROC)* curve.

The ROC curve has

as x-axis: probability of false alarm, $P(fp)$ (false positive), i.e. $P(x \geq x^*|\mathcal{N})$, meaning error of the 2nd kind;

as y-axis: probability of true alarm, $P(tp)$ (true positive), i.e. $P(x \geq x^*|\mathcal{A})$.

Thus the ROC curve is composed from points $(x = P(x \geq x^*|\mathcal{N}),\ P(x \geq x^*|\mathcal{A}))$ taken as function of the variable x^* constituting the decision boundary.

∞

4 Nonparametric methods for density estimation

When probability density functions are known, we know how to construct decision boundaries.

Parametric methods assume distributions of known shape functions, e.g. Gaussian, Gamma, Poisson.

Nonparametric methods estimate distribution functions directly from the data.

We will consider here probability density estimation based on kernel methods [1].

4.1 Data based density estimation for a current point x

Let $\mathbf{x} \in R^d$, $\mathrm{p}(\mathbf{x})$ – probability density function of $\mathbf{x}$,

Let $\mathcal{R} \subset R^d$ denote a region with volume V.

Let $P_{\mathcal{R}}$ denote the probability that a randomly chosen $\mathbf{x}$ falls into the region $\mathcal{R}$:

$$P_{\mathcal{R}} = P(\mathbf{x} \in \mathcal{R}) = \int_{\mathcal{R}} p(\mathbf{x})\mathbf{dx}. \tag{4.1}$$

If $\mathrm{p}(\mathbf{x})$ is continuous, and $\mathcal{R}$ is small – so that $\mathrm{p}(\mathbf{x})$ does not vary significantly within it – we can write for given $\mathbf{x}$ [2]

$$P(\mathbf{x} \in \mathcal{R}) = \int_{\mathcal{R}} p(\mathbf{x})\mathbf{dx} \cong p(\mathbf{x})\, V. \tag{4.2}$$

Suppose now, we have a training sample

$$\mathbf{x}_1, \ldots, \mathbf{x}_n.$$

If n is sufficiently large, an estimate of $P_{\mathcal{R}}$ is provided by the ratio k/n:

$$P_{\mathcal{R}} = P(\mathbf{x} \in \mathcal{R}) \approx \frac{k}{n} \tag{4.3}$$

where n is the size of the training sample, and k is the number of successes, i.e. the number of sample points falling into region $\mathcal{R}$.

Equating (4.2) and (4.3) we obtain an approximation of the probability density $p(\mathbf{x})$

$$p_n(\mathbf{x}) \cong \frac{k}{nV}. \tag{4.4}$$

Estimator (4.4) is biased, however consistent. These properties depend on the choice of $\mathcal{R}$ (should $\to 0$) and n (should $\to \infty$).

4.2 Parzen windows and kernel based density estimates

Let $\varphi(\mathbf{u})$ be the following window function defined on the unit hypercube $\mathcal{H}(\mathbf{0}, 1)$ centered at $\mathbf{0}$, parallel to the coordinate axes and with edges equal to 1:

$$\varphi(\mathbf{u}) = \begin{cases} 1\,, & |u_j| \le 1/2, \quad j = 1, \ldots, d \\ 0\,, & \text{otherwise} \end{cases} \tag{4.5}$$

[1] Notes based on Duda [1], Chapter 4

[2] We may do it, because

$$\int_{\mathcal{R}} p(\mathbf{dx})\mathbf{dx} \approx p(\mathbf{x}) \int_{\mathcal{R}} \mathbf{1dx} = p(\mathbf{x})\mu(\mathcal{R}) = p(\mathbf{x})V \quad \text{for some } \mathbf{x} \in \mathcal{R}.$$

For a current d-variate point $\mathbf{x}$ we define the hypercube $\mathcal{H}(\mathbf{x}, h)$ centered at $\mathbf{x}$, parallel to coordinate axes and with equal edges h. Its volume equals $V = h^d$.

Assume now, we have a training sample of patterns $\mathbf{x}_1, \ldots, \mathbf{x}_n$. We wish to state, which of these sample points fall into the hypercube $\mathcal{H}(\mathbf{x}, h)$. This may be done by substituting for each $\mathbf{x}_i$

$$\mathbf{u} = (\mathbf{x} - \mathbf{x}_i)/h), \quad i = 1, \ldots, n$$

and evaluating

$$\varphi(\mathbf{u}) = \varphi(\frac{\mathbf{x} - \mathbf{x}_i}{h}).$$

Clearly $\varphi((\mathbf{x} - \mathbf{x}_i)/h)$ is equal to unity, if $\mathbf{x}_i$ falls into the hypercube $\mathcal{H}(\mathbf{x},\, h)$.
The number of sample patterns in the hypercube $\mathcal{H}(\mathbf{x}, h)$ is simply:

$$k(\mathbf{x}) = \sum_{i=1}^{n} \varphi(\frac{\mathbf{x} - \mathbf{x}_i}{h}).$$

Substituting this into equation (4.4) we obtain

$$p_n(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{V}\, \varphi(\frac{\mathbf{x} - \mathbf{x}_i}{h}) \quad \text{with} \quad V = h^d\,. \tag{4.6}$$

The above equation expresses an estimate for $p(\mathbf{x})$ as proportional to the *average* of window functions $\varphi(\mathbf{u})$ evaluated at hypercubes $\mathcal{H}(\mathbf{x}, h)$ for subsequent elements of the available sample patterns $\mathbf{x}_1, \ldots, \mathbf{x}_n$. Each pattern contributes to the estimate in accordance with its distance from $\mathbf{x}$.

If h is big, then the estimator $p_n(\mathbf{x})$ is smooth, many sample points influence the value $p_n(\mathbf{x})$, the changes of $p_n(\mathbf{x})$ are slow with changing the argument $\mathbf{x}$.

On the other hand, if h is small, the density function $p_n(\mathbf{x})$ becomes 'jagged' (see Figures 4.1 and 4.2). With $h \to 0$, $p_n(\mathbf{x})$ approaches the average of Dirac's δ functions.

The window width h, and consequently, the volume $V = h^d$, play an essential role when evaluating the estimator $p_n(\mathbf{x})$ using formula (4.6). In practice, the parameter h may depend on the sample size.

The above approach uses d–dimensional hypercubes as the basic elements of reasoning. The window function φ represents *de facto* a uniform distribution.

It is possible (and used frequently in practice) that the window function may be any other function (e.g. Gaussian, Epanechnikov). Alternatively, the window function is called 'kernel function'. The kernel function may depend on some parameters, e.g. h, which in turn should generally depend on n, thus the notation $h = h_n$. One way of defining h_n is to assume h_1 equal to a known constant (e.g. $h_1 = 1$), and then take $h_n = h_1/\sqrt{n}$.

It is natural to ask that the estimate $p(\mathbf{x})$ be a legitimate density function (non-negative and integrate to one). This can be assured by requiring the window function itself be a density function.

In Figure 4.1 we show bivariate Gaussian kernels with window width $h = 1,\ 0.5,\ 0.01$ and the corresponding kernel density estimates based on a sample containing 5 points. Notice the impact (effect) of the kernel width h on the smoothness of the derived density estimate $p(\mathbf{x})$.

Further illustrations: Marchette [4], slides 24–51.

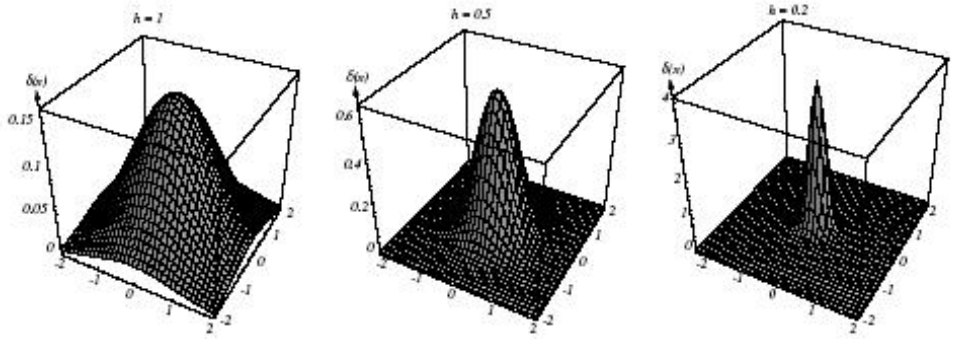


FIGURE 4.3. Examples of two-dimensional circularly symmetric normal Parzen windows for three different values of h. Note that because the $\delta(\mathbf{x})$ are normalized, different vertical scales must be used to show their structure. From: Richard O. Duda,

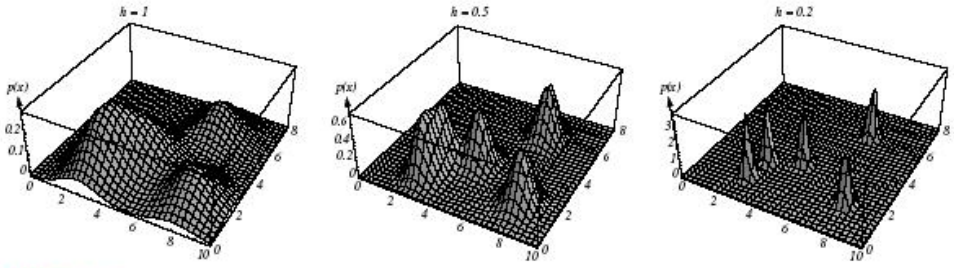


FIGURE 4.4. Three Parzen-window density estimates based on the same set of five samples, using the window functions in Fig. 4.3. As before, the vertical axes have been scaled to show the structure of each distribution.

Figure 4.1: **Top**: Bivariate spherical kernels with width values $h = 1,\ 0.5,\ 02$. **Bottom**: Density estimates constructed for 5 sample points. Files duda4_3.jpg and duda4_4.jpg

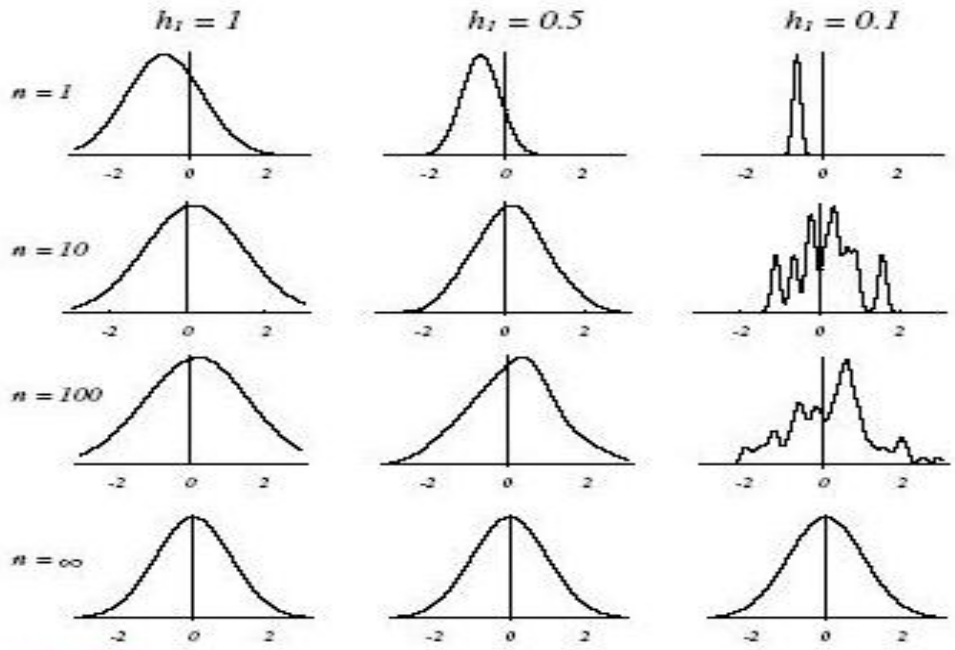


FIGURE 4.5. Parzen-window estimates of a univariate normal density using different window widths and numbers of samples.

Figure 4.2: Kernel density estimates constructed from samples of size $n = 1,\ 10,\ 100$ and ∞ using Gaussian kernels of width $h = h_1/\sqrt{n}$, with $h_1 = 1,\ 0.05$ and 0.1 appropriately. File duda4_5.jpg

In classification problems we estimate the densities for each category separately. Then we classify a test point by the label corresponding to the maximum posterior. Again, the decision boundaries may depend from the width parameter h, see Figure 4.3.

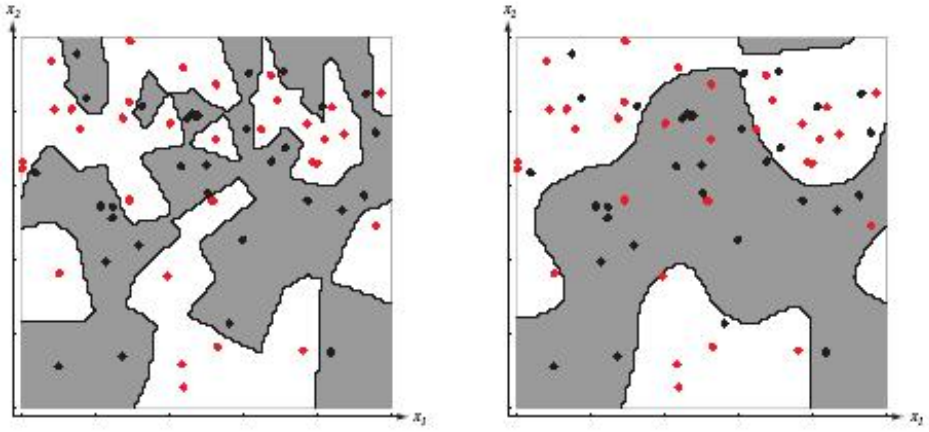


Figure 4.3: Decision boundaries for classification into two groups for 2–dimensional data. The boundaries were constructed using two-dimensional kernel-based densities with a smaller (left) and a larger (right) width parameter h. File duda4_8.jpg

The density estimation and classification examples illustrate some of the power and some of the limitations of non–parametric methods.
Power – lies in the generality of the procedures. E.g. the same procedure may be used for the unimodal normal case and the binomial mixture case. With enough samples, we are essentially assured of convergence to an arbitrarily complicated target distribution.
Limitations – In higher dimensions the number of samples should be large, much greater than would be required if we knew the form of the unknown density. The phenomenon is called 'Curse of dimensionality'. The only way to beat the curse is to incorporate knowledge about the data that is correct.

Below we cite a table illustrating the curse of dimensionality. The table is from Marchette [4], where it is quoted after Siverman, Density Estimation for Statistics and Data Analysis, 1986, Chapman & Hall.

To estimate the density at 0 with a given accuracy, the following sample sizes are needed:

Dimensionality	Required sample size
1	4
2	19
5	786
7	10 700
10	842 000

4.3 Probabilistic Neural Networks

Probabilistic Neural Networks may be viewed as a kind of application of kernel methods to density estimation in multi-category problem.

The neural network has 3 layers: input, hidden, and output.
The **input** layer has d neurons.
The **hidden** layer has n neurons, where n equals the number of training patterns.
The **output** layer has c neurons, representing c categories od data.
The network needs a parameter h denoting width of its working window.
Each data vector x should be normalized to have unit norm. This applies to data vectors belonging to training samples and test samples as well.

A scheme of the PNN is shown in Figure 4.4.

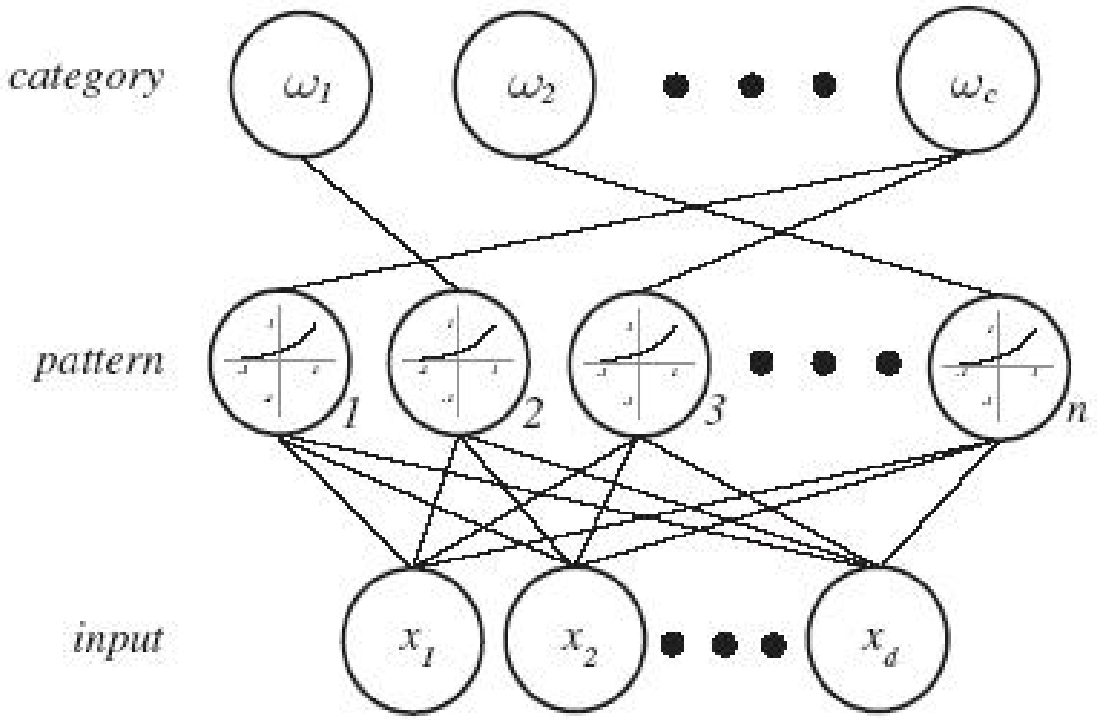


Figure 4.4: A PNN (Probabilistic Neural Network) realizing classification into c classes using Parzen-window algorithm. The network consists of d input units, n hidden units – as many as the number of training vectors – and c category output units. File duda4_9.jpg

Training phase. The training sample contains pairs $\{\mathbf{x}_i, y_i\}$, $i = 1, \ldots, n$, with y_i denoting the target, i.e. the no. of the class to which the pattern belongs. The number of hidden neurons is equal to the number of data vectors in the training sample.
The d-dimensial data vectors constituting the training sample are sequentially presented to the network. After presenting the ith pattern $\{\mathbf{x}_i, y_i\}$ only the i-th hidden neuron becomes active: its weight vector is set equal to the presented data vector: $\mathbf{w}_i \equiv \mathbf{x}_i$ (remember, $\|\mathbf{x}_1\| = 1$). The activated neuron sets his connection to the category indicated by y_i.
Other hidden neurons remain in-active.

Classification phase. It is assumed that the hidden neurons are activated with a Gaussian kernel function centered at $\mathbf{w}_k$, $k = 1, \ldots, n$ with width parameter $h = 2\sigma^2$.

The elements of the test sample are presented sequentially to the network. For a presented vector $\mathbf{x}$ the following actions are taken:

• All hidden neurons ($k = 1, \ldots, n$) become activated (according to the assumed Gaussian kernel function) using the formula:

$$\begin{aligned} \varphi((\mathbf{x} - \mathbf{w}_k)/h) &\propto exp\{-(\mathbf{x} - \mathbf{w}_k)^T(\mathbf{x} - \mathbf{w}_k)/2\sigma^2\} \\ &= exp\{-(\mathbf{x}^T\mathbf{x} + \mathbf{w}_k^T\mathbf{w}_k - 2\mathbf{x}^T\mathbf{w}_k)/2\sigma^2\} \\ &= exp\{(z_k - 1)/\sigma^2\} \end{aligned}$$

where $z_k = net_k = \mathbf{x}^T\mathbf{w}_k$. Notice that $z_k =< \mathbf{x}, \mathbf{w}_k >$ is the scalar product of $\mathbf{x} and \mathbf{w}_k$, i.e. of the normalized (!) data vector $\mathbf{x}$ and of the weight vector $\mathbf{w}_k$ ascribed to the kth hidden neuron. Thus, as such, z_k is contained in the interval $-1 \leq z_k \leq 1$, and it takes the values:
for $z_k =$ –1: $exp\{-2/\sigma^2\}$,
for $z_k =$ 0: $exp\{-1/\sigma^2\}$,
for $z_k =$ +1: 1.0.

Thus each hidden neuron emits to its associated output category unit a signal equal ?proportional? to the probability that the test point was generated by a Gaussian centered on the associated training point.

• The category units summarize the coming signals from their associated hidden neurons; each category unit for its class. In such a way for each class k we obtain the appropriate probability density function $g_k(\mathbf{x})$ – the Parzen-window estimate of the kth distribution. The $max_k\ g_k(\mathbf{x})$ operation returnes the desired category for the test point.

Limitation: Number of hidden neurons equal to number of training data vectors.

The computational complexity of the algorithm is $O(n)$ – it needs n inner products which may be done in parallel.

Applications in problems, where recognition speed is important and storage is not a severe limitation.

Another benefit: new training patterns can be incorporated into previously trained network quite easily; this may be important for on–line application.

4.4 k_n–nearest neighbor estimation

We aim at to build a p.d.f. $p(\mathbf{x})$ in a non-parametric way.

In considerations of the previous subsections we did it by inspecting the frequency of appearing of training samples in cells of fixed size. Now we will let the cell volume to be a function of the training data, rather than some arbitrary function of the overall number of samples.

To estimate $p(\mathbf{x})$ from n training samples (or prototypes) we center a cell about $\mathbf{x}$ and let it grow until it captures k_n samples. These samples are the k_n nearest–neighbors of $\mathbf{x}$. Then we take

$$p_n(\mathbf{x}) = \frac{k_n/n}{V_n}$$

and k_n/n will be a good estimate of the p-lity that a point will fall in the cell of volume V_n.

We could take $k_n = \sqrt{n}$.

Estimation of probabilities *a posteriori*

We place a cell of volume V around $\mathbf{x}$ and capture k samples, k_i of which turn out to be labelled ω_i. The the obvious estimate for the joint probability $p_n(\mathbf{x}, \omega_i)$ is

$$p_n(\mathbf{x}, \omega_i) = \frac{k_i/n}{V_n}.$$

Now we are able to calculate the posterior $P(\omega_i|\mathbf{x})$ as

$$P(\omega_i|\mathbf{x}) = \frac{p_n(\mathbf{x}, \omega_i)}{\sum_{j=1}^{c} p_n(\mathbf{x}, \omega_j)} = \frac{k_i}{k}.$$

Facts:

- We take as the estimate of $p(\mathbf{x})$ fraction of patterns (points) – within the cell centered at $\mathbf{x}$ – that are labelled ω_i.
- For a minimum error rate, the most frequently represented category within the cell is selected.
- If k is large and the cell sufficiently small, the performance will approach the best possible.

How to choose the cell

a) In Parzen-window approach V_n is a specified function of n, such as $V_n = 1/\sqrt{n}$.

b) In k_n-nearest neighbor approach V_n is expanded until some specified number of samples are captured, e.g. $k = \sqrt{n}$.

The nearest neighbor rule

is a special case of the k_n-nearest neighbors for $k = 1$. For a set labelled training patterns $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and given $\mathbf{x}$ we apply the following decision rule:

> Classify $\mathbf{x}$ to the class j, to which $\mathbf{x}_{(j)}$, the nearest labelled pattern belongs.

Facts

- Leads to an error rate greater than the minimum possible, i.e. the Bayes rate.
- If the number of training patterns is large (un limited), then the nearest neighbor classifier is never worse than twice the Bayesian rate.
- If $P(\omega_m|\mathbf{x}) \cong 1$, then the nearest neighbor selection is always the same as the Bayes selection.

In two dimensions, the Nearest Neighbor algorithm leads to a partitionning of the input space into **Voronoi** cells, each labelled by the category of the training point, it contains.

In three dimensons, the decision boundary resembles the surface of a cristal.

Issues of computational complexity

Computing each distance costs $O(d)$. Computing the whole table of distances for all pairs costs $O(dn^2)$. Therefore we try to reduce the general complexity by reducing the number of pairs, for which the evaluations are done. There are generally 3 groups of such methods:

1. Computing partial distances.
2. Pre-structuring the prototypes.
3. Editing.

ad **1)** We compute the distances only for $k < d$ dimensions. if they are greater than a fixed value, we do not continue to account for the remaining dimensions.

ad **2)** Say, we have 2–dimensional data positioned in a square. We may subdivide the square into 4 quadrants and designate in each quadrant one prototype (data point representative for that quadrant).

Then, for each query point $\mathbf{x}$, we find which prototype is nearest and evaluate only distances to points which have as representative the found prototype. In this way, 3/4 of the data points need not be queried, because the search is limited to the corresponding quadrant.

This is a tradeoff of search complexity against accuracy.

ad **3)** Removing points which have as neighbors Voronoi cells with prototypes belonging to the same class – speeds up alculations.

Other procedure: Remove points which are on the wrong sided of the decision boundaries for classifying into 2 classa.

4.5 The RCE – Reduced Coulomb Energy – network

An RCE network is topologically equivalent to the PNN network (see Section 4.3 and Figure 4.4). The scheme of the RCE network is presented in Figure 4.5. The data vectors should be adjusted to have unit norm. The weights of the n hidden vectors are set equal to the n normalized training patterns. Distances are calculated as inner products. Hidden neurons have also a modifiable threshold corresponding to a 'radius' λ. During training, each threshold is adjusted so that its radius is as large as possible without containing training patterns from a different category.

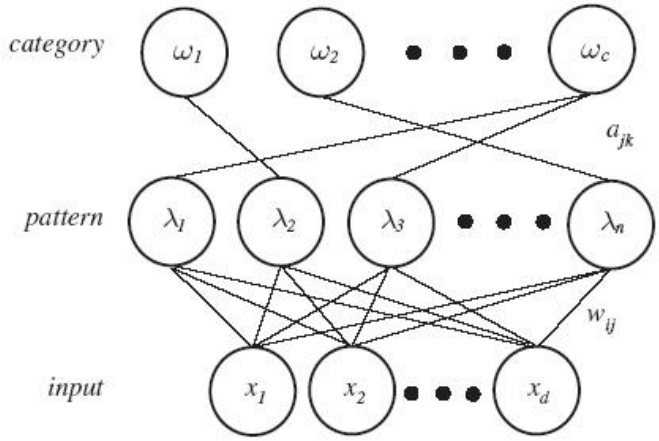


Figure 4.5: Structure of the RCE (Reduced Coulomb Energy) network. File duda4_25.jpg

The process of adjusting the radius-es is illustrated in Figure 4.6. The cells are colored by their category: grey denotes ω_1, pink ω_2, dark red indicates conflicting regions.

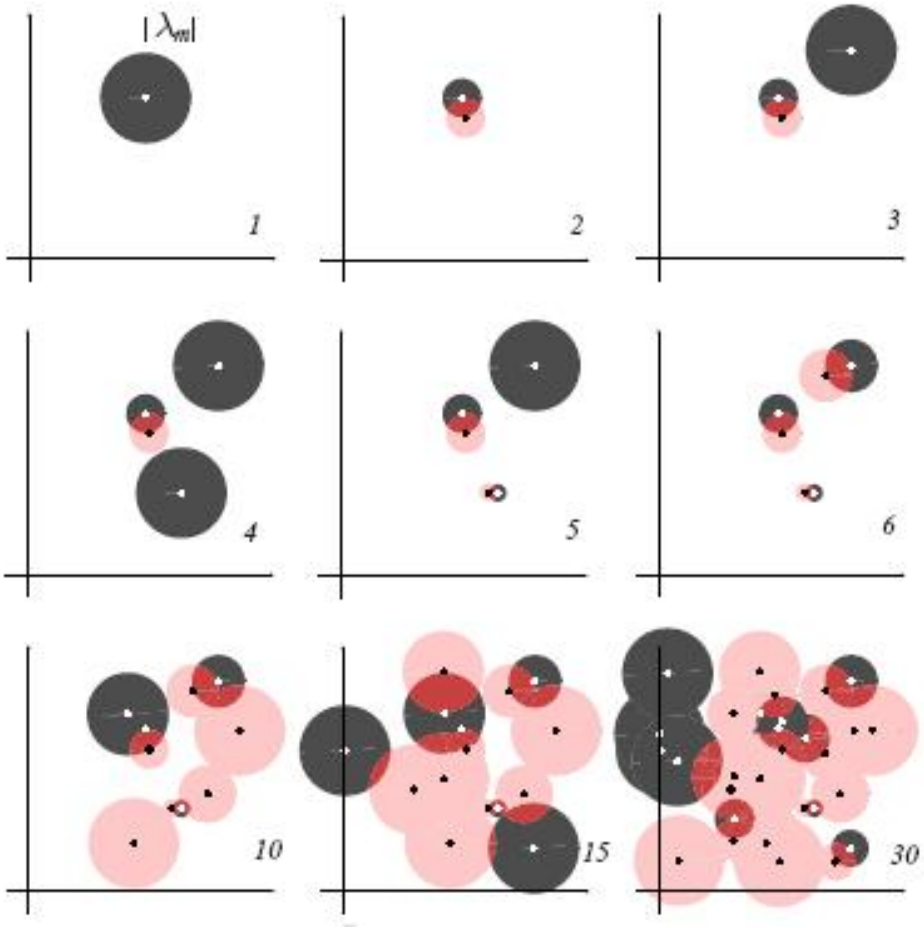


Figure 4.6: The RCE network, phase of training. Finding sequentially the radius parameter λ_i for each hidden neuron No. i. Grey cells indicate membership of class ω_1, pink cells – membership of class ω_2. Dark red indicates conflicting regions. File duda4_26.jpg

4.5.1 Software

- Store_Grabbag is a modification of the nearest-neighbor algorithm. The procedure identifies those samples in the training set that affect the classification and discards the others.

5 Linear Discriminant functions

5.1 Ordinary linear discriminant functions

Ordinary linear discriminant function has the form $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$.

Decision rule for $c = 2$:

Decide	ω_1,	if $g(\mathbf{x}) > 0$,
	ω_2,	if $g(\mathbf{x}) < 0$.

Thus positive values of $g(\mathbf{x})$ indicate class ω_1, and negative values of $g(\mathbf{x})$ indicate ω_2.

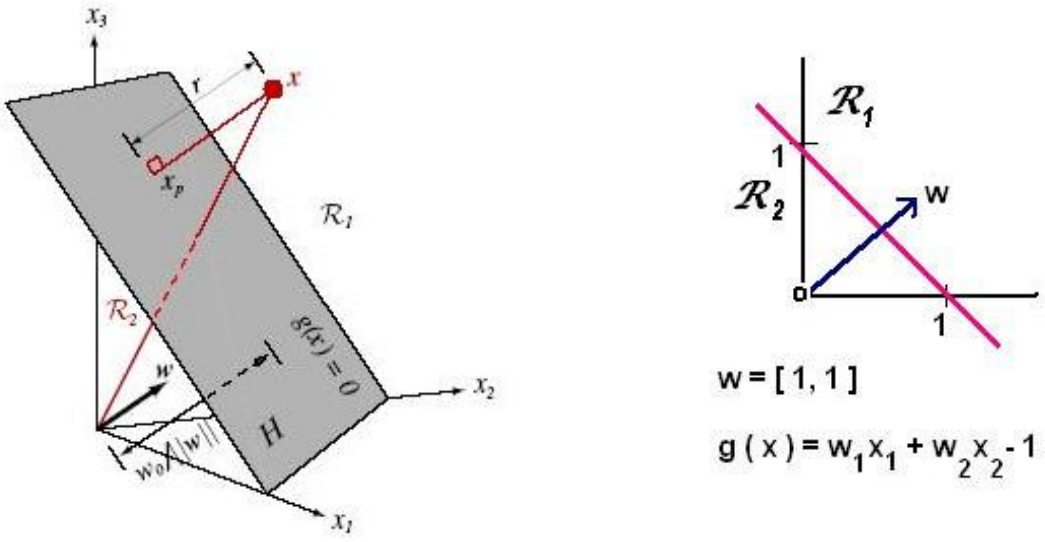


Figure 5.1: The linear decision boundary $\mathcal{H}$, designated by the linear discriminant function $g(\mathbf{x}) = \mathbf{w}^T + w_0$, separates the feature space into two half-spaces R_1 (where g($\mathbf{x}$) is positive) and R_2 (where g($\mathbf{x}$) is negative). Files duda5_2.jpg and linBound.jpg

The equation $g(\mathbf{x}) = 0$ defines the decision surface that separates points assigned to ω_1 from points assigned to ω_2. When $g(\mathbf{x})$ is linear, the decision surface is a *hyperplane*.

For any $\mathbf{x}_1, \mathbf{x}_2$ on the hyperplane we have:

$\mathbf{w}^T\mathbf{x}_1 + w_0 = \mathbf{w}^T\mathbf{x}_2 + w_0$, which implies $\mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2) = 0$.

However $(\mathbf{x}_1 - \mathbf{x}_2)$ is arbitrary. Thus $\mathbf{w}$ is normal to any segment lying in the hyperplane ($\mathcal{H}$) and the whole space is subdivided into R_1 ('positive side' of $\mathcal{H}$, with $g(\mathbf{x}) > 0$) and R_2 ('negative side' of $\mathcal{H}$, with $g(\mathbf{x}) < 0$). The vector $\mathbf{w}$ points to the 'positive side' of the hyperplane, i.e. to the direction R_2.

Signed Distance of any point $\mathbf{x}$ from the plane $\mathcal{H}$: $\delta = g(\mathbf{x})/\|\mathbf{w}\|$

Signed Distance of the origin to the hyperplane: $\delta_\mathbf{0} = w_0/\|\mathbf{w}\|$

If $w_0 > 0$, the origin is on the positive side of $\mathcal{H}$,
if $w_0 < 0$, the origin is on the negative side of $\mathcal{H}$,
if $w_0 = 0$, g($\mathbf{x}$) passes through the origin.

Summarizing, the value of the discriminant function g($\mathbf{x}$) is proportional to signed distance from $\mathbf{x}$ to the hyperplane.

5.2 Generalized linear discriminant functions

We may add to the ordinary equation of the decision boundary $g(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i$ some additional 2nd degree terms and write

$$g(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i + \underbrace{\sum_{i=1}^{d}\sum_{j=i}^{d} w_{ij} x_i x_j}_{\text{additional terms}},$$

which may be viewed as a function of $p(p+1)/2$ additional variables. In such a way the discriminant function designates a 2nd degree (hyperquadric) surface. Thus $g(\mathbf{x})$ is now a nonlinear function of the observed vector $\mathbf{x}$, however the nonlinearity is introduced in such a way that it allows to use the *linear methodology*.

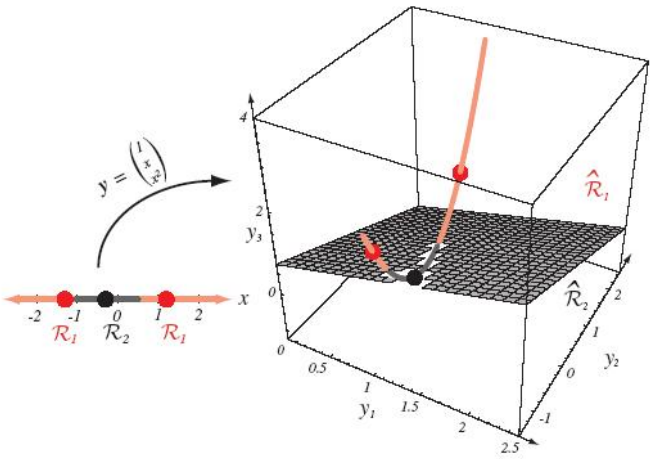


FIGURE 5.5. The mapping $\mathbf{y} = (1, x, x^2)^t$ takes a line and transforms it to a parabola in three dimensions. A plane splits the resulting y-space into regions corresponding to two categories, and this in turn gives a nonsimply connected decision region in the one-dimensional x-space. From: Richard O. Duda, Peter E. Hart, and David G. Stork,

Figure 5.2: The mapping $\mathbf{y} = (1,\ x,\ x^2)$ transforms a line into a parabola in 3 dimensions. File roadFigs/duda5_5.jpg

We may go further, and define *Polynomial* discriminant functions (generally *Phi* functions) by using arbitrary polynomial or other functions of $\mathbf{x}$:

$$g(\mathbf{x}) = \sum_{i=1}^{\ddot{d}} a_i y_i(\mathbf{x}),$$

where $y_i(\mathbf{x})$ – arbitrary function from $\mathbf{x}$, and $\ddot{d}$ denotes the number of the used *Phi* functions. In particular, the first term might be constant: $y_1(\mathbf{x}) \equiv 1$.

The linear discriminant function (decision boundary) can be now written as:

$$g(\mathbf{x}) = \mathbf{a}^T\mathbf{y}, \text{ where } \mathbf{a} = (a_1, \ldots, a_{\ddot{d}})^T,\ \mathbf{y} = (y_1, \ldots, y_{\ddot{d}})^T.$$

Practically we are in outer space of dimension $\ddot{d}$, however we use the methodology of ordinary linear discriminant functions.

The approach has its advantages and disadvantages.

Augmented feature and weight vectors

Define

$$\mathbf{y} = [1, x_1, \cdots, x_d]^T = [1; \mathbf{x}^T],$$

$$\mathbf{a} = [1, a_1, \cdots, a_d]^T = [1; \mathbf{a}^T]$$

and consider the discriminant function $g(\mathbf{y}) = \mathbf{a}^T\mathbf{y}$.

The augmented vectors are called also extended vectors:
Extended weight vector: $\vec{\mathbf{w}} = (w_1, \ldots, w_d, w_0)$
Extended input vector: $\vec{\mathbf{x}} = (x_1, \ldots, x_d, 1)$.
Their scalar product is denoted as $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ and called **dot product**.
The output function $g(\mathbf{x}) : \ R^d \longrightarrow \{-1, +1\}$ of the perceptron is $g(\mathbf{x}) = \text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$. Mathematically the frame above represents a trivial mapping from $\mathbf{x}$ to $\mathbf{y}$. However the mapping makes the computational tasks much easier.

Facts:

1. The addition of a constant component to $\mathbf{x}$ preserves all distance relationships among data vectors; the resulting $\mathbf{y}$ vectors lie all in the same $\mathbf{x}$-space.
2. The decision surface $\dot{\mathcal{H}}$ defined by $\mathbf{a}^T\mathbf{y} = 0$ passes through the origin in $\mathbf{y}$–space, even though the corresponding $\mathcal{H}$ can be in any position in $\mathbf{x}$–space.
3. The distance $\dot{\delta}$ from $\mathbf{y}$ to $\dot{\mathcal{H}}$ is given by $|\mathbf{a}^T\mathbf{y}|/\|\mathbf{a}\| = |w_0 + \mathbf{w}^T\mathbf{x}|/\|w_0 + w_1 + \cdots + w_g\|$, or $g(\mathbf{x})/\|\mathbf{a}\|$. Because $\|\mathbf{a}\| \geq \|\mathbf{w}\|$, $\dot{\delta} \leq \delta$.
4. Reduced task of finding the weight vector $\mathbf{w}$, without looking additionally for w_0.

5.3 Linearly separable case

Geometry and terminology

Suppose, we have the sample $\mathbf{y}_1, \ldots, \mathbf{y}_n$, $\mathbf{y}_i \in R^d$, $(i = 1, \ldots, n)$, containing augmented patterns belonging to class ω_1 or ω_2.

The sample is said to be **linearly separable**, if there exist such a weight vector $\mathbf{a} \in R^d$ that classifies the sample correctly.

A pattern y_i is classified correctly, if:

$$\begin{aligned} \mathbf{a}^T\mathbf{y_i} &> 0, \quad \text{for } y_i \in \omega_1 \\ \mathbf{a}^T\mathbf{y_i} &< 0, \quad \text{for } y_i \in \omega_2 \end{aligned}$$

In the following we simplify this condition by introducing specifically '**normalized data**': we replace all data samples labelled ω_2 by their negatives.

With this normalization, we may say that a sample of patterns is separable, when

$$\exists_{\mathbf{a}} \text{ such that } \mathbf{a}^T\mathbf{y_i} > 0$$

for all patterns belonging to the sample.

The weight vector $\mathbf{a}$ may not be unique. Generally, it belongs to a solution space. This is illustrated in Figure 5.3.

In Figure 5.3 we show four data vectors, two of them belonging to class ω_1 and the other two to class ω_2. Because we use augmented vectors notation, the separating plane

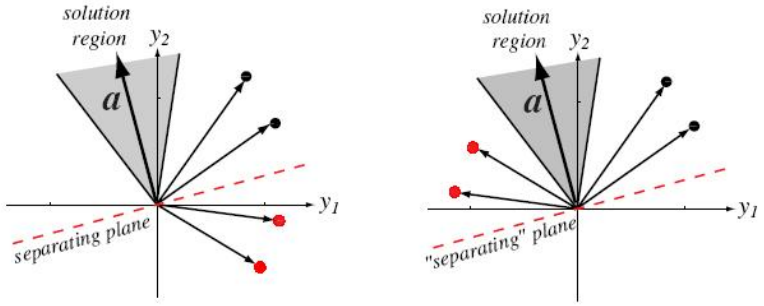


FIGURE 5.8. Four training samples (black for ω_1, red for ω_2) and the solution region in feature space. The figure on the left shows the raw data; the solution vectors leads to a plane that separates the patterns from the two categories. In the figure on the right, the red points have been "normalized"—that is, changed in sign. Now the solution vector leads to a plane that places all "normalized" points on the same side. From: Duda, . . .

Figure 5.3: The solution space for original data vectors (left) and specifically normalized – negated (right). Files duda5_8.jpg

should pass through the origin, and the 2 groups of vectors should be on different 'sides' of that plane. This is shown in the left plot of Figure 5.3. However, using specifically normalized (negated when in ω_2) data, all data vectors correctly classified appear on the same (positive) side of the separating plane – and it is easy to see wrongly classified data vectors. This is illustrated in Figure 5.3, right plot. We may see also, that the separating plane – marked by the stripped line – is not unique.

Each data vector $\mathbf{y}_i$ places a constraint on the possible location of a solution vector $\mathbf{a}$. The equation $\mathbf{a}^T\mathbf{y}_i = 0$ defines a hyperplane through the origin of weight space having $\mathbf{y}_i$ as a normal vector. The solution vector – if it exists – must be on the positive side of every hyperplane. Thus, a solution vector must lie in the intersection of n half-spaces.

It is obvious, that the solution vector is not unique. To make it unique, some additional constraints should be added. One possibility is to seek for the *minimum-length* vector satisfying

$$\mathbf{a}^T\mathbf{y}_i > b,$$

for all i, where b is a positive constant and is called **the margin**. This means, that the solution region is shifted by the distance $b/\|\mathbf{y}_i\|$.

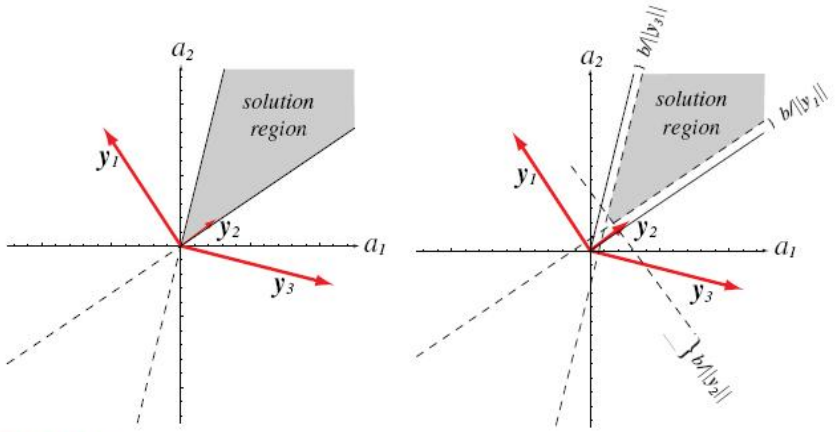


FIGURE 5.9. The effect of the margin on the solution region. At the left is the case of no margin ($b = 0$) equivalent to a case such as shown at the left in Fig. 5.8. At the right is the case $b > 0$, shrinking the solution region by margins $b/\|\mathbf{y}_i\|$. From: Duda, . . .

5.4 Seeking for the solution: Basic Gradient Descent methods

For a given sample od n data vectors we seek for a solution $\mathbf{a}$ satisfying the set of linear inequalities $\mathbf{a}^T\mathbf{y}_i > 0$,. This leads to defining a criterion function $J(\mathbf{a})$ that is minimized if $\mathbf{a}$ is a solution vector. This reduces our problem to one of minimizing a scalar function – and the problem can often be solved by a gradient descent procedures.

• The **basic gradient descent procedure** `'BasicGradientDescent'` is very simple. We start from an arbitrary start value $\mathbf{a}(1)$ and move in the direction of steepest descent, i.e. along the negative of the gradient – this yields us the next approximation $\mathbf{a}(2)$. Generally, $\mathbf{a}(k+1)$ is obtained from $\mathbf{a}(k)$ by the equation:

$$\mathbf{a}(k+1) = \mathbf{a}(k) \; - \; \eta(k)\boldsymbol{\nabla} J(\mathbf{a})|_{\mathbf{a}=\mathbf{a}(k)}, \tag{5.1}$$

where $\eta(k)$ is a positive scale factor or **learning rate**, and the gradient $\boldsymbol{\nabla} J(\mathbf{a})$ is evaluated at $\mathbf{a}$=$\mathbf{a}$(k).

The steps $k, k+1, \ldots$ are iterated until a stop criterion is met. This could be

$$|\eta(k)\boldsymbol{\nabla} J(\mathbf{a})| < \theta,$$

where θ is a small constant.

There are many problems associated with gradient descent procedures. One is of finding an appropriate learning rate $\eta(k)$. If $\eta(k)$ is too small, convergence is needlessly slow, whereas if $\eta(k)$ is too large, the correction process will overshoot and can even diverge.

One way to find a proper $\eta(k)$ is to use the formula

$$\eta(k) = \frac{\|\boldsymbol{\nabla} J\|^2}{\boldsymbol{\nabla} J^T \mathbf{H} \boldsymbol{\nabla} J},$$

where $\mathbf{H}$ depends on $\mathbf{a}$, and thus indirectly on k. Both $\mathbf{H}$ and $\boldsymbol{\nabla} J$ are evaluated at $\mathbf{a}(k)$.

Note that if the criterion function $J(\mathbf{a})$ is quadratic throughout the region of interest, then $\mathbf{H}$ is constant and η is a constant independent of k.

• An alternative is to use **Newton's algorithm** `'Newton_descent'`. The algorithm is very similar to that formulated above as basic gradient descent, with the difference, that eq. (5.1) is now replaced by the following one

$$\mathbf{a}(k+1) = \mathbf{a}(k) \; - \; \mathbf{H}^{-1}\boldsymbol{\nabla} J. \tag{5.2}$$

Generally speaking, Newton's algorithm gives usually a greater improvement *per step*, however is not applicable, if the Hessian matrix is singular. Moreover, it needs matrix inversion at each step, which is computationally expensive (inversion $\mathbf{H}^{-1}$ is $O(d^3)$). Sometimes it is preferable to make more simple basic descent steps then invert the Hessian at each step.

5.5 The perceptron criterion function, Error correcting procedures

In this subsection we consider so called **error correcting rules**. We correct (update) the weights using only misclassified patterns.

Throughout this subsection – unless it is stated otherwise – we consider **separable data**. Also, the patterns belonging to the training sample are specially normalized (i.e. patterns from class ω_2 are negated). This should have the effect that for a 'good' separating hyperplane we should obtain for all training patterns $\mathbf{a}^T\mathbf{y_i} > 0 \ \forall_i$; only misclassified vectors will yield negative values of the classifier $\mathbf{a}^T\mathbf{y_i} < 0$.

Let $\mathcal{Y}$ denote the set of wrongly classified patterns. Our updating rule for the actual weight vector $\mathbf{a}$ will be based just taking into account only the patterns contained in $\mathcal{Y}$.

Say, we have a decision surface (boundary) $\mathcal{H}$ designated by the actual weight vector $\mathbf{a}$. Does it separate the data belonging to the classes ω_1 and ω_1? We should define a criterion of fitness of the decision boundary.

The most obvious choice of the criterion function J evaluating the quality of the classifier is to let $J(\mathbf{a}; \mathbf{y}_1, \ldots, \mathbf{y}_n)$ be the number of sample patterns – misclassified by $\mathbf{a}$. However, this function is piecewise constant, and therefore is a poor candidate for a gradient search. Better criterion is the **Perceptron criterion function**

$$J_p(\mathbf{a}) = \sum_{\mathbf{y}\in\mathcal{Y}} (-\mathbf{a}^T\mathbf{y}), \tag{5.3}$$

where $\mathcal{Y}$ is the set of samples misclassified by $\mathbf{a}$. Remind, the samples are misclassified by the vector $\mathbf{a}$ if $\mathbf{a}^T\mathbf{y} \leq 0$. If no samples are misclassified, $\mathcal{Y}$ is empty and we define J_p to be zero.

The criterion $J_p(\mathbf{a})$ defined by eq. 5.3 is never negative, being zero only if $\mathbf{a}$ is a solution vector for the separable sample. Geometrically, $J_p(\mathbf{a})$ is *proportional* to the sum of the distances from the misclassified samples to the decision boundary.

The gradient of J_p is

$$\boldsymbol{\nabla} J_p = \sum_{\mathbf{y}\in\mathcal{Y}} (-\mathbf{y}).$$

We may update the weight vector $\mathbf{a}$ *in batch* or by *s*ingle sample updating.

• The **gradient batch perceptron update rule** 'perceptron_batch' is

$$\mathbf{a}(k+1) = \mathbf{a}(k) + \eta(k) \sum_{\mathbf{y}\in\mathcal{Y}} (-\mathbf{y}), \tag{5.4}$$

where $\mathcal{Y}$ is the set of patterns misclassified by $\mathbf{a}$.

This rule can be described very simply: The next weight vector is obtained by adding some multiple of the sum of the misclassified samples to the present weight vector.

We call the update rule above 'the batch update rule', because it updates the weight vector once with a large group of samples.

• The **Fixed-Increment Single-Sample Perceptron** `'Perceptron_FIS'` makes correction of $\mathbf{a}$ based upon a single misclassified pattern. We present to the network the training samples in a randomized order. Obtaining a misclassified sample we make a correction of the actual value of the weight vector $\mathbf{a}$. The update rule is:

$$\begin{array}{ll} \mathbf{a}(1), & \text{arbitrary} \\ \mathbf{a}(k+1) = \mathbf{a}(k) + \mathbf{y}^k, & k \geq 1 \end{array}$$

The updating is performed only if $\mathbf{y}^k$, the pattern evaluated in the kth iteration, is misclassified (i.e. it yielded $\mathbf{a}(k)^T\mathbf{y}^k \leq 0$).

It can be proved, that for separable samples the batch perceptron rules and the Fixed-Increment Single-Sample Perceptron rules will terminate at a solution vector.

Some direct generalizations are:

• **Variable-Increment Perceptron with margin b** `Perceptron_VIM`. The updating rule is

$$\begin{array}{ll} \mathbf{a}(1), & \text{arbitrary} \\ \mathbf{a}(k+1) = \mathbf{a}(k) + \eta(k)\mathbf{y}^k. & k \geq 1 \end{array}$$

The updating is now performed immediately after finding misclassified $\mathbf{y}^k$, i.e. for those samples which do not satisfy $\mathbf{a}(k)^T\mathbf{y}^k > b$ (i.e. which yield $\mathbf{a}^T\mathbf{y}^k \leq b$).

It can be shown that if the samples are linearly separable and if

$$\lim_{m\to\infty} \sum_{k=1}^{m} \eta(k) = \infty \quad (\eta(k) \geq 0)$$

and

$$\lim_{m\to\infty} \frac{\sum_{k=1}^{m} \eta(k)^2}{(\sum_{k=1}^{m} \eta(k))^2} \; = \; 0,$$

then $\mathbf{a}$(k) converges to a solution vector $\mathbf{a}$ satisfying $\mathbf{a}^T\mathbf{y}_i > b$ for all i.

In particular, these conditions on $\eta(k)$ are satisfied if $\eta(k)$ is a positive constant or if it decreases at $1/k$.

• **Batch Variable-Increment Perceptron with Margin** `'Perceptron_BVI'`. The updating rule is

$$\begin{array}{ll} \mathbf{a}(1), & \text{arbitrary} \\ \mathbf{a}(k+1) = \mathbf{a}(k) + \eta(k) \sum \mathbf{y}^k \in \mathcal{Y}, & k \geq 1 \end{array}$$

where $\mathcal{Y}$ is the set of training patterns misclassified by $\mathbf{a}$(k), i.e. not satisfied the desired inequality $\mathbf{a}^T\mathbf{y}_i > b$, with $b > 0$ denoting the desired margin.

The benefit of batch gradients descent is that trajectory of weight vector is smoothed, compared to single-vector algorithms.

• **Winnow's algorithm** `'Balanced_Winnow'` applicable to separable training data. In the version 'Balanced_Winnow', the components of the returned weight vector are divided into 'positive' and 'negative' weight vectors, $\mathbf{a}^+$ and $\mathbf{a}^-$, each associated with one of the two categories to be learned. Corrections are made only for the vectors corresponding to the misclassified counterparts:
if $\mathcal{Y}$ contains misclassified patterns from ω_1, then adjust $\mathbf{a}^+$,
if $\mathcal{Y}$ contains misclassified patterns from ω_2, then adjust $\mathbf{a}^-$.
In one iteration both $\mathbf{a}^+$ and $\mathbf{a}^-$ may be adjusted.

The convergence is usually faster than when using the perceptron rule, especially, when a larger number of irrelevant redundant features are present.

• **Relaxation Procedures**
Use differently formulated criteria. One such criterion is

$$J_q(\mathbf{a}) = \sum_{\mathbf{y}\in\mathcal{Y}} (\mathbf{a}^T\mathbf{y})^2, \tag{5.5}$$

where $\mathcal{Y}(\mathbf{a})$ again denotes the set of training samples misclassified by $\mathbf{a}$. The gradient of J_q is continuous, whereas the gradient of J_p is not. Thus J_q presents a smoother surface to search.

A better criterion (not so much influenced by the length of the training vectors) is **Batch relaxation with Margin** `'Relaxation_BM'`

$$J_q(\mathbf{a}) = \frac{1}{2}\sum_{\mathbf{y}\in\mathcal{Y}} \frac{(\mathbf{a}^T\mathbf{y} - b)^2}{\|\mathbf{y}\|^2}, \tag{5.6}$$

where now $\mathcal{Y} = \mathcal{Y}(\mathbf{a})$ is the the set of training samples for which $\mathbf{a}^T\mathbf{y} \leq b$. If $\mathcal{Y}(\mathbf{a})$ is empty, we define J_r to be zero. Thus, $J_r(\mathbf{a})$ is never negative, and is zero iff $\mathbf{a}^T\mathbf{y} \geq b$ for all of the training samples. The gradient of J_r is

$$\boldsymbol{\nabla} J_r = \sum_{\mathbf{y}\in\mathcal{Y}} \frac{\mathbf{a}^T\mathbf{y} - b}{\|\mathbf{y}\|^2}$$

The update rule: Initialize weights $\mathbf{a}$, margin b. Then iterate

$$\begin{array}{ll} \mathbf{a}(1) & \text{arbitrary} \\ \mathbf{a}(k+1) = \mathbf{a}(k) + \eta(k)\sum_{\mathbf{y}\in\mathcal{Y}} \frac{\mathbf{a}^T\mathbf{y}-b}{\|\mathbf{y}\|^2}\mathbf{y} & k \geq 1 \end{array}$$

until Y={ }, i.e. the set of misclassified patterns is empty.

There is also a rule for **Single-Sample Relaxation with Margin** `Relaxation_SSM`.

The procedures considered up to now were *error correcting procedures*. The call for a modification of the weight vector was executed when and only when an error in classification (i.e. misclassification) was encountered. Thus it was a relentless search for error–free solution.

When the relaxation rule is applied to a set of linearly separable samples, the number of corrections may or may not be finite. If it is finite, then of course we have obtained a solution vector. If it is not finite, it can be shown that $\mathbf{a}$(k) converges to a limit vector on the boundary of the solution region.

Nonseparable behavior

In principle the algorithms shown in this subsection work good for linearly separable samples. In practice one could expect that in the non-separable case the methods will work good if the error for the optimal linear discriminant function is low.

Obviously, a good performance in the training sample does not mean a good performance in the test sample.

Because no weight vector can correctly classify every sample in a nonseparable set (by definition), the corrections in an error-correction procedure can never cease. Each algorithm produces an infinite sequence of weight vectors, any member of which may or may not yield a 'solution'.

A number of similar heuristic modifications to the error-correction rules have been suggested and studied empirically. The goal of these modifications is to obtain acceptable performance on nonseparable problems while preserving the ability to find a separating vector on separable problems. A common suggestion is the use of a variable increment $\eta(k)$.

5.6 Minimum squared-error procedures

This is an approach that sacrifices the ability to obtain a separating vector for good compromise performance on both separable and nonseparable problems. A criterion function that involves *all* the samples will be considered. We shall try to make $\mathbf{a}$ satisfy the equations $\mathbf{a}^T\mathbf{y}_i = b_i$, with b_i denoting some arbitrarily specified positive constants.

In the following we consider 2 algorithms: 'LS', the classical least-square algorithm, and 'LMS', the iterative least-mean-square (Widrow-Hoff) algorithm.

• **The classical least-square algorithm** `'LS'`. The problem of solving a set of linear inequalities is replaced by the problem of solving a set of linear equalities

$$\mathbf{Ya} = \mathbf{b}. \tag{5.7}$$

Define the error vector $\mathbf{e} = \mathbf{Ya} - \mathbf{b}$. One approach to find $\mathbf{a}$ minimizing the squared length of the *sum-of-squared-error* vector or the sum-of-squared-error criterion function

$$J_s(\mathbf{a}) = \|\mathbf{Ya} - \mathbf{b}\|^2 = \sum_{i=1}^{n}(\mathbf{a}^T\mathbf{y}_i - b_i)^2. \tag{5.8}$$

The gradient of J_s equalled to zero yields the set of equations

$$\boldsymbol{\nabla} J_s = \sum_{i=1}^{n} 2(\mathbf{a}^T\mathbf{y}_i - b_i)\mathbf{y}_i = 2\mathbf{Y}^T(\mathbf{Ya} - \mathbf{b}) = \mathbf{0}.$$

Hence the normal equations

$$\mathbf{Y}^T\mathbf{Ya} = \mathbf{Y}^T\mathbf{b}. \tag{5.9}$$

Getting the solution. The matrix $\mathbf{Y}^T\mathbf{Y}$ is square and often nonsingular. If it is nonsingular, we can solve eq. (5.9) obtaining a unique solution

$$\mathbf{a} = (\mathbf{Y}^T\mathbf{Y})^{-1}\mathbf{Y}^T\mathbf{b} = \mathbf{Y}^\dagger\mathbf{b},$$

where $\mathbf{Y}^\dagger = (\mathbf{Y}^T\mathbf{Y})^{-1}\mathbf{Y}^T$ is called pseudoinverse.

Note that $\mathbf{Y}^\dagger\mathbf{Y}) = \mathbf{I}$, but generally $\mathbf{YY}^\dagger \neq \mathbf{I}$.

Pseudoinverse can be defined more generally as

$$\mathbf{Y}^\dagger \equiv \lim_{\epsilon\to 0}(\mathbf{Y}^T\mathbf{Y} + \epsilon\mathbf{I})^{-1}\mathbf{Y}^T.$$

It can be shown that this limit always exists and that $\mathbf{a} = \mathbf{Y}^\dagger\mathbf{b}$ is a *minimum-square-error* (MSE) solution to the equation set $\mathbf{Ya} = \mathbf{b}$.

If $\mathbf{b}$ is fixed arbitrarily, there is no reason to believe that the MSE solution yields a separating vector in the linearly separable case. However, it is reasonable to hope that

by minimizing the squared-error criterion function we might obtain a useful discriminant function in both the separable and the nonseparable case.

In Duda [1] pp. 241–243 there are two examples, which sustain this hope. In particular, to get Fisher's discriminant, we put

$$\mathbf{b} = \begin{bmatrix} \frac{n}{n_1}\mathbf{I}_1 \\ \frac{n}{n_2}\mathbf{I}_2 \end{bmatrix}$$

Asymptotic Approximation to an Optimal Discriminant

If $\mathbf{b} = \mathbf{1}_n$, the MSE solution approaches a minimum mean-square-error approximation to the Bayes discriminant function

$$g_0(\mathbf{x}) = P(\omega_1|\mathbf{x}) - P(\omega_2|\mathbf{x}).$$

Proof in Duda [1], pp. 244.

• **The iterative least-mean-square (Widrow-Hoff) algorithm** `'LMS'`

The method minimizes the criterion $J_s(\mathbf{a}) = \|\mathbf{Ya} - \mathbf{b}\|^2$ using the gradient descent algorithm. Advantages:

i) avoids the problem, when $\mathbf{Y}^T\mathbf{Y}$ is singular,

ii) permits to work sequentially with subsequent data vectors, does not needs the whole matrix $\mathbf{Y}$ to be kept in the memory.

The gradient of $J_s(\mathbf{a})$ equals $\boldsymbol{\nabla} J_s = 2\mathbf{Y}^T(\mathbf{Ya} - \mathbf{b})$.
The obvious update rule is:

$$\begin{aligned} &\mathbf{a}(1), \qquad \text{arbitrary} \\ &\mathbf{a}(k+1) = \mathbf{a}(k) + \eta(k)\mathbf{Y}^T(\mathbf{Ya} - \mathbf{b}). \end{aligned} \tag{5.10}$$

Theorem. (problem 5.26 in Duda):
If $\eta(k) = \eta(1)/k$, where $\eta(1)$ is any positive constant, then the rule (5.10) generates a sequence of weight vectors that converges to a limiting vector $\mathbf{a}$ satisfying

$$\mathbf{Y}^T(\mathbf{Ya} - \mathbf{b}) = 0.$$

Thus, the descent algorithm yields a solution regardless of whether or not $\mathbf{Y}^T\mathbf{Y}$ is singular.

The rule (5.10) can be further reduced by considering the samples sequentially and using the Widrow–Hoff or LMS (least-mean-squared) rule

$$\begin{aligned} &\mathbf{a}(1), \qquad \text{arbitrary}, \\ &\mathbf{a}(k+1) = \mathbf{a}(k) + \eta(k)(\mathbf{b}(k) - \mathbf{a}^T)\mathbf{y}^k), \end{aligned} \tag{5.11}$$

where $\mathbf{y}^k$ denotes subsequent sample vector. Stop criterion: $|\eta(k)(\mathbf{b}(k) - \mathbf{a}^T)\mathbf{y}^k)| < \theta$, with θ a presumed small constant.

Comparison with the relaxation rule

1. LMS is not an error-correction rule, thus the corrections never cease.
2. To obtain convergence, the learning rate must decrease with k; the choice $eta(k) = eta(1)/k$ being common.
3. Exact analysis in the deterministic case is rather complicated, and merely indicates that the sequence of weight vectors tends to converge to the desired solution.

The Ho–Kashyap procedures

The procedures train a linear classifier using the Ho-Kashyap algorithm. Type of training may be simple or modified. Additional output: The weights for the linear classifier and the final computed margin.

Voted Perceptron Classifier `Perceptron_Voted`

The voted perceptron is a simple modification over the Perceptron algorithm. The data may be transformed using a kernel function so as to increase the separation between classes. Coded by Igor Makienko and Victor Yosef. Belongs to the non-parametric group of procedures.

Pocket Algorithm `Pocket`

Performs training of the perceptron. Updates are retained only if they perform better on a random sample of the data. The perceptron is trained for 10 iterations. Then the updated weights $\mathbf{a}$ and the old weights $\mathbf{a}_{\_}$ are tested on a training sample. The new vector $\mathbf{a}$ replaces the old one ($\mathbf{a}_{\_}$), if it provides a smaller number of misclassified patterns.

Farthest-Margin perceptron `Perceptron_FM`

The wrongly classified sample *farthest* from the current decision boundary is used to adjust the weight of the classifier.

Additional input: The maximum number of iterations and the slack for incorrectly classified samples.

5.7 Summary of the hereto presented algorithms

- The MSE procedures yield a weight vector whether the samples are linearly separable or not, but there is no guarantee that this is a separating vector in the separable case, see Figure 5.4 (Fig. 5.17 from Duda). If the margin vector $\mathbf{b}$ is chosen arbitrarily, all we can say is that the MSE procedures minimize $\|\mathbf{Ya} - \mathbf{b}\|^2$.

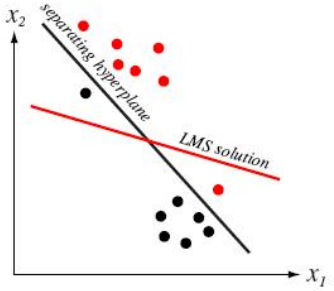


FIGURE 5.17. The LMS algorithm need not converge to a separating hyperplane, even if one exists.

Figure 5.4: The LMS algorithm may yield a different solution as the separating algorithm (points left of the separating plane are black, to the right are red). File duda5_2.jpg

- The Perceptron and Relaxation find separating vectors if sample is separable, but do not converge on non–separable problems.

Considered algorithms:

`BasicGradientDescent`, not in Patt box
`Newton_descent`, not in Patt box

`Perceptron_batch`, gradient batch perceptron
`Perceptron_FIS`, Fixed-Increment Single-Sample perceptron
`Perceptron_VIM`, Variable-Increment Perceptron with Margin
`Perceptron_BVI`, Batch Variable-Increment Perceptron with Margin
`Balanced_Winnow`
`Relaxation_BM`, Batch Relaxation with Margin
`Relaxation_SSM`, Single-Sample Relaxation with Margin

`LS`, direct Least-squares classifier
`LMS`, iterative Least-mean-square, Widrow–Hoff classifier
`Perceptron_Voted` , Voted Perceptron classifier
`Pocket`, Pocket Algorithm
`Perceptron_FM`, Farthest-margin perceptron

Next subsection:
`SVM`, Support Vector Machine

5.8 Linear programming methods

The Linear Programming (LP) method finds the solution to the problem

$$\textit{Minimize objective function} \quad z = \boldsymbol{\alpha}^T \mathbf{u}, \quad \textit{subject to} \quad \mathbf{A}\mathbf{u} \geq \boldsymbol{\beta}. \tag{5.12}$$

The problem may be solved[3] by the **simplex algorithm**, which needs additional constraint:

$$\mathbf{u} \geq \mathbf{0}. \tag{5.13}$$

When looking for a good classifier, we seek a weight vector $\mathbf{a}$ which clearly my have both positive and negative components, thus which does not satisfies eq. (5.13). However, we may redefine the vector $\mathbf{a}$ as

$$\mathbf{a} = \mathbf{a}^+ - \mathbf{a}^-, \quad \text{with} \quad \mathbf{a}^+ \geq \mathbf{0} \text{ and } \mathbf{a}^- \geq \mathbf{0}, \tag{5.14}$$

with $\mathbf{a}^+$ and $\mathbf{a}^-$ having only non-negative components. This may be done by defining $\mathbf{a}^+$ and $\mathbf{a}^-$ the following way

$$\mathbf{a}^+ \equiv \frac{1}{2}(|\mathbf{a}| + \mathbf{a}), \quad \mathbf{a}^- \equiv \frac{1}{2}(|\mathbf{a}| - \mathbf{a}).$$

Having done this, the simplex algorithm can be applied in the classification problem for search of the weight vector $\mathbf{a}$ – as shown below.

5.8.1 Formulation of the classification algorithm as a LP task, in the linearly separable case

We have training samples $\mathbf{y}_1, \ldots, \mathbf{y}_n$. The patterns are classified into ω_1 or ω_2 and their group membership is known. We seek for a weight vector $\mathbf{a}$ that satisfies

$$\mathbf{a}^T \mathbf{y}_i \geq b_i > 0, \quad i = 1, \ldots, n,$$

with given margin $\mathbf{b} = (b_1, \ldots, b_n)^T$. The margin can be the same for all i, i.e. $\mathbf{b} = \underbrace{(1, \ldots, 1)}_{n \text{ times}}^T$. How to formulate the problem as a LP problem?
We introduce an artificial variable $\tau \geq 0$ satisfying

$$\mathbf{a}^T \mathbf{y}_i + \tau \geq b_i. \tag{5.15}$$

There should be one τ for all i.
One solution is: $\mathbf{a}$=$\mathbf{0}$ and $\tau = \max_i b_i$.
However our problem is that we want minimize τ over all values of $(\tau, \mathbf{a}$ that satisfy the constraints $\mathbf{a}^T \mathbf{y}_i \geq b_i$ and $\tau \geq 0$.

By some algorithm (e.g. by the algorithm outlined below) we will obtain some τ. Then our reasoning will be the following:
If the answer is zero, the samples are linearly separable – we have a solution.
If the answer is positive, there is no separating vector, but we have proof that the samples are nonseparable.

[3] In linear programming a *feasible solution* is any solution satisfying the constraints.
A feasible solution, for which the number of nonzero variables does not exceed the number of constraints (not counting the simplex requirement for non-negative variables) is called *a basic feasible solution*. Possession of such a solution simplifies the application of the simplex algorithm.

Formally our problem in the LP framework may be formulated as follows: Find $\mathbf{u}$ that minimizes $z = \boldsymbol{\alpha}^T\mathbf{u}$ subject to $\mathbf{Au} \geq \boldsymbol{\beta},\ \mathbf{u} \geq \mathbf{0}$, with $\mathbf{A}, \mathbf{u}, \boldsymbol{\alpha}, \boldsymbol{\beta}$ defined as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{y}_1^T & -\mathbf{y}_1^T & 1 \\ \mathbf{y}_2^T & -\mathbf{y}_2^T & 1 \\ \vdots & \vdots & \\ \mathbf{y}_n^T & -\mathbf{y}_n^T & 1 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \mathbf{a}^+ \\ \mathbf{a}^- \\ \tau \end{bmatrix}, \quad \boldsymbol{\alpha} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ 1 \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

We have here n constraints $\mathbf{Au} \geq \boldsymbol{\beta}$, plus the simplex constraints $\mathbf{u} \geq \mathbf{0}$.
The simplex algorithm yields minimum of the objective function $z = \boldsymbol{\alpha}^T\mathbf{u}$ $(= \tau)$ and the vector $\hat{\mathbf{u}}$ yielding that value. Then we put $\mathbf{a} = \mathbf{a}^+ - \mathbf{a}^-$.

5.8.2 Minimizing the Perceptron–Criterion function with margin *b*

The problem of minimizing the Perceptron–criterion function can be posed as a problem in Linear Programming – the minimization of this criterion function yields a separating vector in the separable case and a reasonable solution in the nonseparable case.

$$J_p(\mathbf{a}) = \sum_{\mathbf{y}\in\mathcal{Y}} (-\mathbf{a}^T\mathbf{y}),$$

where $\mathcal{Y}(\mathbf{a})$ is the set of training samples misclassified by $\mathbf{a}$.

To avoid the useless solution $\mathbf{a}$=$\mathbf{0}$, we introduce a positive margin $\mathbf{b}$ and write

$$J'_p(\mathbf{a}) = \sum_{\mathbf{y}\in\mathcal{Y}'} (b_i - \mathbf{a}^T\mathbf{y}_i),$$

where $\mathbf{y}_i \in \mathcal{Y}'$, if $\mathbf{a}^T\mathbf{y}_i \leq b_i$.

J'_p is a piece-wise linear function of $\mathbf{a}$, and linear programming techniques are not immediately applicable. However, by introducing n artificial variables and their constraints, we can construct an equivalent linear objective function.

Task: Find $\mathbf{a}$ and $\boldsymbol{\tau}$ that minimize the linear function

$$z = \sum_{i=1}^{n} \tau_i \quad \text{subject to} \quad \tau_k \geq 0, \quad \text{and } \tau_k \geq b_i - \mathbf{a}^T\mathbf{y}_i. \tag{5.16}$$

We may write it as the following problem:

$$\text{Minimize} \quad \boldsymbol{\alpha}^T\mathbf{u} \quad \text{s.t.} \quad \mathbf{Au} \geq \boldsymbol{\beta} \quad \text{and} \quad \mathbf{u} \geq \mathbf{0}, \tag{5.17}$$

where

$$\mathbf{A} = \begin{bmatrix} \mathbf{y}_1^T & -\mathbf{y}_1^T & 1 & 0 & \dots & 0 \\ \mathbf{y}_2^T & -\mathbf{y}_2^T & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \\ \mathbf{y}_n^T & -\mathbf{y}_n^T & 0 & 0 & \dots & 1 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \mathbf{a}^+ \\ \mathbf{a}^- \\ \boldsymbol{\tau} \end{bmatrix}, \quad \boldsymbol{\alpha} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The choice $\mathbf{a}$=$\mathbf{0}$ and $\tau_i = b_i$ provides a basic feasible solution to start the simplex algorithm, which provides an $\hat{\mathbf{a}}$ minimizing $J'_p(\mathbf{a})$ in a finite number of steps.

There are other possible formulations, the ones involving the so–called *dual problem* being of particular interest from computational standpoint.

The linear programming methods secure the advantage of guaranteed convergence on both separable and nonseparable problems.

5.9 Support Vector Machines I. Separable Samples

The Support Vector Machines were introduced by Vladimir Vapnik (see book by Vapnik[4], also: Cortes and Vapnik [9], Steve Gunn [5], Pontil and Verri [8]). The basic concept used in this domain is the *dot product*.

Let $\mathbf{a}, \mathbf{b} \in R^n$ be two real n dimensional vectors.
The **dot product** of $\mathbf{a}, \mathbf{b}$ is defined as

$$(\mathbf{a} \cdot \mathbf{b}) = \mathbf{a}^T\mathbf{b} = \sum_{i=1}^{n} a_i b_i.$$

Let $\mathbf{y}$ denote the analyzed data vector. This data vector might be obtained from the original observed data vector $\mathbf{x}$ by a transformation $\mathbf{y} = \varphi(\mathbf{x})$. Usually $\varphi(\,.\,)$ denotes a kernel method – see subsection 5.11 for a closer specification of the kernels. Generally, $\mathbf{y}$ has a larger dimension as $\mathbf{x}$; we say that $\mathbf{x}$ is the *observed* and $\mathbf{y}$ the *feature* vector.

Suppose, we have a training sample of feature vectors $\mathbf{y}_1, \ldots, \mathbf{y}_n$, with elements belonging to two classes: ω_1 and ω_2. We assume throughout this subsection that the sample is linearly separable, which means that there exists a separating hyperplane with the following property: all samples belonging to class ω_1 are located on the 'positive' side of the hyperplane, while samples belonging to class ω_2 are located on the 'negative' side of that hyperplane. The separation is sharp: no sample is located exactly *on* the hyperplane, see Fig. 5.5 for an illustration.

Denote by

$$\mathcal{H} : g(\mathbf{y}) = \mathbf{a}^T\mathbf{y} + b$$

the separating hyperplane, where ($\mathbf{a}$, b) are called weights and bias respectively.

From the assumption of the separability of the sample we have ($k = 1, \ldots, n$) {*sep* }:

$$\begin{cases} g(\mathbf{y}_k) > 0, & \text{for } \mathbf{y}_k \in \omega_1, \\ g(\mathbf{y}_k) < 0, & \text{for } \mathbf{y}_k \in \omega_2, \end{cases} \tag{5.18}$$

Let us define the variable z_k as:

$$z_k = +1, \quad \text{if } \mathbf{y}_k \in \omega_1, \quad \text{and} \quad z_k = -1, \quad \text{if } \mathbf{y}_k \in \omega_2.$$

Then, using the variables z_k, the inequalities (5.18) may be converted to the following ones {*sep1* }:

$$z_k g(\mathbf{y}_k) > 0, \quad k = 1, \ldots, n. \tag{5.19}$$

Because the inequality (5.19) is sharp, we can always find a value $m > 0$ such that for each k, the distance of $\mathbf{y}_k$ from the separating hyperplane $\mathcal{H}$ is not smaller then m {*margin*}:

$$\frac{z_k g(\mathbf{y}_k)}{\|\mathbf{a}\|} \geq m, \quad k = 1, \ldots, n. \tag{5.20}$$

The value m satisfying (5.20) is called **the margin**.

The points (data vectors) satisfying (5.20) with the equality sign, i.e. lying on the margin line, are called **support vectors**.

[4] V. Vapnik, The Nature of Statistical Learning. Springer-Verlag 1995

We could say also that support vectors are those data points that lie closest to data points of another class.

In Figure 5.5 we show two groups of data separated by a thick line (indicating the separating hyperplane $\mathcal{H}$). Parallel to the separating line two others (thin) lines are depicted; they constitute (delimit) the margin. For the data set exhibited in the figure, each margin is supported by one support point lying exactly in the margin line; generally, for other data sets, there could be more support points.

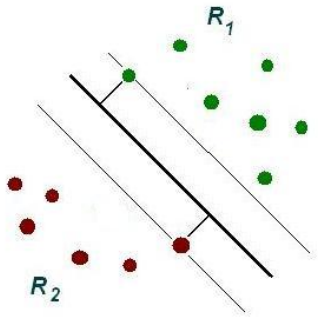


Figure 5.5: Support Vector Machine. The problem of finding the separating hyperplane with the largest margin. The data points belong to 2 classes and are separable. File sv1.jpg

Our goal now is to find the hyperplane (i.e. weight vector $\mathbf{a}$) that *maximizes* the margin m (it is believed that the larger the margin, the better generalization of the classifier).

From (5.20) we see that m depends from $\|\mathbf{a}\|$: taking smaller $\|\mathbf{a}\|$ makes the margin larger. Therefore we assume the additional constraint {*constr1*}:

$$m \cdot \|\mathbf{a}\| = 1. \tag{5.21}$$

Thus, minimizing $\|\mathbf{a}\|$ we maximize the margin m.

Substituting $m := 1/\|\mathbf{a}\|$, the inequalities (5.20) are converted to {*constr2*}:

$$z_k g(\mathbf{y}_k) \geq 1, \quad k = 1, \ldots, n. \tag{5.22}$$

The hyperplane, for which the above inequalities (5.22) hold, is called **canonical hyperplane**.

Now let us formulate mathematically our optimization problem:
(Problem P1 by Verri [7]). Minimize the weight vector $\mathbf{a}$ subject to the constraints (5.22):

Problem P1
Minimize $\frac{1}{2}\|\mathbf{a}\|^2$
subject to: $z_k g(\mathbf{y}_k) \geq 1, \quad k = 1, \ldots, n$

The solution to our optimization problem is given by the saddle point of the primal Lagrangian (primal problem) {*primal*}:

$$L(\mathbf{a}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{a}\|^2 - \sum_{k=1}^{n} \alpha_k [z_k(\mathbf{a}^T \mathbf{y}_k + b) - 1], \tag{5.23}$$

with $\alpha_k \geq 0$ denoting the Lagrange multipliers.

The variables to optimize are: the weight vector $\mathbf{a}$ and the bias b (primal variables) and the Lagrange constraints α_k, $\alpha_k \geq 0$, $k = 1, \ldots, n$. We seek to minimize $L(..)$ with respect to the weight vector $\mathbf{a}$ and the bias b; and to maximize it with respect to the 2nd term on the right (expressing the goal of classifying the points correctly). It may be stated

that the solutions $(\mathbf{a}^*, b^*,\ \boldsymbol{a^*})$ lie at a saddle point of the Lagrangian; at the saddle point $\mathbf{a}$ attains its minimum and $\boldsymbol{\alpha}$ attains its maximum.

Classical Lagrangian duality enables the primal problem (5.23) to be transformed to its *dual* problem, which is easier to solve. The dual problem is formulated as

$$\max_{\boldsymbol{\alpha}}\ W(\boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}} \{ \min_{\mathbf{a},b} L(\mathbf{a}, b, \boldsymbol{\alpha})\}$$

This means that we find firstly minimum of $L(\mathbf{a}, b, \boldsymbol{\alpha})$ with respect to ($\mathbf{a}$,b) and next – with the found solution $\mathbf{a}^*$ – we maximize $L(\mathbf{a}^*, \boldsymbol{\alpha})$ withe respect to $\boldsymbol{\alpha}$.

We use here so called Kuhn–Tucker theorem, which is an extension of Lagrange's multiplier method and allows to optimize problems in which the constraints are inequalities. According to the Kuhn–Tucker theorem, at the solution $\mathbf{a}^*, b$, the derivatives of L with respect to the primal variables must vanish, i.e.

$$\frac{\partial L(\mathbf{a}, b, \boldsymbol{\alpha})}{\partial \mathbf{a}} = \mathbf{0}, \quad \frac{\partial L(\mathbf{a}, b, \boldsymbol{\alpha})}{\partial b} = 0.$$

This condition leads to the solution {*astar*}:

$$\mathbf{a}^* = \sum_{k=1}^{n} \alpha_k z_k \mathbf{y}_k. \tag{5.24}$$

and another constraint $\sum_{k=1}^{n} \alpha_k z_k$=0. We do not obtain at this stage any solution for b.

Looking at the equation (5.24) one may see that the solution vector $\mathbf{a}^*$ is a linear combination of the traning vectors, namely those vectors whose α_k are not equal to zero.

Substituting the solution $\mathbf{a}^*$ into (5.23) we obtain the dual problem (Problem P2 in Verri [7])

Problem P2

Maximize $\quad -\frac{1}{2}\boldsymbol{\alpha}^T \mathbf{D}\boldsymbol{\alpha} + \sum_{k=1}^{n} \alpha_k$

subject to: $\quad \sum_{k=1}^{n} z_k \alpha_k = 0, \quad k = 1, \ldots, n$

$\qquad\qquad \alpha_k \geq 0,$

where $\quad \mathbf{D} = \{D_{ij}\} = \{z_i z_j\ (\mathbf{y}_i \cdot \mathbf{y}_j)\}, \quad (i, j = 1, \ldots, n).$

The corresponding Lagrangian – *to be minimized*, thus taken with the '–' sign – is {*Lp2*}

$$L(\boldsymbol{\alpha}) = +\frac{1}{2} \sum_{k,j} \alpha_k \alpha_j z_k z_j \mathbf{y}_j^T \mathbf{y}_k \ - \ \sum_{k=1}^{n} \alpha_k \ = \ \frac{1}{2}\, \boldsymbol{\alpha}^T \mathbf{D}\boldsymbol{\alpha} \ - \ \mathbf{1}_n^T \boldsymbol{\alpha}. \tag{5.25}$$

The Kuhn-Tucker's theorem implies that the α_i's must satisfy the so–called **Karush–Kuhn–Tucker** (KKT) **complementarity conditions**, which are formed from the product of the Lagrange multipliers and the corresponding inequality constraints entering the Lagrangian of the primal problem, taken with an equality sign[5]. For our problem this results in the equality {*KT3* }:

$$\alpha_k^* [z_k g(\mathbf{y}_k) - 1] = 0, \quad (k = 1, \ldots, n). \tag{5.26}$$

[5] The Kuhn-Tucker theorem states that the Lagrange parameters can be non-zero only, if the corresponding inequality constraints – entering the primal Lagrangian – taken at the solution, are equality constraints.

The above condition implies, that either the Lagrange multipliers α_k^* or the the expression inside the square brackets [...] have to be zero.

The bracket [...] can be zero only for points lying on the margins, i.e. for the support points. Then the α_k^* may be strictly positive.

For other points, not being support vectors and lying beyond the margins, to satisfy the KKT condition, the multipliers α_k must be equal to 0.

Thus most of the constraints α_k will be equal to zero. Let n_{SV} denotes the number of support vectors, and $\mathcal{SV}$ the set of their indices.

The multipliers $\alpha_k' s$ are usually obtained from the Lagrangian (5.25) using the method of *quadratic programming* [6]. Let $\boldsymbol{\alpha^*}$ denote the solution of problem P2. Then the solution $\mathbf{a}^*$ given by eq. (5.24) can be rewritten as {*astar1*}:

$$\mathbf{a}^* = \sum_{k \in \mathcal{SV}} \alpha_k^* z_k \mathbf{y}_k. \tag{5.27}$$

We need still determine the bias b. Is is established from the KKT condition eq. (5.26) stating that at the solution the following equalities hold:

$$\alpha_j^* [z_j(\mathbf{a}^* \cdot \mathbf{y}_j) + b) - 1] = 0, \quad (j = 1, \ldots, n).$$

Points, for which the expression in square brackets [...] is equal 0, are support vectors lying on the margin. Thus we have

$$z_j \mathbf{a}^* \cdot \mathbf{y}_j + b^* z_j = 1, \quad \forall j \in \mathcal{SV},$$

wherefrom

$$b^* = z_j - \mathbf{a}^* \cdot \mathbf{y}_j.$$

Taking the average, we obtain ($n_{\mathcal{SV}}$ denotes the number of support vectors)

$$b^* = \frac{1}{n_{\mathcal{SV}}} \sum_{j \in \mathcal{SV}} z_j - \frac{1}{n_{\mathcal{SV}}} \mathbf{a}^* \cdot \sum_{j \in \mathcal{SV}} \mathbf{y}_j.$$

Substituting for $\mathbf{a}^*$ the formula (5.27) we obtain

$$b^* = \frac{1}{n_{\mathcal{SV}}} \sum_{j \in \mathcal{SV}} z_j - \frac{1}{n_{\mathcal{SV}}} \Big(\sum_{k \in \mathcal{SV}} \alpha_k^* z_k \mathbf{y}_k \Big) \cdot \sum_{j \in \mathcal{SV}} \mathbf{y}_j.$$

Finally {*bstar1* }:

$$b^* = \frac{1}{n_{\mathcal{SV}}} \sum_{j \in \mathcal{SV}} z_j - \frac{1}{n_{\mathcal{SV}}} \Big[\sum_{k \in \mathcal{SV}} \sum_{j \in \mathcal{SV}} \alpha_k^* z_k (\mathbf{y}_k \cdot \mathbf{y}_j) \Big]. \tag{5.28}$$

Thus, the separating hyperplane is determined only by few data points, the support vectors. Other data points do not matter, provided that they are located at the proper side of the hyperplane.

We do not need individual transformed data vectors $\mathbf{y}_k, \mathbf{y}_j$, only their dot product $(\mathbf{y}_k \cdot \mathbf{y}_j)$.

[6] Matlab offers for that purpose the function quadprog from the Matlab Optimization Toolbox. The classification toolbox 'Patt' offers for this purpose two other methods: 'perceptron' and 'Lagrangian SVM', implemented directly inside the procedure 'SVM'

The complexity of the resulting classifier is characterized by the number of support vectors rather than the dimensionality of the transformed space.

As a result, it may be inferred that SVMs tend to be less prone to problems of overfitting than some other methods.

The support vectors define the optimal separating hyperplane and are the most difficult pattern to to classify. At the same time they are the most informative for the classification task.

It may be shown that the upper bound on the expected error rate of the classifier depends linearly upon the expected number of support vectors.

XOR example.

5.10 Support Vector Machines II. Non-Separable Samples

In the previous subsection, dealing with separable samples, the classifier $g(\mathbf{y})$ has satisfied the following margin inequalities[7]:

$$g(\mathbf{y}_k) \geq +1, \quad \text{for } \mathbf{y}_k \in \omega_1 \ (= \text{ class } \mathcal{C}_1), \quad \text{and } g(\mathbf{y}_k) \leq -1, \quad \text{for } \mathbf{y}_k \in \omega_2 \ (= \text{ class } \mathcal{C}_2),$$

that – after introducing the target variables z_k[8] – were combined together as eq. (5.22)

$$z_k g(\mathbf{y}_k) \geq 1,$$

The above inequalities were derived on the assumption of separable samples. Now we will consider the situation, when the training data for class ω_1 and ω_2 are not separable, which means that they do not satisfy the above inequalities (5.22) stating that $z_k g(\mathbf{y}_k) \geq 1$. Such a situation is depicted in Figure 5.6. From the configuration of points presented in that figure, obviously *two* data points have surpassed their margins and have deviated in the direction of the opposite class. We have to relax for them the margin inequalities (5.22).

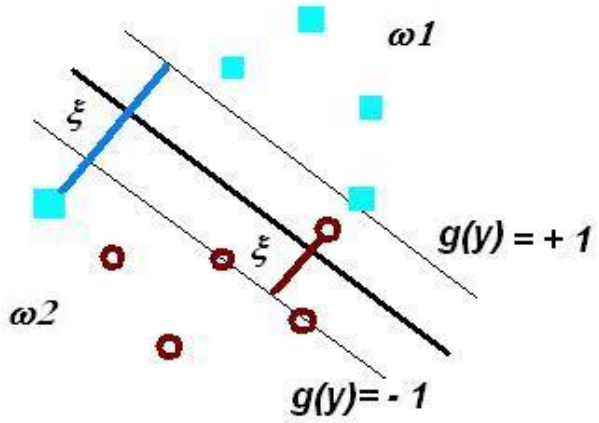


Figure 5.6: The case of non-separable samples. Values provided by the separating hyperplane $g(\mathbf{y}) = 0$. The data vectors are allowed to deviate from their margins (i.e. values $g(\mathbf{y}) = +1$ or $g(\mathbf{y}) = -1$) by a slack variable $\xi > 0$. File sv2.jpg

[7] in other words, the classifier $g(.)$ is calibrated in such a way that the margin is equal to 1, see eq. 5.22

[8] where $z_k = +1$ for $\mathbf{y}_k \in \omega_1$ $(=$ class $\mathcal{C}_1)$, and $z_k = -1$ for $\mathbf{y}_k \in \omega_2$ $(=$ class $\mathcal{C}_2)$

Dealing the case of nonseparable samples, Cortes and Vapnik [9] introduced the following inequalities $\{ksi\}$:

$$z_k g(\mathbf{y}_k) \geq 1 - \xi_k, \text{ with } \xi_k \geq 0, \ k = 1, \ldots, n. \tag{5.29}$$

The new introduced variables are called **slack** variables.

The slack variables allow to take into account all samples that have a geometrical margin less than $1/\|\mathbf{a}\|$, and a functional margin (with respect to the function $g(.)$) less then 1. For such samples the slack variables $\xi_k > 0$ denote the difference between the value $g(\mathbf{y}_k)$ and the appropriate margin.

Additionally, we introduce another constraint stating that the sum of the slack variables is not to big. We express the new constraint in the form

$$C \sum_k \xi_k.$$

The constant C plays here the role of a regularization parameter. The sum $\sum_k \xi_k$ expresses our tolerance in the values of the classifier g for misplaced data vectors (i.e. those that are on the wrong side of the separating hyperplane).

Now our optimization problem is (problem P3 in Verri [7])

Problem P3
Minimize $\frac{1}{2}\|\mathbf{a}\|^2 + C \sum_k \xi_k$
subject to: $z_k g(\mathbf{y}_k) \geq 1 - \xi_k$,
$\xi_k \geq 0$

Again, to solve the problem, we use the Kuhn–Tucker theorem. The primal form of the Lagrangian now becomes $\{Lp3\}$:

$$L(\mathbf{a}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\|\mathbf{a}\|^2 + C \sum_{k=1}^{n} \xi_k \ - \ \sum_{k=1}^{n} \alpha_k [z_k(\mathbf{a}^T \mathbf{y}_k + b) \ - \ 1 \ + \xi_k] \ - \ \sum_{k=1}^{n} \beta_k \xi_k. \tag{5.30}$$

which may be rewritten as $\{Lp3a\}$:

$$L(\mathbf{a}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{1}{2}\|\mathbf{a}\|^2 + C \sum_{k=1}^{n} \xi_k - \sum_{k=1}^{n} \alpha_k [z_k(\mathbf{a}^T \mathbf{y}_k + b) - 1\,] - \sum_{k=1}^{n} \alpha_k \xi_k - \sum_{k=1}^{n} \beta_k \xi_k. \tag{5.31}$$

The β_k's are Lagrange multipliers to ensure positivity of the ξ's. As before, in the case of separable samples, classical Lagrangian duality enables the primal problem to be transformed to the dual problem. The dual problem is given by $\{duall\}$:

$$\max_{\boldsymbol{\alpha}, \boldsymbol{\beta}} \ W(\boldsymbol{\alpha}, \boldsymbol{\beta}) = \max_{\boldsymbol{\alpha}, \boldsymbol{\beta}} \{ \min_{\mathbf{a}, b, \boldsymbol{\xi}} L(\mathbf{a}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) \}. \tag{5.32}$$

This means that firstly we minimize L with respect to $\mathbf{a}, b, \boldsymbol{\xi}$, which results in obtaining as solution $\mathbf{a}^*$ and eventually $\boldsymbol{\xi}^*$, and next we maximize $W(\boldsymbol{\alpha}, \boldsymbol{\beta}) = L(\mathbf{a}^*, b, \boldsymbol{\xi}^*, \boldsymbol{\alpha}, \boldsymbol{\beta})$ with respect to $\boldsymbol{\alpha}, \boldsymbol{\beta}$.

Taking the derivatives of L with respect to $\boldsymbol{a}$, b and $\boldsymbol{\xi}$ we obtain

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{a}} &= \mathbf{a} - \sum_{k=1}^{n} \alpha_k z_k \mathbf{y}_k = 0. \\ \frac{\partial L}{\partial b} &= \sum_{k=1}^{n} \alpha_k z_k = 0. \\ \frac{\partial L}{\partial \boldsymbol{\xi}} &= C\mathbf{1}_n - \boldsymbol{\alpha} - \boldsymbol{\beta} = 0. \end{aligned}$$

Equating the derivatives $\partial L/\partial \mathbf{a}$ to zero we obtain the solution for $\mathbf{a}^*$, which is the same as in the separable case (see eq. 5.24)

Equating the derivative $\partial L/\partial b$ to zero we obtain the constraint $\sum_{k=1}^{n} \alpha_k z_k = 0$, which again is the same as in the separable case.

Hence

$$\begin{aligned} \text{from } \partial L/\partial \mathbf{a}: \quad \mathbf{a}^* &= \sum_k \alpha_k z_k \mathbf{y}_k, \\ \text{from } \partial L/\partial b: \quad \sum_{k=1}^{n} \alpha_k z_k &= 0 \\ \text{from } \partial L/\partial \boldsymbol{\xi}: \quad \beta_k^* &= C - \alpha_k, \quad k = 1, \ldots, n. \end{aligned}$$

Taking into account the equality $C = \boldsymbol{\alpha} + \boldsymbol{\beta}$ and the inequalities $\boldsymbol{\alpha} \geq 0,\ \boldsymbol{\beta} \geq 0$ we obtain that the α_k's satisfy the so called **box constraints** $\{boxconstr\}$:

$$\forall_k \quad 0 \leq \alpha_k \leq C\,. \tag{5.33}$$

Substituting the β_k's into the modified Lagrangian (5.31) we state that expressions containing C and $\boldsymbol{\xi}$ do vanish and we are left with the simplified Lagrangian $\{Lp3simple\}$:

$$L(\mathbf{a}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{a}\|^2 \; - \; \sum_{k=1}^{n} \alpha_k [z_k(\mathbf{a}^T \mathbf{y}_k + b) \; - \; 1\,]. \tag{5.34}$$

Substituting the solution $\mathbf{a}^*$ into the simplified Lagrangian (eq. (5.34) above) we obtain the dual problem (Problem P4 in Verri [7])

Problem P4

Maximize	$-\frac{1}{2}\boldsymbol{\alpha}^T \mathbf{D} \boldsymbol{\alpha} + \sum_{k=1}^{n} \alpha_k$
subject to:	$\sum_{k=1}^{n} z_k \alpha_k = 0, \quad k = 1, \ldots, n$
	$0 \leq \alpha_k \leq C$
where	$\mathbf{D} = \{D_{ij}\} = \{z_i z_j \, (\mathbf{y}_i \cdot \mathbf{y}_j)\} \; (i, j = 1, \ldots, n)$

The dual problem P4 above has the same form as the problem P2 for the separable case, with the only change that now we have an upper bound on the α_k's (they satisfy the box constraints $0 \leq \alpha_k \leq C$).

The corresponding Lagrangian – to be minimized (we have changed the sign) – can be rewritten as $\{Lp4\}$:

$$L(\boldsymbol{\alpha}) = \frac{1}{2} \sum_{k,j} \alpha_k \alpha_j z_k z_j \mathbf{y}_j^T \mathbf{y}_k \; - \; \sum_{k=1}^{n} \alpha_k \;, \tag{5.35}$$

Again, the Lagrangian (eq. 5.35 above) has the sam form as the corresponding Lagrangian (eq. 5.25) in the separable case, with the only change that now we have an upper bound on the $\alpha_k' s$. The solution $\boldsymbol{\alpha}^*$ can be obtained, e.g., using the method of quadratic programming (see the comments about solving equation 5.25).

The Karush–Kuhn–Tucker (KKT) conditions are

$$\begin{aligned} \text{I}: \quad & \alpha_k^*(z_k g(\mathbf{y}_k) - 1 + \xi_k) &= 0, \\ \text{II}: \quad & \beta_k^* \xi_k = (C - \alpha_k^*)\xi_k &= 0. \end{aligned}$$

I. Analyzing the first KKT condition, we state that $\alpha_i > 0$ only when [..]=0. Patterns, for which $\alpha_k > 0$ are termed the **support vectors**. Only these patterns enter into the evaluation formula for the solution $\mathbf{a}^*$.

II. Analyzing the second KKT condition, we state that $\xi_i > 0$ only when $C = \alpha_i$. Alternatively (i.e. when $\alpha < C$), the slack variables have to be equal to 0.
Considering various values of ξ_i combined with $\alpha_i > 0$ we have the cases:
• If $\xi_i = 0$, the corresponding point is on the margin, because then $[z_i g(\mathbf{y}_i) = 1]$
• If $0 < \xi_i < 1$, then we have $[0 < z_i g(\mathbf{y}_i) < 1]$, the corresponding point $\mathbf{y}_i$, although having a slack, is classified correctly
• If $\xi = 1$, the point $\mathbf{y}_i$ is on the separating hyperplane, because then $[z_i g(\mathbf{y}_i) = 1 - 1 = 0]$.
• If $\xi > 1$, then $[z_i g(\mathbf{y}_i) < 0]$, the point $\mathbf{y}_i$ is on the wrong side of the separating hyperplane and is evidently wrongly classified.

Define by $\mathcal{SV}$ the set of data vectors, for which $0 < \alpha_i \leq C$. These vectors are termed support vectors.

The $\mathcal{SV}$ set designates the solution a^*.

Define by $\overline{\mathcal{SV}}$ the set of data vectors, for which $0 < \alpha_i < C$. These data vectors have slack variables equal to zero and are located on the margin (because they satisfy $z_i g(\mathbf{y}_i) = 1$ by the I KKT condition).

The $\overline{\mathcal{SV}}$ set designates the solution b^*.

The support vectors satisfying $0 < \alpha_k < C$ must have $\xi_k = 0$, which means that they lie on the canonical hyperplane at a distance $1/\|\mathbf{a}\|$ from the separating hyperplane.

Non–zero slack variables can only occur when $\alpha_k = C$. In this case the points $\mathbf{y}_k$ are misclassified if $\xi_k > 1$. If $\xi_k < 1$, they are classified correctly, but lie closer to separating hyperplane than the distance $1/\|\mathbf{a}\|$.

If $\xi_k = 0$, the point $\mathbf{y}_k$ is located on the decision boundary and might belong both to class 1 and class 2.

The only free parameter is C. It may be chosen by varying C through a range of values and monitoring the performance of the classifier on a separate validation set, or — using cross validation.

The solution ($\mathbf{a}^*$) is the same as in the separable case, see equations 5.27:

$$\mathbf{a}^* = \sum_{k \in \mathcal{SV}} \alpha_k^* z_k \mathbf{y}_k.$$

The solution (b^*) is derived analogously as in the separate samples case (see equations 5.28), however using data vectors only from the set $\overline{\mathcal{SV}}$. We obtain then

$$b^* = \frac{1}{n_{\overline{\mathcal{SV}}}} \sum_{j \in \overline{\mathcal{SV}}} z_j - \frac{1}{n_{\overline{\mathcal{SV}}}} \left[\sum_{k \in \mathcal{SV}} \sum_{j \in \overline{\mathcal{SV}}} \alpha_k^* z_k (\mathbf{y}_k \cdot \mathbf{y}_j) \right].$$

Again one may see that the evaluation of the classifier $g(\mathbf{y})$ requires only the scalar products (dot products) $\mathbf{y}_i^T \mathbf{y} = (\mathbf{y}_i \cdot \mathbf{y})$ and never the underlying values $\mathbf{y}_i$ and $\mathbf{y}$.

5.11 The kernel trick and MatLab software for SVM

Properties of kernels

The classifier $g(\mathbf{y})$ requires only the evaluation of scalar products $(\mathbf{y}_j \cdot \mathbf{y}) = \mathbf{y}_j^T \mathbf{y},\ j \in \mathcal{SV}$ and never the pattern $\mathbf{y} = \varphi(\mathbf{x})$ in its explicit form.

When using special functions $\varphi(.)$ termed *kernels*, the scalar product can be evaluated in a simple way as

$$k(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}_j). \tag{5.36}$$

E.g., using the polynomial kernel $k(x, y) = (\mathbf{x} \cdot \mathbf{y})^d$ it can be shown that it corresponds to mapping $\mathbf{x}, \mathbf{y}$ into a space spanned by all products of exactly d dimensions.

The formula (5.36) is valid for all kernels satisfying the Mercer's theorem.

The kernels should satisfy the so called Mercer's condition (see, e.g., Gunn[5] or Shawe-Taylor and Christianini [9]).

Example of the Polynomial kernel transformation – from Verri [7].

We have 2 points: $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$. From these we construct polynomial of degree 2:
$z(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$ and
$z(\mathbf{y}) = (1, \sqrt{2}y_1, \sqrt{2}y_2, y_1^2, y_2^2, \sqrt{2}y_1y_2)$.
The corresponding kernel equals
$K(\mathbf{x}, \mathbf{y}) = z(\mathbf{x}) \cdot z(\mathbf{y}) = (1 + \mathbf{x} \cdot \mathbf{y})^{\mathbf{2}}$.

Software for SVM

The classification toolbox 'Patt' by David G. Stork and Elad Yom-Tov

The 'Patt' toolbox [3] uses the kernels:

- Gaussian (RBF) `RBF`, one parameter: width of the Gaussian kernel
- Polynomial `Poly`, 1 parameter: polynomial degree
- Linear `Linear`, no parameter needed
- Sigmoid `Sigmoid`, two parameters: [slope constant]

To solve the optimization problems **P2** or P4, the function SVM offers the following solvers:
Quadprog, taken from the MatLab Optimization Toolbox vs. 6;
Perceptron, implemented directly inside the function 'SVM'; the weights are updated using the worse classified pattern;
Lagrangian, algorithm from Mangasarian & Musicant, Lagrangian Support Vector Machines.

The function SVM may be used from the GUI, or directly by writing some macro instructions in MatLab.

The GUI can be used only for 2-dimensional data; then we obtain a graphical visualization of the decision boundary, of the margins and support vectors.

When using data with more then d=2 variables, we obtain only numerical results, which eventually we might visualize using own macros.

An example of calculations using own macro:

[9] J. Shawe-Taylor and N. Christianini, Kernel Methods for Pattern Analysis, Cambridge University Press, 2004

```
      % calculations using SVM from 'Patt'
      % needs data of size (nxd), y of size(nx1)
      % y  has elements '0' and '1'
C=300: % slack variable, can be 'inf'
par='[''Poly'',2,''Quadprog'',C]';
[test_targets,a_star]=SVM(data',z',data',par);
length(find(z'~=test_targets'))
S=find(z'~=test_targets');
disp([num2str(length(S)), ' misclassified]')
```

The svm toolbox developed by Steve Gunn

This toolbox [5] offers for use the following kernels:

```
%  Values for ker: 'linear'  -
%                  'poly'    - p1 is degree of polynomial
%                  'rbf'     - p1 is width of rbfs (sigma)
%                  'sigmoid' - p1 is scale, p2 is offset
%                  'spline'  -
%                  'bspline' - p1 is degree of bspline
%                  'fourier' - p1 is degree
%                  'erfb'    - p1 is width of rbfs (sigma)
%                  'anova'   - p1 is max order of terms
```

The parameters of the kernel functions are denoted by $p1$ and $p2$ and are global variables.

The **calculations** may be performed using **the graphical GUI** called 'uiclass', which – when called the first time – sets the global variables $p1$ and $p2$. The GUI may use the user's data, e.g.:

```
% data should be in MAT format,
% declaration of global variables is not necessary
% X=data; Y=z; save and2.mat X Y -mat;
 uiclass  %activates gui, which asks for data in the '.mat' format
```

Alternatively, The **calculations** may be performed **using a macro** written directly by the user. An example:

```
ker='poly'; C=300; % 'linear', 'rbf', 'sigmoid'; inf
[nsv, alpha, b0] = svc(data,z,ker,C);
[predictedY,err] = svcerror1(data,z,data,z,ker,alpha,b0);
mis=find(z~=predictedY);
disp([num2str(length(mis)), ' misclassified'])
[h] = svcplot(data,z,ker,alpha,b0);% plots only 2-dimensional data
```

The 'data' should have the size (nxd), the 'z' vectors of size (nx1) represents the target vectors, and is composed of elements '1' and '-1' denoting the 1st and 2nd class appropriately. The visualization is only possible for two-dimensional data.

6 Mixture models

Defining the mixture model

Let $\mathbf{t} = (t_1, \ldots, t_d)^T$ denote an observational vector, $\mathbf{t} \in R^d$. Generally, $\mathbf{t}$ has a probability distribution $p(\mathbf{t})$ characterized by a set of parameters $\boldsymbol{\theta}$:

$$\mathbf{t} \sim p(\mathbf{t}) = p(\mathbf{t}; \boldsymbol{\theta}) .$$

E.g. the probability distribution might be the normal distribution $N_d(\boldsymbol{\mu}, \sigma^2)$ with parameters $\boldsymbol{\theta} = ((\mu), \sigma^2)$.

In practice we may have difficulties to model the observed data using a single distribution $p(\mathbf{t}, \boldsymbol{\theta}$. However, often we may do it assuming that the observed data represent *de facto* M different groups of data; each group (j) is modelled by the same probability distribution $p(\mathbf{t})$, using different parameter set $(\boldsymbol{\theta}_j)$. This may be written as the **mixture model** {mix1}

$$p(\mathbf{t}) = \sum_{j=1}^{M} \pi_j p(\mathbf{t}|\boldsymbol{\theta}_j) , \quad \pi_j \geq 0, \quad \sum_j \pi_j = 1 \tag{6.1}$$

where
π_j – denotes the jth mixing proportions, $j = 1, \ldots, M$,
$p(\mathbf{t}|\boldsymbol{\theta}_j)$ – denotes the probability distribution function governing the jth group.

How many parameters characterize the mixture?

We have to know

- M, the number of components of the mixture
- $p(\mathbf{t}|j)$, the functional form of the distribution of $\mathbf{t}$ in the jth group,
- $\boldsymbol{\theta}_j$, the parameters characterizing the conditional distribution function $p(\mathbf{t}|j; \boldsymbol{\theta}_j)$,
- $\pi_1, \ldots, \pi_M$, the mixing coefficients.

Often we assume a given value of M. We assume also the general shape of the distribution function $p(\mathbf{t}|j; .)$. The remaining task is then to estimate the values of the parameters $\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M$ and $\pi_1, \ldots, \pi_M$.

Estimation of the parameters

For an independent sample of N vectors, $(\mathbf{t}_1, \ldots, \mathbf{t}_N)$ drawn from the distribution $p(\mathbf{t}, \boldsymbol{\theta})$ we have {tsample}

$$\mathbf{t}_n \sim p(\mathbf{t}_n; \boldsymbol{\theta}), \quad n = 1, \ldots, N. \tag{6.2}$$

We seek for such values of the parameters which *maximize* the likelihood {L}

$$\mathcal{L} = \prod_{n=1}^{N} p(\mathbf{t}_n) = \prod_{n=1}^{N} \sum_{j=1}^{M} p(\mathbf{t}_n|j; \boldsymbol{\theta}_j)\, \pi_j. \tag{6.3}$$

Usually the solution (i.e. the estimates of the parameters) are obtained by seeking for the *minimum* of the error function E, which is defined as the negative logaritm $-lnL$ {logL}

$$E = -ln\mathcal{L} = -\sum_{n=1}^{N} ln\, p(\mathbf{t}_n) = -\sum_{n=1}^{N} ln \sum_{j=1}^{M} p(\mathbf{t}_n|j; \boldsymbol{\theta}_j)\pi_j . \tag{6.4}$$

The error function E, to be minimized, is a function of the parameters

$$E = E(\mathbf{t}_1, \ldots, \mathbf{t}_N; \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M, \pi_1, \ldots, \pi_M),$$

with the linear constraint $\sum_{j=1}^{M} \pi_j = 1$.

We know from the general optimization theory that at the solution the gradients with respect to the parameters should be equal to zero. This yields the following set of equations

$$\frac{\partial E}{\partial \boldsymbol{\theta}_j} = \mathbf{0}, \quad (j = 1, \ldots, M),$$

$$\frac{\partial E}{\partial \pi_j} = 0, \quad (j = 1, \ldots, M).$$

The solution is obtained in an iterative way and is usually complicated. Usually we solve the equations using the EM algorithm.

For isotropic Gaussian distributions we may obtain a relatively simple explicit solutions which are adjustable sequentially.

The EM algorithm

We suppose that we have a set of 'incomplete' data vectors $\{\mathbf{x}\}$ and we wish to maximize $L(\boldsymbol{\Psi}) = p(\mathbf{x}|\boldsymbol{\Psi})$ where $\boldsymbol{\Psi} = [\pi_1, \ldots, \pi_M; \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_M]$.

Let $\mathbf{y}_i^T = (\mathbf{x}_i^T, \mathbf{z}_i^T)$, where $\mathbf{z}_i = (z_{1i}, \ldots, z_{Mi})^T$ is defined the following way

$$z_{ji} = \begin{cases} 1, & \text{if } \mathbf{x}_i \text{ belongs to the } j\text{th component,} \\ 0, & \text{otherwise.} \end{cases}$$

The EM algorithm is based on the likelihood function and proceeds in two steps:
The E-step – Expectation. We seek for the expectation of the missing values $\mathbf{z}_j$.
The M-step – Maximization. We maximize the likelihood L substituting for the missing values the estimates found in the E–step.

Software

Netlab [10] is a very reliable software for calculating the mixtures.

To get familiar, one may run the demo examples.

demnlab

Choose the section 'Density Modelling and Clustering'

run the examples 'full', 'diagonal', 'spherical', 'EM'

'Neuroscale' and 'GTM' are methods for visualization of multivariate data in a plane using a mixture of RBF functions.

The demos for these methods may be accessed independently from the gui running 'demns1' and 'demgtm2' from the command window.

The demo 'demmlp2' demonstrates the use of mixtures when building an Bayesian classifier. The performance of the constructed Bayesian classifier is compared with that obtained when using a multilayer perceptron (MLP).

7 Comparing and combining classifiers

1. Mc Nemar's test
2. ROC curve
3. Averaging results provided by Committees of networks
4. Re-sampling, Bootstrap, Bagging
5. Stacked generalization
6. Mixture of experts
7. Boosting
8. Model comparison by Maximum Likelihood

7.1 Statistical tests of equal error rate

After Webb [2], p. 267.

Let A and B be two classifiers trained on a train set $\mathcal{D}_{train}$ and tested on a test set $\mathcal{D}_{test}$.

Consider now the performance of the two classifiers on the test set $\mathcal{D}_{test}$.

Let: n_{00} – nb of *m*isclassified vectors by both A and B,
n_{01} – nb of *m*isclassified vectors by A but not A,
n_{10} – nb of *m*isclassified vectors by B but not by A,
n_{11} – nb of *m*isclassified vectors by neither A nor B.

To test whether the differences between the counts $\{n_{ij},\ i,j = 0,1\}$ are statistically significant, we apply the Mc Nemar's test. Calculate the statistics

$$z = \frac{|n_{01} - n_{10}| - 1}{\sqrt{n_{01} - n_{10}}}.$$

Under the null hypothesis that both classifiers have the same error rate, the statistics z has χ^2 distribution with 1 degree of freedom:

$$z|H_0 \sim \chi^2_1.$$

7.2 The ROC curve

The ROC curve was described in Section 2. We are specially concerned with properly recognizing one group of data, e.g. computer attacks.

Say, we have L classifiers serving this purpose. We may depict the behavior of these classifiers (the ability of detecting 'false positives' and 'true positives') in one graph displaying the respective ROC curves.

7.3 Averaging results from Committees of networks

Text based on Bishop, p. 364–369

We train several (L) candidate networks and next combine their results with the hope that this will lead to significant improvements in the predictors on new data, while involving little additional computational effort. We hope also that the performance of such combining will be better than the performance of the best single network.

Let $y_i(\mathbf{x})$ be the prediction done by the i-th network ($i = 1, \ldots, L$).

We have:

$$y_i(\mathbf{x}) = h(\mathbf{x}) + \epsilon_i(\mathbf{x}),$$

where $h(\mathbf{x})$ is the due value of the prediction and $\epsilon_i(\mathbf{x}$ is the error (deviation from $h(\mathbf{x})$) observed for given $\mathbf{x}$.

Let $E_i = \mathcal{E}[\{y_i(\mathbf{x}) - h(\mathbf{x})\}^2] = \mathcal{E}[\epsilon_i^2]$ be the expected error of the ith prediction. Then the average of the expected errors of the L predictors is

$E_{AV} = \frac{1}{L}\sum_{i=1}^{L} E_i = \frac{1}{L}\sum_{i=1}^{L} \mathcal{E}[\epsilon_i]^2$. Thus

$$E_{AV} = \frac{1}{L}\sum_{i=1}^{L} \mathcal{E}[\epsilon_i]^2.$$

This is the average error of the expected predictions by each of the L networks.

Now let us consider the prediction y_{COM} obtained by averaging the L predictions. We have

$$E_{COM} = \mathcal{E}[(\frac{1}{L}\sum y_i(\mathbf{x}) - h(\mathbf{x}))^2] = \mathcal{E}[\epsilon_i^2].$$

Assuming $\mathcal{E}[\epsilon_i] = 0$, $\mathcal{E}[\epsilon_i\epsilon_j] = 0,\ i \neq j$, we obtain

$$E_{COM} = \frac{1}{L^2}\sum_{i=1}^{L} \mathcal{E}[(\epsilon_i)^2] = \frac{1}{L}E_{AV}.$$

The formula above was derived under the assumption that the errors are independent and uncorrelated. This is not always true. However, we may also derive the formula above using the Cauchy inequality

$$(\sum_{i=1}^{L}(\epsilon_i)^2) \leq L\sum_{i=1}^{L}\epsilon_i^2.$$

Using this inequality we obtain

$$E_{COM} \leq E_{AV}.$$

We might also ask, whether taking generally a **weighted** linear combination of the predictions y_i we might further reduced error. The problem, and how to solve it, is described in Bishop [ ]. In any case we have

$$E_{GEN} \leq E_{AV},$$

where E_{GEN} denotes the expected error of the linear combination

$$y_{GEN}(\mathbf{x}) = \sum_{i=1}^{L} \alpha_i y_i(\mathbf{x}), \quad \text{with } \sum_{i=1}^{L} \alpha_i = 1.$$

7.4 Re-sampling: bootstrap and bagging

Bootstrap

We perform the calculations using resampling with replacement. The process of resampling is repeated B times. Let $\boldsymbol{\theta}$ be the statistic of interest. Then the bootstrap estimate $\hat{\boldsymbol{\theta}}^{*(.)}$, of the statistic $\boldsymbol{\theta}$, is simply the mean of the B estimates on the individual bootstrap data sets:

$$\hat{\boldsymbol{\theta}}^{*(.)} = \frac{1}{B}\sum_{b=1}^{B}\hat{\boldsymbol{\theta}}^{*(b)}.$$

Bootstrap variance estimate

$$Var_{boot} = \frac{1}{B}\sum_{b=1}^{B}[\hat{\boldsymbol{\theta}}^{*(b)} - \hat{\boldsymbol{\theta}}^{*(.)}]^2.$$

If the statistic $\boldsymbol{\theta}$ is the mean, then in the limit of $B \to \infty$, the bootstrap estimate of the variance is the traditional variance of the mean.

Generally speaking, the larger the number B of bootstrap samples, the more satisfactory is the estimate of a statistic and its variance.

Bagging

Bagging (from: 'bootstrap aggregation', see Duda p. 476) uses multiple versions of a training set, each created by drawing $n' < n$ samples from $\mathcal{D}$ with replacement.

Each of these bootstrap data sets is used to train a different *component classifiers* and the final decision (classification decision) is based on the vote of each component classifiers.

Traditionally, the component classifiers are of the same general form; merely the final parameter values differ among them due to their different sets of training parameters.

A learning algorithm carried out on a data set may be 'unstable' if 'small' changes in the data lead to significantly different classifiers and relatively 'large' changes in accuracy. In general, bagging improves recognition for unstable classifiers because it effectively averages over such discontinuities. There are no convincing theoretical derivations or simulation studies showing that bagging will help all unstable classifiers, however (quoted after Duda, p.476).

The global decision rule in bagging – a simple vote among the component classifiers – is the most elementary method of pooling or integrating the outputs of the component classifiers.

7.5 Stacked generalization

After Webb [2], p. 290.

Let: $\mathcal{D}$ – the (training) data set, L – nb. of classifiers, V – nb of partitions of the data, $\{\mathbf{x}_i\}$, $\{t_i\}$, $i = 1, \ldots, n$ – the patterns and corresponding targets contained in $\mathcal{D}$.

Divide the data set $\mathcal{D}$ into V partitions.

Execute for $v = 1, \ldots, V$

a) train the classifier no. j ($j = 1, \ldots, L$) on all training data except partition v, thus obtaining the classifier (classification rule) $g_{j(-v)}$;

b) test each classifier, $g_{j(-v)}$, on all patterns in partition v.

This produces a data set of L predictions, $g_j(\mathbf{x}_i)$, on each pattern $(\mathbf{x}_i)$ contained in the training data set.

Then we have the following data:

Training set	New set (predictions)
$\mathbf{x}_1, \quad t_1$	$g_1(\mathbf{x}_1), \quad \ldots, g_L(\mathbf{x}_1)$
$\mathbf{x}_2, \quad t_2$	$g_1(\mathbf{x}_2), \quad \ldots, g_L(\mathbf{x}_2)$
. . .	. . .
$\mathbf{x}_n, \quad t_n$	$g_1(\mathbf{x}_n), \quad \ldots, g_L(\mathbf{x}_n)$

We may now take the set $g_j(\mathbf{x}_i)$, $(j = 1, \ldots, L;\ i = 1, \ldots, n)$ as predictors and t_i as targets and build a new prediction rule. In case, when the predictors are categorial (no.s of assigned classes), this may be done by correspondence analysis. Merz (1999) has done this with an approach based on a multivariate analysis method (*correspondence analysis*).

7.6 Mixture of experts

The method was proposed by Jacobs et al., 1991. see Webb [2], p. 291.

We consider the problem of mapping in which the form of the mapping is different for different regions of the input space. We assign different *expert* networks to tackle each of the different regions and also an extra *gating* network which sees also the input vector, to decide which of the experts should be used to determine the output. The experts and the gating network are determined as part of the learning process.

The system is depicted in Figure 7.1.

The goal of the training procedure is to have the gating network learn an appropriate decomposition of the input space into different regions, with one of the expert networks responsible for generating the outputs for input vectors falling within each region.

The gating network is trained simultaneously with the constituent classifiers. The gating network provides a 'soft' partitioning of the input space, with experts providing local predictions.

In the basic approach, the output of the ith expert, $\mathbf{o}_i(\mathbf{x})$, is a generalized linear function of the input, $\mathbf{x}$

$$\mathbf{o}_i(\mathbf{x}) = \mathbf{g}(\mathbf{x}; \boldsymbol{\theta}_i)$$

where $\boldsymbol{\theta}_i$ is the vector of parameters associated with the ith expert and $\mathbf{g}(.)$ is a fixed nonlinear function.

The gating network is also a generalized linear function, $\mathbf{w}$, of the input $\mathbf{x}$. The gating network sees the input and makes a soft partitioning of the input space, into L parts, by assigning a proper weight to the given $\mathbf{x}$. The weight for the ith component network is assigned by the function softmax as

$$w_i = w(\mathbf{x}, \boldsymbol{\alpha}_i) = \frac{\exp(\boldsymbol{\alpha}_i^T \mathbf{x})}{\sum_{k=1}^{L} \exp(\boldsymbol{\alpha}_k^T \mathbf{x})}, \quad i = 1, \ldots, L.$$

These outputs of the gating network are used to weight the outputs of the experts to give the overall output, $\mathbf{o}(\mathbf{x})$, of the mixture of experts

$$\mathbf{o}(\mathbf{x}) = \sum_{k=1}^{L} w_k \mathbf{o}_k(\mathbf{x}).$$

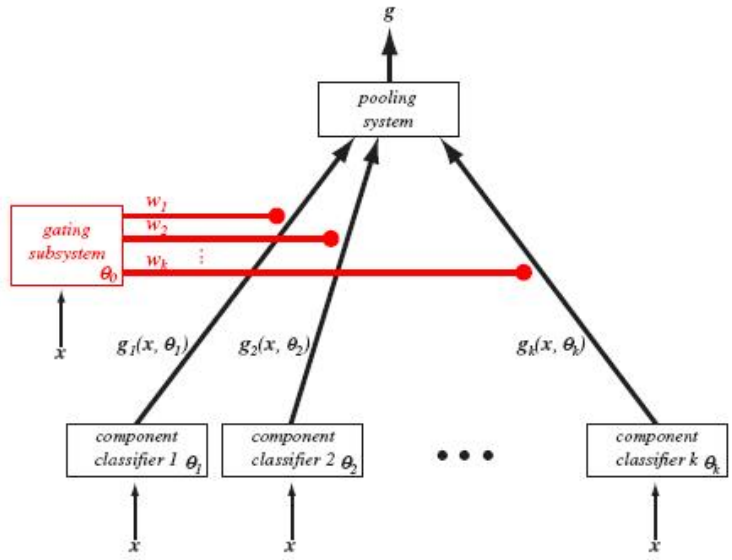


Figure 7.1: Mixture-of-experts. The architecture consists of k component classifiers or 'experts', each of which has trainable parameters $\boldsymbol{\theta}_i, \quad i = 1, \ldots, k$. For each input $\mathbf{x}$ each component classifier i gives estimates of the category membership $g_{ir} = P(\omega_r|\mathbf{x}, \boldsymbol{\theta}_i)$. The outputs are weighted by the gating subsystem governed by parameter vector $\boldsymbol{\theta}_0$ and are pooled for ultimate classification. Graph and description from Duda et al., p. 496. File duda9_19.jpg

The toolbox 'Patt' offers two functions for mixtures of experts:
Components_with_DF – the outputs $\mathbf{o}_k(\mathbf{x})$ of the components are calculated using the logistic discriminant function;
Components_without_DF – the outputs $\mathbf{o}_k(\mathbf{x})$ of the components are calculated using declared classifiers $\mathbf{g}_k(\mathbf{x}; \boldsymbol{\theta}_k)$.

7.7 Boosting, AdaBoost

Proposed by Freund and Schapire (1996), boosting assigns a weight to each pattern in the training set, reflecting its importance, and constructs a classifier using the training set and the set of weights. As compared with bagging, it generates the training sets sequentially, based on the results of the previous iteration. In contrast, bagging generates the training set randomly and can generate the classifiers in parallel.

General description in [3]: AdaBoosts builds a nonlinear classifier by constructing an ensemble of 'week' classifiers (i.e. ones that need perform only slightly better than chance) so that the joint decision has better accuracy on the training set. It is possible to iteratively add classifiers so as to attain any given accuracy on the training set. In AdaBoost each sample of the training set is selected for training the weak with a probability proportional to how well it is classified. An incorrectly classified sample will be chosen more frequently for the training, and will thus be more likely to be correctly classified by the new classifier.

Additional inputs: The number of boosting iterations, the name of week learner and its parameters.

The AdaBoost algorithm implemented in the classification toolbox 'Patt' works according the scheme described in [3], p. 112. The same algorithm, slightly modified, is presented in Table 7.1.

Table 1: The AdaBoost algorithm after Duda et al. [1], p.478.

AdaBoost()

1 **input** $\mathcal{D} = \{\mathbf{x}_i, y_i\}, i = 1, \ldots, n,\ k_{max}$
2 **initialize** $W_1(i) = 1/n,\ i = 1, \ldots, n,$
3 $k \leftarrow 0$;
4 **repeat**
5 $k \leftarrow k + 1$
6 train weak learner $\mathcal{C}_k$ using $\mathcal{D}_k$ sampled from $\mathcal{D}$ according to $W_k(i)$
7 $E_k \leftarrow$ training error of $\mathcal{C}_k$ measured on $\mathcal{D}$ using $W_k(i)$
8 **break if** $E_k = 0$ $\{or\ \ E_k \geq 0.5\}$
9 $\alpha_k \leftarrow \frac{1}{2} \ln[(1 - E_k)/E_k]$
10 $W_{k+1}(i) \leftarrow \frac{W_k(i)}{Z_k} \times \begin{cases} e^{-\alpha_k} & \text{if} \quad g_k(\mathbf{x}_i) = y_i \\ e^{\alpha_k} & \text{if} \quad g_k(\mathbf{x}_i) \neq y_i \end{cases}$
11 Z_k is a normalizing constant
12
13 **until** $k \leftarrow k_{max}$
14 **return** C_k and α_k for $k = 1, \ldots, k_{max}$ (ensemble of classifiers with weights)
15 The value of the final AdaBoost classifier for a new $\mathbf{x}$ is then estimated as
16 $\hat{C}(\mathbf{x}) = \sum_{k=1}^{k_{max}} ln[(1 - E_k)/E_k]\, C_k(\mathbf{x}]$

Initially, all samples are assigned equal weights $w_i = 1/n$. At each stage (k) a classifier (C_k) is constructed using the actual weights $\mathbf{W}_k$ (line 5).

Then the error E_k, taken as the sum of weights of misclassified vectors from the training set, is calculated (line 6), and an error factor $\boldsymbol{\alpha}_k$ (line 7).

Next the weights of misclassified patterns are increased – by multiplying them by a factor $exp(\boldsymbol{\alpha}_k)$ – and the weights of correctly classified patterns are decreased – by multiplying them by a factor $exp(\boldsymbol{\alpha}_k)$. This stage is described as 'boosting' the weights. The boosting should be applied only when $E_k < 0.5$. The effect is that in the next iteration $(k+1)$ the higher weight pattern will influence the learning classifier more, and thus cause the classifier focus more on the misclassification. Generally those are patterns that are nearest to the decision boundary.

The performance of the boosting in subsequent iterations is illustrated in Figure 7.2 (extracted from Raetsch et al., Soft margins for AdaBoost, Machine Learning 42(3), 287–320, March 2001). We may see in that Figure the evolution of the AdaBoost algorithm.

Another illustration of the performance of the AdaBoost algorithm using the 'Patt' toolbox is shown in Figure 7.3, left. The plot shows the 'four_spiral' data containing 1000 data points and subdivided into 2 classes.

Call of the function AdaBoost in 'Patt':

[test_targetsP, E] = ada_boost(train_patterns, train_targets, test_patterns, params);

The function makes break if $E_k = 0$, however does not make break when $E_k \geq 0.5$.

The plot of error E calculated in subsequent iterations is shown in Figure 7.3, right; we see here oscillations around the value $E = 0.5$ beginning from about the 10th iteration. The decision boundary (drawn by the classifier's GUI) is shown in Figure 7.3, left.

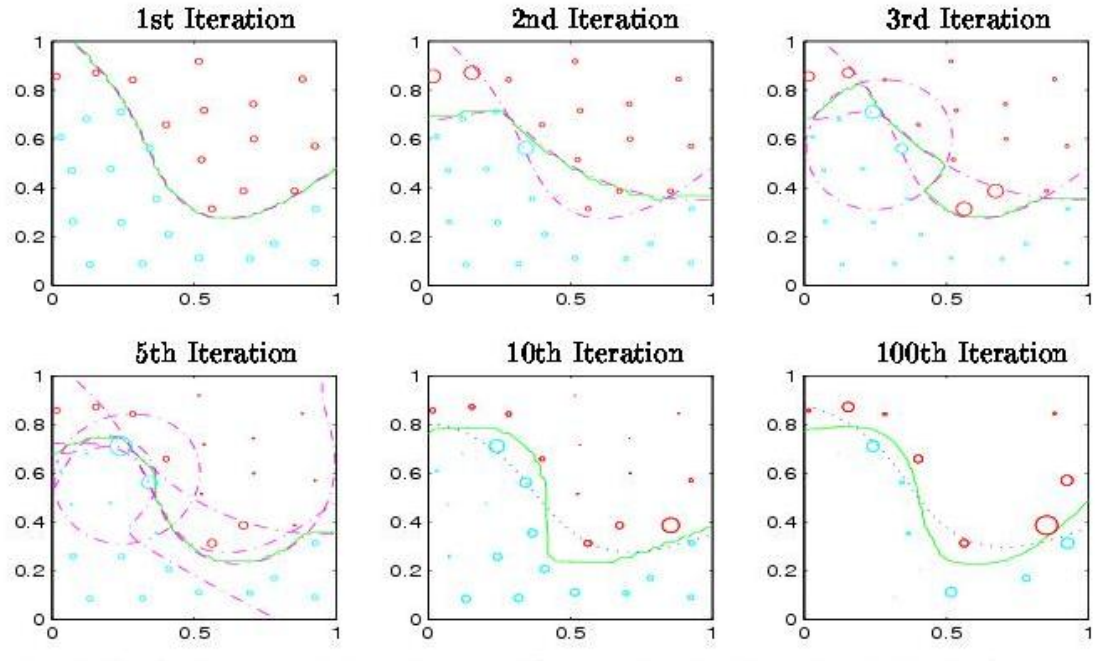


Figure 7.2: Illustration of AdaBoost using a 2D toy data set: The color indicates the label and the diameter is proportional to the weight of the patterns. Weighted patterns in the first, second, third, 10th and 100th iterations are shown. The dashed lines show the decision boundaries of the single classifiers (up to the 5th iteration). The solid line shows the decision line of the combined classifier. In the last two plots the decision line obtained from Bagging is shown for comparison. Figure taken from Raetsch et.al. 2001. File rae.jpg

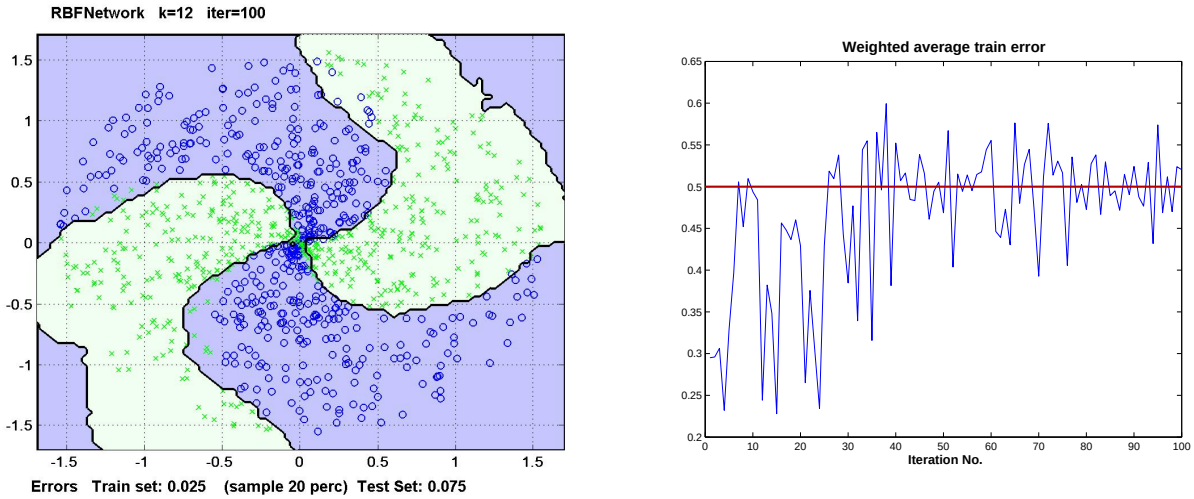


Figure 7.3: AdaBoost for the four_spiral data. Learner: *RBF_network* with 12 RBF centers. **Left:** Decision boundaries after 100 iterations. File rbf100j.jpg. **Right:** Misclassification error for the training data set (taken without weights) noticed in subsequent 100 iterations. File rbfErr100.eps

7.8 Model comparison by Maximum Likelihood

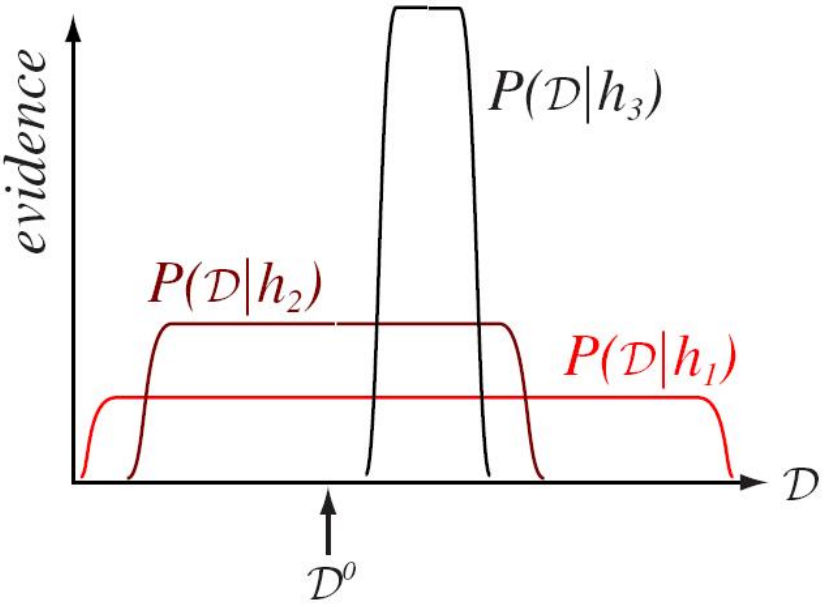


Figure 7.4: The evidence (i.e. probability of generating different data sets given a model) is shown for three models of different expressive power or complexity. Model h_1 is the most expressive, because with different values of its parameters the model can fit a wide range of data sets. Model h_3 is the most restrictive of the three. If the actual observed data is $\mathcal{D}^{(0)}$, then the maximum likelihood model selection states that we should choose h_2, which has the highest evidence. Model h_2 'matches' this particular data set better than do the other two models, and it should be selected. Graph and description from Duda et al., p.487. File duda9_12.jpg

8 Visualization of high-dimensional data

8.1 Visualization of individual data points using Kohonen's SOM

8.1.1 Typical Kohonen's map

The method of Kohonens' self-organizing mas is a non-linear method of projection of data from a multi-dimensional space onto (usually) a plane called the 'map'. The data vectors are represented by their representatives, called codebooks or prototypes. We do not obtain projections of individual data vectors. The map is a stiff one. The projections reflect only the topological neighborhood of the prototypes and not their true distances. However, the u–mat technique permits – by using color painting – to depict in a specific way the approximate distances between neighboring prototypes.

8.1.2 The new method proposed by Liao et.al. 2003

The proposed method permits to project individual data points.

8.1.3 Example 1, Synthetic

8.1.4 Example 2, monitoring gearbox condition

8.1.5 Example 3, visualization of the LowHi erosion data

8.2 The concepts of *volume*, *distance* and *angle* in multivariate space

Some weird facts about the concepts of 'volume', 'distance' and 'angle' in multivariate space may be found in the papers by Verleysen et al. and Herault et al. (see references below).

8.3 CCA, Curvilinear Component Analysis

Based on the papers by Herault et al. – see references below.

8.3.1 The principle of CCA

8.3.2 Scene classification

An image is analyzed by a bank of spatial filters, according to four orientation and 5 frequency bands, ranging from very low spatial frequencies to medium ones. The global energies of the 20 filters' outputs constitute a 20–dimensional feature space. Using CCA, Herault et.al. obtained the visualization shown in Figure 8.9. The obtained organization of the data is in surprising accordance with some semantic meaning: Natural/Artificial for each side of the grey line, and and Open/Closed along this line.

1. J. Herault, A. Guerin–Dugue, P. Villemain, and unsolved questions. ESANN'2002 proceedings, Bruges (Belgium), d-side publi., 173–184.
2. J. Herault, Aude Oliva, Anne Guerin-Dugue, Scene Categorisation by Curvilinear Component Analysis of Low Frequency Spectra. ESANN'1997 proceedings, Bruges (Belgium), d-side publi., 96–96.

3. G. Liao, S. Liu, T. Shi and G. Zhang, Gearbox condition monitoring using self-organizing feature maps. Proc. Instn. Mech. Engrs Vol. 218 Part C, 119–129. J. Mechanical Engineering Science, ©IMechE 2004,
4. M. Verleysen, D. François, G. Simon, W. Wertz, On the effects of dimensionality on data analysis with neural networks. J. Mirra (Ed.): IWANN 2003, LNCS 2687, 105–112.
5. M. Verleysen, Machine Learning of High-Dimensional Data, PHD Thesis, Universite Catholique de Louvain, Laboratoire de Microelectronique, Louvain-la-Neuve, 2000.

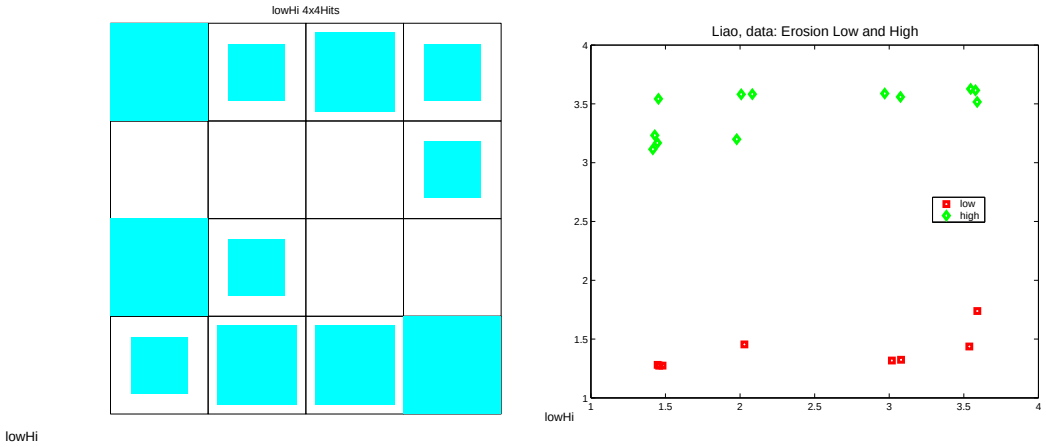


Figure 8.1: Erosion data LowHigh. **Left**: Kohonen map. **Right**: Liao' visualization. Files maplow.eps, liaoLH.eps

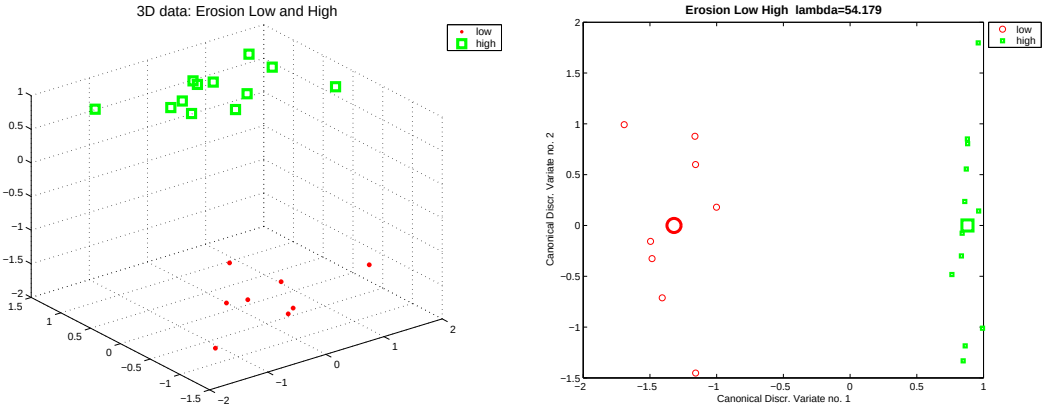


Figure 8.2: Erosion data LowHigh. **Left**: 3D plot. **Right**: Canonical discrimination. Files 3DLH.eps, canLH.eps

`Full version of Chapter 8 may be found in a separated file entitled 'roadmap8'.`