Pattern Recognition >short< Summary

Shorter Summary for Learning

im Fachgebiet Informatik

vorgelegt von: Christopher Syben

Studienbereich: Informatik

This is a 'short' Summary of Pattern Recognition. If you don't get something look up in the normal Summary. ':)'. There are mistakes in the English grammar but the theory should be correct ':)'.

Inhaltsverzeichnis

1 Pattern Recognition Short Summary

1.1 Definitions

$p(y)$: **prior probability** of pattern class y Priors depend not on a current Observation

$p(\vec{x})$: **evidence**. The probability that we observe a certain feature vector $\vec{x}$ independent on his class assignment

$p(\vec{x}, y)$: **joint probability density function** (pdf). More mathematical is $p(\vec{x} \cap y)$ the joint probability of having a feature Vector $\vec{x}$ and the feature vector belongs to class y

$p(\vec{x}|y)$: **class conditional density** the probability that we observe $\vec{x}$ given a certain class y. if you observe feature of a selected class y what is the pdf of the features belonging to this class

$p(y|\vec{x})$: **posterior probability**. the probability of class y given a certain feature vector $\vec{x}$

1.1.1 Bayes Rule

$$p(\vec{x}, y) = p(y) \cdot p(\vec{x}|y) = p(\vec{x}) \cdot p(y|\vec{x})$$

$$p(y|\vec{x}) = \frac{p(y) \cdot p(\vec{x}|y)}{p(\vec{x})}$$

Marginalisation:

$$p(\vec{x}) = \sum_y p(y) \cdot p(\vec{x}|y)$$

1.1.2 Density Function

$$\mathcal{N}(\vec{x}; \vec{\mu}, \Sigma) = \frac{1}{\sqrt{det(2\pi\Sigma_y)}} \cdot e^{-\frac{1}{2}(\vec{x}-\vec{\mu})^T\Sigma^{-1}(\vec{x}-\vec{\mu})}$$

1.1.3 ML-Estimations for Covariance matrices

in general:

$$\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}(\vec{x}_i - \vec{\mu})(\vec{x}_i - \vec{\mu})^T$$

for a specific class:

$$\hat{\Sigma}_1 = \frac{1}{\#\{y_i = 1\}} \sum_{\substack{i=1 \\ y_i=1}}^{N} (\vec{x}_i - \vec{\mu}_1)(\vec{x}_i - \vec{\mu}_1)^T$$

the joint covariance matrix:

$$\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}(\vec{x}_i - \vec{\mu}_{y_i})(\vec{x}_i - \vec{\mu}_{y_i})^T$$

1.1.4 Other interesting stuff

- the Hessian matrix is symmetric !
- mahalanobis distance is the following (if $\Sigma = I$ then its the euclidean):

$$(\vec{x} - \vec{\mu})^T \Sigma^{-1} (\vec{x} - \vec{\mu})$$

Postulates for PR

1 Availability of a representative sample of patterns for the given field of problems

2 A (simple) pattern has features, which characterize its membership in a certain class

3
 - small intra-class distance
 - high inter-class distance

4 A (complex) pattern consists of simpler constituents, which have certain relations to each other. A pattern may be decomposed into these constituents.

5 A (complex) pattern has a certain structure. Not any arrangement of simple constituents is a valid pattern

6 Two patterns are similar if their features or simpler constituents differ only lightly

1.2 Bayesian Decision Rule

- decides for the class y that maximizes the posterior probability

Type: both generative and discriminative

discriminative:

$$y^* = \delta(\vec{x}) = \arg\max_y p(y|\vec{x})$$

generative:

$$y^* = \arg\max_y \frac{p(y) \cdot p(\vec{x}|y)}{p(\vec{x})}$$

optimality: with respect to the 0/1-Loss function. Which means the decision rule comes from the minimization of the average loss

$$l(y_1, y_2) = \begin{cases} 0, & \text{if } y_1 = y_2 \\ 1, & \text{otherwise} \end{cases}$$

$$y^* = \arg\min_y \mathrm{AL}(\vec{x}, y) = ... = \arg\max_y p(y|x) \qquad \text{with} \quad \mathrm{AL}(\vec{x}, y) = \sum_{y'} l(y, y')p(y'|\vec{x})$$

1.2.1 Discriminative Way

1.2.1.1 Logistic Regression

Is a discriminative Model and can classify using the Bayesian decision rule. The posterior probability can be modelled by Sigmoid Function where $F(\vec{x})$ defines the decision Boundary.

$$p(y = 0|\vec{x}) = \frac{1}{1 + e^{-F(\vec{x})}} \qquad \text{sigmoid func:} \quad g(x) = \frac{1}{1 + e^{-x}}$$

using the classes $y \in \{0, 1\}$ gives

$$p(y|\vec{x}) = g(F(\vec{x}))^y (1 - g(F(\vec{x}))^{1-y} \tag{1.1}$$

Training

Their exists different ways to get the decision Boundary $F(\vec{x})$.
For Gaussian distributed Classes the decision boundary is either

quadratic $F(\vec{x}) = \vec{x}^T A\vec{x} + \vec{\alpha}^T\vec{x} + \alpha_0$ or linear $F(\vec{x}) = \vec{\alpha}^T\vec{x} + \alpha_0$.
You can get the parameters for the decision Boundary as follows:

$$F(\vec{x}) = 0 = \log\frac{p(y_0|\vec{x})}{p(y_1|\vec{x})} = \log\frac{p(y_0)p(\vec{x}|y_0)}{p(y_1)p(\vec{x}|y_1)}$$

now models the class conditionals with Gaussians and simplify the formula. For quadratic:

$$A = \frac{1}{2}(\Sigma_1^{-1} - \Sigma_0^{-1})$$
$$\vec{\alpha} = \vec{\mu}_0^T \cdot \Sigma_0^{-1} - \vec{\mu}_1^T \cdot \Sigma_1^{-1}$$

and the constant:

$$\alpha_0 = log\left(\frac{p(y=0)}{p(y=1)}\right) + \frac{1}{2}\Big(log\left(\frac{det(2\pi\Sigma_1)}{det(2\pi\Sigma_0)}\right) + \vec{\mu}_1^T\Sigma_1^{-1}\vec{\mu}_1 - \vec{\mu}_0^T\Sigma_0^{-1}\vec{\mu}_0\Big)$$

linear:

$$A = 0 \qquad \vec{\alpha} = (\vec{\mu}_0 - \vec{\mu}_1)^T \cdot \Sigma^{-1}$$

and the constant:

$$\alpha_0 = log\frac{p(y=0)}{p(y=1)} + \frac{1}{2}((\vec{\mu}_1 + \vec{\mu}_0)^T\Sigma^{-1}(\vec{\mu}_1 - \vec{\mu}_0))$$

With the Maximum-Likelihood approach you can estimate the parameters for the Gaussians.
Another way to get the decision Boundary is to parametrize the sigmoid function and estimate the parameter with ML-estimation.

$$F(\vec{x}) = \vec{\theta}^T\vec{x} \qquad \text{with (for a quadratic dB)} \quad \vec{x} = \begin{pmatrix} x_1^2 \\ x_2^2 \\ x_1 \cdot \vec{x}_2 \\ x_1 \\ x_2 \\ 1 \end{pmatrix} \quad \vec{\theta} = \begin{pmatrix} a_{11} \\ a_{22} \\ a_{12} + a_{21} \\ \alpha_1 \\ \alpha_2 \\ \alpha_0 \end{pmatrix}$$

set up the likelihood function

$$\mathcal{L}(\vec{\theta}^T\vec{x}) = \log\prod_{i=1}^{m} p(y_i|x_i)$$

using 1.1

$$= \sum_{i=1}^{m} \log g(\vec{\theta}^T\vec{x}_i)^{y_i} \cdot (1 - g(\vec{\theta}^T\vec{x}_i))^{1-y_i}$$

the log-likelihood function is concave, to get the $\vec{\theta}$ you can use the iterative Newton-Raphson scheme to optimize the log-likelihood function.
Hint: You can compute the decision Boundary with a linear combination of the component of the feature vector $\vec{x}$ weighted by α (as you have done directly above) put that into the sigmoid function → this is a very simple perceptron !

1.2.2 Generative Way

If the classes are Gaussians then the Bayes-Classificator is called Gaussian Classifier. Model the Class-Conditional probability with Gaussians, and estimate the parameters for the Gaussians with the ML-estimation approach.

$$y^* = \arg\max_y p(y) \cdot p(\vec{x}|y) \qquad \text{with} \quad p(\vec{x}|y) = \mathcal{N}(\vec{x}; \vec{\mu}_y, \Sigma_y)$$

Training

1. Estimate the prior for each class.

2. Estimate the $\vec{\mu}_y$ and Σ_y for each class with the ML-estimation approach:

$$\max \to \mathcal{L}(\vec{\mu}_y, \Sigma_y) = \sum_{i=1}^{m} \log p(x; \vec{\mu}_y, \Sigma_y) = \sum_{i=1}^{m} \log \mathcal{N}(\vec{x}; \vec{\mu}_y, \Sigma_y)$$
$$\vec{\mu}_y = \frac{1}{m} \sum_{k=1}^{m} \vec{x}_k \qquad \Sigma_y = \frac{1}{n} \sum_{k=1}^{n} (\vec{x}_k - \vec{\mu}_y)(\vec{x}_k - \vec{\mu}_y)^T$$

1.2.3 Facts about Gaussian Classifier

- in general the decision Boundary is quadratic
- if all classes share the same covariance, the decision Boundary is linear
- if all covariance matrices are diagonal matrices then you get a Naive Bayes classifier
- if the joint covariance matrix is Σ and the priors are identical.....
- if all covariance matrices are identity matrix, you geht the Nearest Neighbor classifier based on L_2-norm:

$$y^* = \arg\min_y \frac{1}{2}(\vec{x} - \vec{\mu}_y)^T (\vec{x} - \vec{\mu}_y)$$

1.3 Naive Bayes

For Classification Naive Bayes uses also the Bayes decision rule.
Naive Bayes assumes that **all components** of the feature vector $\vec{x} \in \mathbb{R}^d$ are **mutually independent**!
with this assumption you can write the class conditional probability as:

$$p(\vec{x}|y) = \prod_{i=1}^{d} p(x_i|y) \longrightarrow \text{decision rule:} \quad y^* = \arg\max_y p(y) \prod_{i=1}^{d} p(x_i|y)$$

Curse of Dimensionality

The implication of this is that the covariance Matrix is a diagonal Matrix!
The independent assumption from Naive Bayes reduces the parameters which needs to be estimated for the Gaussians extremly: For a normal Gaussian classifier:

$$\vec{\mu}_y \in \mathbb{R}^d \quad \text{and} \quad \Sigma_y \in \mathbb{R}^{d\times d}$$

number need to be estimated with ML approach:

$$d + \frac{d\cdot(d+1)}{2} \quad \text{e.g.: } d = 100 \longrightarrow 100 + \frac{100\cdot(100+1)}{2} = 5150 \text{ Parameters}$$

with the Naive Bayes assumption you have for each component an Gaussian $\mathcal{N}(x_i; \vec{\mu}, \sigma^2)$ whose parameters need to be estimated:

$$p(\vec{x}|y) = \prod_{i=1}^{d} \mathcal{N}(\vec{x}_i; \vec{\mu}_i, \sigma_i^2) \quad \text{with} \quad \vec{\mu}_i \in \mathbb{R}, \sigma_i^2 \in \mathbb{R}$$

number need to be estimated with ML approach:

$$d + d \quad \text{e.g.: } d = 100 \longrightarrow 100 + 100 = 200 \text{ Parameters}$$

Instead of a optimization Problem with 5150 unknown you end up with 200 unknown, which is feasible!

Decision Boundary

$\Sigma_1 \neq \Sigma_2 \longrightarrow$ quadratic decision Boundary

$\Sigma_1 = \Sigma_2 \longrightarrow$ linear decision Boundary

$\Sigma_1 = \Sigma_2 = I \longrightarrow$ Nearest Neighbour Classifier where reference Vectors are the mean Vectors (if the priors are the same)

The decision boundary can written in gernal form:

$$F(\vec{x}) = \log \frac{p(y=0|\vec{x})}{p(y=1|\vec{x})} = \alpha_0 + \sum_{i=1}^{d} \alpha_{0,i}(\vec{x}_i)$$

where $\alpha_{0,i}(..)$ are functions which are only depend on one component.

Statistical Dependencies of limited Order

This is something between the full dependencies of the Gaussian classifier and the fully independent assumption of the Naive Bayes.
This means assume that only n-components are dependent. For example the component is dependent on the component before.
This is Order one:

$$p(\vec{x}|y) = p(x_1|y) \cdot p(\vec{x}_2|y, x_1) \cdot p(x_3|y, x_2) \cdot .... = p(x_1|y) \prod_{i=2}^{d} p(x_i|y, x_{i-1})$$

1.4 Transformations

Feature Transforms are applied to hit one or more of these points:

- Transform the features in a space where they are normally distributed with a Identity Matrix as the Covariance Matrix (To get the NN-Classifier !)
- transform the features in a higher dimensional space, where the decision Boundary is linear
- transform the features in a subspace where the classification problem can be solved sufficiently

Achieve Identity Matrix as covariance

Use the SVD to decompose the covariance Matrix and reformulate it such that you got the transformation:

$$\mathcal{N}(\vec{x};\vec{\mu}_y,\Sigma_y) = \frac{1}{\sqrt{det(2\pi\Sigma_y)}} \cdot e^{-\frac{1}{2}(\vec{x}-\vec{\mu}_y)^T\Sigma_y^{-1}(\vec{x}-\vec{\mu}_y)}$$

$$= \frac{1}{\sqrt{\det 2\pi\Sigma}} \cdot e^{-\frac{1}{2}\left(\underbrace{(D_y^{-\frac{1}{2}}U_y^T)\vec{x}}_{\text{transformed } \vec{x}} - \underbrace{(D_y^{-\frac{1}{2}}U_y^T)\vec{\mu}_y}_{\text{transformed } \vec{\mu}_y}\right)^T \cdot I \cdot \left(\underbrace{(D_y^{-\frac{1}{2}}U_y^T)\vec{x}}_{\text{transformed } \vec{x}} - \underbrace{(D_y^{-\frac{1}{2}}U_y^T)\vec{\mu}_y}_{\text{transformed } \vec{\mu}_y}\right)}$$

the Transformation is

$$\Phi(\vec{x}) = D_y^{-\frac{1}{2}}U_y^T\vec{x}$$

The problem is that the matrices are Class dependent!

LDA

The LDA solves the problem with the class dependent matrices!
Compute the joint covariance Matrix and get the transformation with SVD out of that Matrix, then the transformation is class independent.

$$\hat{\Sigma} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \vec{\mu}_y)(x_i - \vec{\mu}_y)^T \longrightarrow \vec{x}' = \Phi(\vec{x}) = D^{-\frac{1}{2}}U^T\vec{x}$$

put that into the Bayes classifier:

$$y^* = \arg\max_y p(y|\Phi(x)) = \arg\min_y \frac{1}{2}\left\|\Phi(\vec{x}) - \Phi(\vec{\mu}_y)\right\|_2^2 - \log p(y)$$

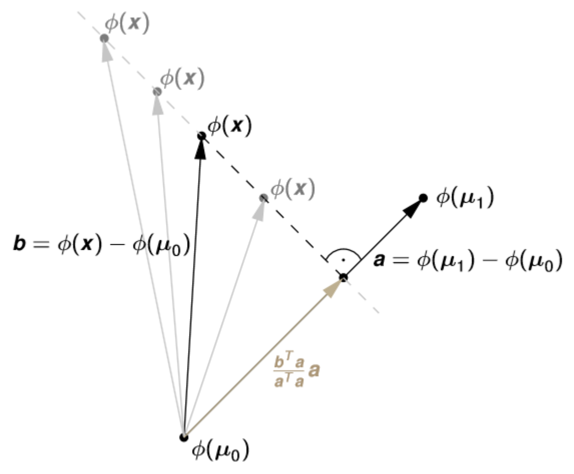
Which its nothing else then the Nearest Neighbour Classifier which take care about the prior probability (regularized)!

Spherical Data

Do the transform based on the joint Covariance matrix, this creates spherical data ! Its a kind of Data normalization. If you use the LDA - transformation without dimensionality reduction, then the class centroids of the spherical data span a $k-1$ subspace, where you can do the classification.

Ranked Reduced LDA

In 1.4 you can see that all points on the line can projected on the vector $\vec{\mu}_0 - \vec{\mu}_1$ and a valid classification is still possible. In a two class problem you can reduce the di-


$\phi(\boldsymbol{x})$
$\phi(\boldsymbol{x})$
$\phi(\boldsymbol{x})$
$\phi(\boldsymbol{x})$
$\phi(\boldsymbol{\mu}_1)$
$\boldsymbol{b} = \phi(\boldsymbol{x}) - \phi(\boldsymbol{\mu}_0)$
$\boldsymbol{a} = \phi(\boldsymbol{\mu}_1) - \phi(\boldsymbol{\mu}_0)$
$\frac{\boldsymbol{b}^T\boldsymbol{a}}{\boldsymbol{a}^T\boldsymbol{a}}\boldsymbol{a}$
$\phi(\boldsymbol{\mu}_0)$

mensionality to one and do the nearest neighbour classification in the 1-D subspace and get the same results.
So you can project to a $K-1$-subspace (where k are the number of classes) and can still Classify with the NN as good as before but in a lower dimension.

Dimension reduction

Ranked Reduced LDA

Find a transformation $\Phi \in \mathcal{R}^{L\times(k-1)}$ that the Interclass distance is maximized and the Intraclass distance is minimized.
Objective Function (M-Classes):

$$\vec{a}* = \arg\max_{\vec{a}} \frac{\vec{a}^T \Sigma_{\text{inter}} \vec{a}}{\vec{a}^T \Sigma_{\text{intra}} \vec{a}}$$

with $\vec{a}$ is the eigenvector of $\Sigma_{\text{intra}}^{-1}\Sigma_{\text{inter}}$ that belongs to the largest eigenvalue

$$\Sigma_{intra} = \frac{1}{m}\sum_i (\vec{x}_i - \vec{\mu}_{y_i})(\vec{x}_i - \vec{\mu}_{y_i})^T \qquad \Sigma_{inter} = \frac{1}{m}\sum_y (\vec{\mu}_y - \vec{\mu})(\vec{\mu}_y - \vec{\mu})^T$$

$$\vec{\mu} = \frac{1}{m}\sum_y \vec{\mu}_y$$

PCA

Find a transformation Matrix Φ that maximizes the spread of features:
Objective Function:

$$\Phi^* = \arg\max_{\Phi} \sum_{i,j} (\Phi(x_i) - \Phi(x_j))(\Phi(x_i) - \Phi(x_j))^T + \lambda(\|\Phi\|_2^2 - 1)$$

Construction of Φ

- the optimization of the objective function leads to an Eigenvalue-/eigenvector problem
$$\Sigma\Phi^T = \lambda'\Phi^T$$
- this can be solved by applying the SVD on the covariance Matrix of the Dataset (needs de-meaned date)
$$\Sigma = \frac{1}{N}\sum_{i=1}^{N} (\vec{x}_i - \vec{\mu})(\vec{x}_i - \vec{\mu})^T$$
- Sort the eigenvectors according to their eigenvalues (descent) !
- Use the most significant M eigenvectors
- The M eigenvectors of Σ becomes the rows of Φ

Covariance matrix is $\Sigma \in \mathcal{R}^{d\times d}; x \in \mathcal{R}^d$ **Differenzen between PCA and LDA**

- PCA does not require a classi

  ed set of feature vectors like LDA
- PCA transformed features are approximately normally distributed (central limit theorem)
- Components of PCA transformed features are mutually independent (cool for a following Naive Bayes)
- PCA transforms to arbitrary dimension, LDA depends on the number of classes (K-1)

1.5 Shape Modelling

Is used to get statistical Models. Using PCA, you can model shapes and their principal components. This gives you a parametrized model, where you can change some parameters and get new models which are realistic.
1. Sample an object at n surface points and encode these points as a feature vector.
2.Compute the covariance matrix from m shapes.
3.compute the principal components to get spectral decomposition($\lambda_i =$ eigenvalues and $\vec{e}_i =$) eigenvectors.

$$\Sigma_L = \sum_i \lambda_i \vec{e}_i \vec{e}_i^T$$

Changing λ_i gives you new realistic models. A Shape can be computed using linear combination of l eigenvectors:

$$\vec{x}^* = \bar{\vec{x}} + \sum_{i=1}^{l} a_i \vec{e}_i \quad \text{where}$$

1.6 Linear Regression

Decision Rule:

$$y^* = \text{sgn}(\vec{\alpha}^T \vec{x} + \alpha_0)$$

Hint: This could be trained with logistic Regression or Gaussians.
Another way to train $\vec{\alpha} \quad \alpha_0$ is the least square estimation, for this you reformulate the expression in the sgn(...) function into a matrix formulation with a parameter $\vec{\theta} = (\vec{\alpha} \quad \alpha_o)^T$. And solve the following problem with SVD

$$X\vec{\theta} = y \quad \text{with} X^+ = (X^T X)^{-1} X^T \quad \text{(pseudo inverse)}$$

estimate $\vec{\theta}$ and solve the linear regression problem:

$$\begin{aligned} \text{objective function: } \hat{\vec{\theta}} &= \arg\min_{\vec{\theta}} \left\| \underbrace{X\vec{\theta} - y}_{\text{residual } r} \right\|_2^2 \\ \text{solution for } L_2\text{-norm: } &= (X^T X)^{-1} X^T y \end{aligned}$$

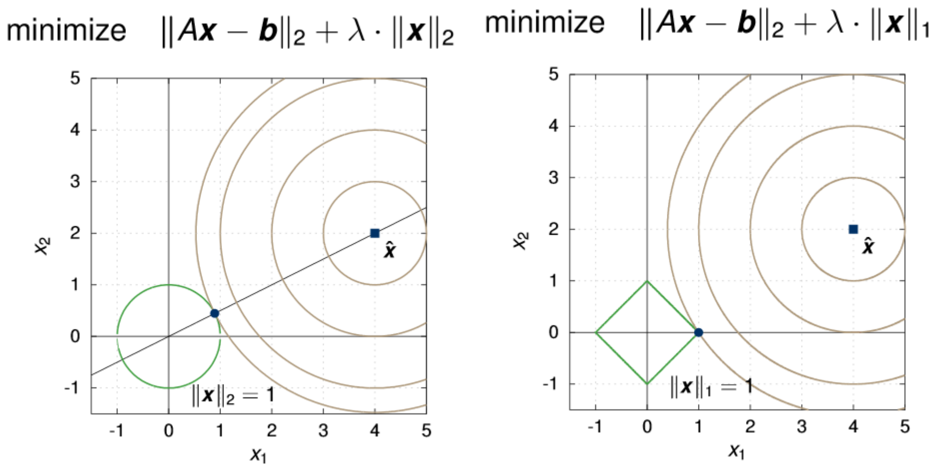
using the L_2-norm and the residual is > 1 you penalize higher deviation a way higher. (outliers have a too high impact !)

Ridge Regression

you need a constraint on $\vec{\theta}$ such that $\left\|\vec{\theta}\right\|_2^2 = 1$

$$\begin{aligned} \text{objective function: } \hat{\vec{\theta}} &= \arg\min_{\vec{\theta}} \left\|X\vec{\theta} - y\right\|_2^2 + \lambda \left\|\vec{\theta}\right\|_2^2 \\ \text{solution for } L_2\text{-norm: } &= (\underbrace{X^T X + \lambda I}_{A})^{-1} X^T y \end{aligned}$$

the more weight you put on the length of $\vec{\theta}$ the more A looks like a diagonal Matrix! In 1.6 you can see the drawn level set and the geometric interpretation, the big blue point is the solution.



Lasso

replace the L_2-norm as a constraint from $\vec{\theta}$ with the L_1-norm, then you get the lasso. The lasso produce sparse solutions, imagine in the 1.6 instead of the unit ball the Diamond from the L_1-norm, then you can see that the solution lies on the peaks $(1,0),(0,1)$ of the Diamond.

$$\hat{\vec{\theta}} = \arg\min_{\vec{\theta}} \left\|X\vec{\theta} - y\right\|_2^2 + \lambda \left\|\vec{\theta}\right\|_1$$

Only if the L_2-norm is used here: $\left\|X\vec{\theta} - y\right\|_2^2$ you get a closed form solution!

1.7 Norms

A norm is a measure for the length of a vector.

- the inner product of vectors $\vec{x}, \vec{v} \in \mathbb{R}^d$ is defined by:

$$\langle \vec{x}, \vec{v} \rangle = \vec{x}^T \vec{v} = \sum_{i=1}^{d} x_i v_i$$

- the inner product of matrices $\mathbf{X}\mathbf{Y}, \in \mathbb{R}^{m \times n}$ is defined by:

$$\langle X, Y \rangle = \operatorname{tr}(\mathbf{X}^T \mathbf{Y}) = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{i,j} y_{i,j}$$

- The Forbenius norm in term of an inner matrix product corresponds to the L_2-norm for vectors:

$$||\mathbf{X}||_F = \sqrt{\langle \mathbf{X}, \mathbf{X} \rangle} = \sqrt{\operatorname{tr}(\mathbf{X}^T \mathbf{X})}$$

L_0-norm: denotes the number of non-zero entries. The L_0-norm is not a norm because it is not homogeneous

L_p-norm: (p > 0)

$$||\vec{x}||_p = \left(\sum_{i=1}^{d} |x_i|^p \right)^{\frac{1}{p}}$$

L_1-norm: sum of absolute values

$$||\vec{x}||_1 = \sum_{i=1}^{d} |\vec{x}_i|$$

L_2-norm: sum of squared values

$$||\vec{x}||_2 = \sqrt{\sum_{i=1}^{d} \vec{x}_i^2}$$

L_∞-norm: maximum norm

$$||\vec{x}||_\infty = \max_i \{|x_i|; i = 1, 2, ..., d\}$$

$L_\mathbf{P}$-norm: $\mathbf{P}$ is a symmetric positive definite matrix
the quadratic $L_\mathbf{P}$-norm is defined by:

$$||\vec{x}||_\mathbf{P} = \sqrt{\vec{x}^T \mathbf{P} \vec{x}} = \sqrt{(\mathbf{P}^{\frac{1}{2}} \vec{x})^T \mathbf{P}^{\frac{1}{2}} \vec{x}} = ||\mathbf{P}^{\frac{1}{2}} \vec{x}||_2$$

Norms can be also used to measure the distance between to vectors $\vec{x}$ and $\vec{v}$:

$$\text{dist}(\vec{x}, \vec{v}) = ||\vec{x} - \vec{v}||$$

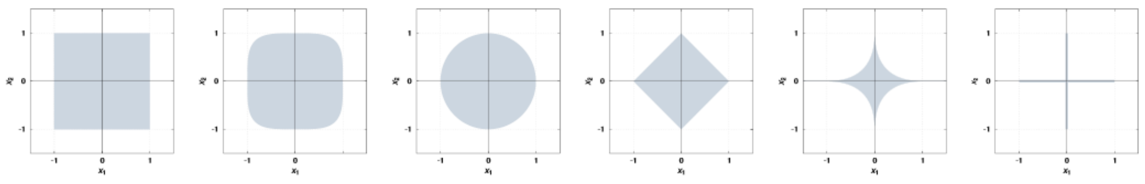
The Malahanobis-Distance is a $L_{\mathbf{P}}$-norm where $\mathbf{P}$ is the inverse of the covariance Matrix Σ:

$$||\vec{x} - \vec{v}||_{\Sigma^{-1}} = \sqrt{(\vec{x} - \vec{v})^T \Sigma^{-1} (\vec{x} - \vec{v})}$$

The interpretation with respect to PR is that points in the direction of the first principal axis of the covariance matrix are less penalized than points in the direction of the last principal axis.

Unit Balls

In 1.7 you see the unit balls for the $L_\infty-, L_4-, L_2-, L_1-, L_{0.5}-, L_0-$norms The



L_P-Norm (P is a Matrix) is usually an ellipsis!

Linear Regression with Norms

$\preceq$ means component-wise smaller or equal. its all about minimizing the residual $\vec{r}$:

$$\textbf{minimize} \left\| \underbrace{A\vec{x} - \vec{b}}_{\text{residual } \vec{r}} \right\|_?$$

if $\vec{b}$ is in the range of A, the residual will be the zero vector!

Chebyshev Linear Regression

Minimization of the residual using $L_\infty-$norm:

$$\text{minimize} \left\| A\vec{x} - \vec{b} \right\|_\infty = \max\{|r_1|, ..., |r_m|\}$$

$$\text{rewritten such that the the problem is convex:}$$

$$\text{minimize } \vec{r} \quad \text{subject to} \quad -r \cdot \mathbf{1} \succeq \mathbf{A}\vec{x} - \vec{b} \preceq r \cdot \mathbf{1}$$

Sum of Absolute Residual

Minimization of the residual using $L_\infty-$norm:

$$\text{minimize} \left\| A\vec{x} - \vec{b} \right\|_1$$

rewritten such that the the problem is convex:

$$\text{minimize} \qquad \underbrace{\vec{1}^T\vec{r}}_{\text{sum of the entries of r}} \qquad \text{subject to} \quad -r \preceq \mathbf{A}\vec{x} - \vec{b} \preceq r$$

Compressed Sensing

Assume you have fewer measurements than required to estimate the parameter vector $\vec{x}$. Solution of the underdetermined case required.

$$minimize \, \|\vec{x}\|_1 \quad \text{subject to } A\vec{x} = \vec{b}$$

That is basically the Lasso!

1.8 Penalty Functions

Minimizing the Penalty function:

$$\text{minimize} \sum_{i=1}^{m} \phi(\vec{r_i}) \quad \text{subject to } \vec{r} = A\vec{x} - \vec{b}$$

The penalty functionϕ assign costs to residuals, if ϕ is convex, the penalty function approximation problem is a convex optimization problem.
Norms are a special case of penalty functions. you can use norms to calculate a value for the residual and minimize that.

The Log Barrier Function

$$\phi_{barrier}(r) = \begin{cases} -a^2 \log(1 - (\frac{r}{a})^2) & \text{if } \; |r| < a \\ \infty & \text{otherwise} \end{cases}$$

the log barrier function only accepts solutions in a tube. This can be used to exlude solutions that would not be feasible.

Dead Zone Linear Penalty Function

$$\phi_{barrier}(r) = \begin{cases} 0 & \text{if } |r| < a \\ |r| - a & \text{otherwise} \end{cases}$$

defines a dead zones where the penalty is 0 and otherwise behaves like the L_1-norm.

1.8.1 The Large Error Penalty Function

$$\phi_{barrier}(r) = \begin{cases} r^2 & \text{if } |r| < a \\ a^2 & \text{otherwise} \end{cases}$$

penalize below a like the L_2-norm and outside with a constant.

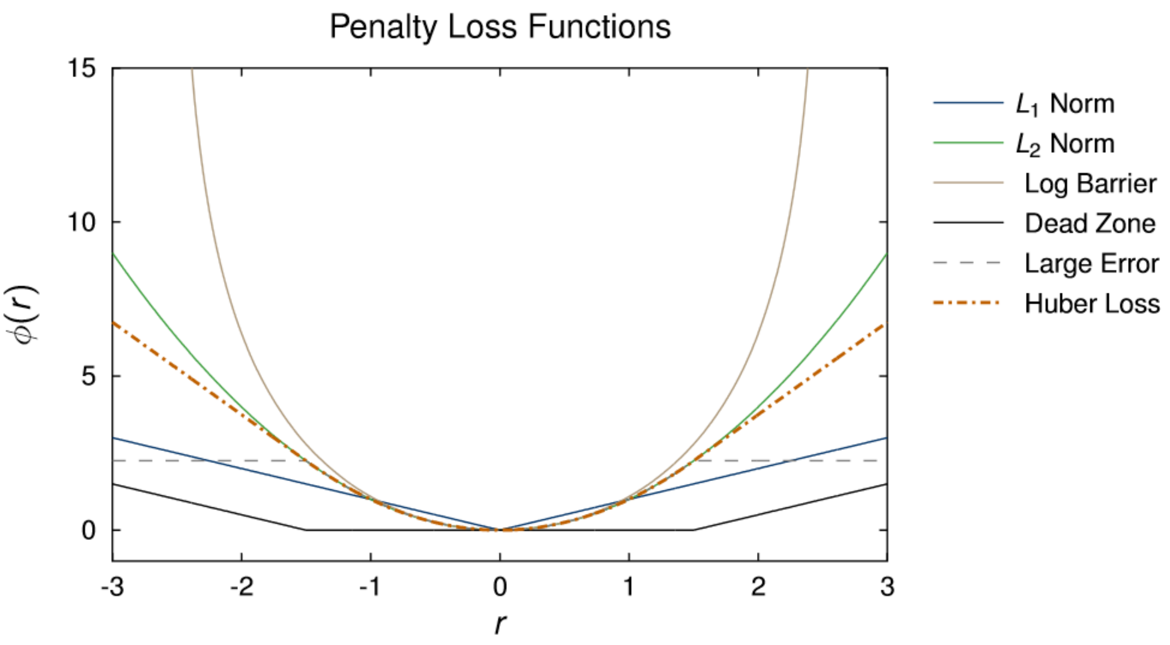
The Huber Function

$$\phi_{barrier}(r) = \begin{cases} r^2 & \text{if } |r| < a \\ a \cdot (2|r| - a) & \text{otherwise} \end{cases}$$

The Huber function is an approxmiation of the absolute value, but is smooth at the origin. It penalize below a like the L_2-norm and over a like L_1. The transition between these two functions is smooth.

Overview Penalty Functions

In 1.8.1 all penalty functions are drawn.



1.9 Perceptron

The Perceptron computes a linear decision Boundary. Decision Rule:

$$y^* = \operatorname{sgn}(\vec{\alpha}^T \vec{x} + \alpha_0)$$

The Perceptron is another way to estimate $\vec{\alpha}$ and α_0.
It have a discrete optimization Problem!
Objective Function:

$$\text{minimize } D(\vec{\alpha}, \alpha_0) = - \sum_{x_i \in \mathcal{M}} \underbrace{y_i \cdot (\vec{\alpha}^T \vec{x} + \alpha_0)}_{\text{always negative}}$$

$\mathcal{M}$ is set of misclassified features. The cardinality of $\mathcal{M}$ changes in each Iteration, due to that the problem is not Linear !
The iterative scheme for each misclassified feature is derivated from the objective function:

$$\begin{pmatrix} \alpha_0^{(k+1)} \\ \vec{\alpha}^{(k+1)} \end{pmatrix} = \begin{pmatrix} \alpha_0^{(k)} \\ \vec{\alpha}^{(k)} \end{pmatrix} + \lambda \begin{pmatrix} y_i \\ y_i \cdot \vec{x}_i \end{pmatrix} \tag{1.2}$$

The initialization should be done with 0, so this value falls out of the equation! The decision Boundary is then:

$$F(\vec{x}) = \underbrace{\left(\sum_{i\in\epsilon} y_i \cdot \vec{x}_i\right)^T}_{\vec{\alpha}} \vec{x} + \underbrace{\sum_{i\in\epsilon} y_i}_{\alpha_0} = \sum_{i\in\epsilon} y_i \cdot \langle \vec{x}_i, \vec{x}\rangle + \sum_{i\in\epsilon} y_i \tag{1.3}$$

where ϵ is the set of indices that required an update.

Notes for the Perceptron

- The parameter $\vec{\alpha}, \alpha_0$ will be calculated through a gradient descent method
- The Perceptron linearly separable classes! (or the algorithm will not end)
- its a non-linear optimization Problem!($\#(\mathcal{M})$ changes each iteration)
- the number of iterations does not depend on the dimension of $\vec{x}$
- its a discrete optimization problem
- the final decision Boundary depends on the Initialization

Algorithm in short words

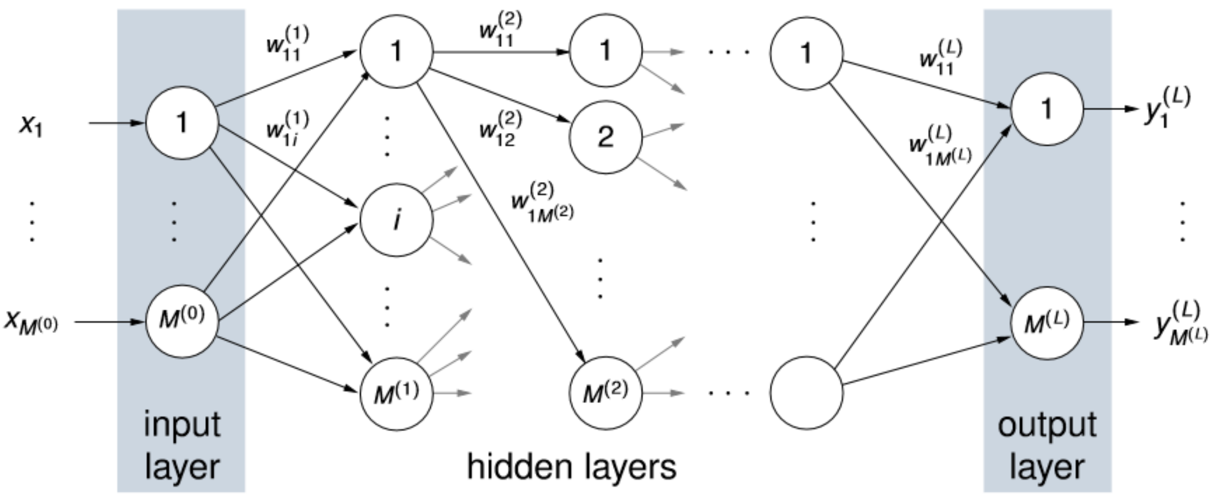
The Algorithm in short words: for each feature vector where $y_i \cdot (\vec{\alpha}^T\vec{x} + \alpha_0) < 0$ use the update scheme and update $\vec{\alpha}, \alpha_0$. This will be done until $y_i \cdot (\vec{\alpha}^T\vec{x} + \alpha_0) > 0$ is fulfilled for all the feature vectors! This will not end if the classes are not linearly separable!

Multi-Layer Perceptron

A Multi-Layer Perceptron consists of several Perceptrons. The topology is shown in 1.9 The weights will be computed with the Backpropagation Algorithm which is a gradient descent method !
One Perceptron in the Multi-Layer Perceptron can be modelled by:

$$net_j^{(l)} = \sum_{i=1}^{M^{(l-1)}} y^{(l-1)} \underbrace{w_{ij}^{(l)}}_{\vec{\alpha}} - \underbrace{w_{0j}^{(l)}}_{\alpha_0} \tag{1.4}$$

$$y_j^{(l)} = f(net_j^{(l)}) \tag{1.5}$$



$f(...)$ is the so called activation function, Rosenblatts Perceptron uses here the sign-function. But its not limited to that, you can also use for example the sigmoid function.
To train the mulit-layer perceptron, use the gradient descent method to ajust the weights:

unconstrained Optimization

The unconstrained optimization is the solution for the minimization Problem:

$$\vec{x}^* = \arg\min_{\vec{x}} f(\vec{x}) \quad \text{where the solution is at :} \Delta f(\vec{x}^*) = 0$$

In the most cases there exist no close Form solution, choose a numerical iterative scheme.

Gradient Descent Method

The principal Idea of finding the minimum is to go into the negative direction of the Gradient:

$$\vec{x}^{(k+1)} = g(\vec{x}^{(k)}) = \vec{x}^{(k)} + t^{(k)} \Delta \vec{x}^{(k)} \quad \text{with } \Delta \vec{x}^{(k)} \in \mathbb{R}^d : \text{is the search direction in the k-th iteration}$$
$$t^{(k)} \in \mathbb{R} : \text{denotes the step length in the k-th iteration}$$

A good value for the Stepsize t is important. If t is oversized you jump back and forth, if t will be reduced after each iteration step there is a high probability to get

stuck before you reach the optimal point.
To find a stepsize its called line search, A very good algorithm to estimate t is the **Armijo-Goldstein Algorithm**.

There is a more general Way to get the Direction, where you include Norms:

$$\Delta\vec{x} = \arg\min_{u}\{\nabla f(\vec{x})^T\vec{u}; ||\vec{u}||_p = 1\}$$

L_1-norm leads to a coordinate descent method, the choosen direction is always a coordinate axis

L_2-norm is the normal steepest descent method

$L_\infty-$norm the direction will be always one of the 45° diagonals according to the coordinate system

L_P-norm norm means you project the gradient on the greatest principal component. A rewritten form:

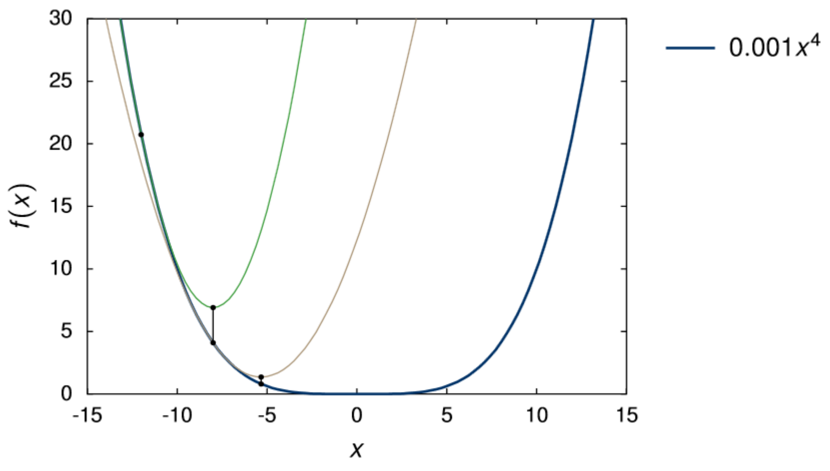
$$\Delta\vec{x} = -\mathbf{P}^{-1}\nabla f(\vec{x})$$

Newton's Method

The Newton's Method finds the nearest Extrema and needs no Stepsize t!
It's also a iterative scheme(shown in 1.9).

$$x^{(n+1)} = \vec{x}^{(n)} - \frac{f(\vec{x}^n)}{}$$

At the Point $x^{(k)}$ you have to approximate the function with a second order Taylor



approximation and compute the minimum of this approximation. The minimum is the new point $x^{(k+1)}$. You can compute the direction with:

$$\Delta\vec{x} = -\underbrace{(\nabla^2 f(\vec{x}))^{-1}}_{\text{inverse of the Hessian}} \nabla f(\vec{x})$$

Note: This is the same you see at the gradient descent method using the $L_{\mathbf{P}}$-norm !
The Newton method is an $\vec{x}$-dependent steepest descent method regarding the $L_{\mathbf{P}}$-norm, where $\mathbf{P} = \nabla^2 f(\vec{x})$ is the Hessian matrix.

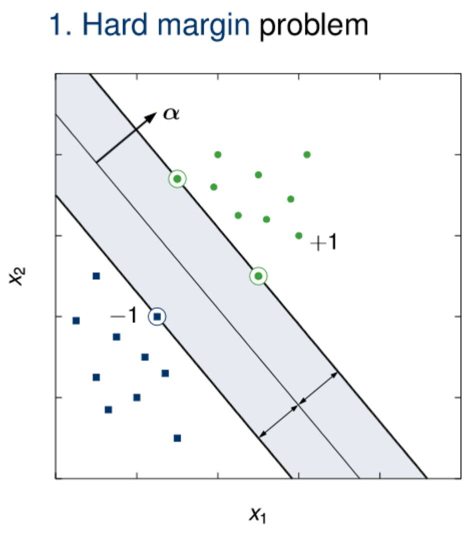
1.10 Support Vector Machines

SVM's are very good in terms of their generalization properties.
The main Idea is to look for an decision boundary that separates to classes but maximize the distance the nearest points of each class. The solution is **unique** and depends only on the features that a close to the decision boundary.
Original decision Boundary:

$$f(\vec{x}) = \vec{\alpha}^T\vec{x} + \alpha_0$$

Hard Margin Problem

The hard margin SVM 1.10 needs linearly separable classes. objective function for

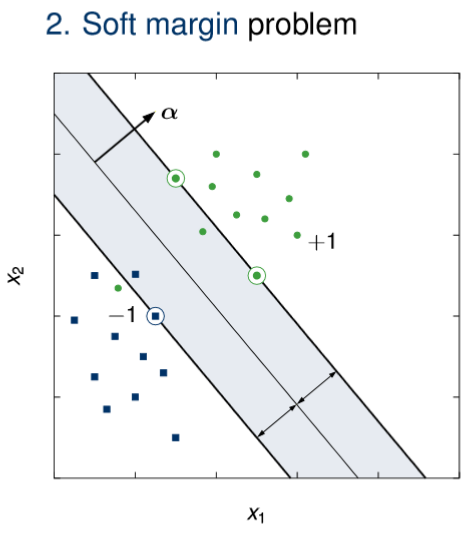


Hard Margin:

$$\begin{aligned} &\text{minimize } \frac{1}{2}||\vec{\alpha}||_2^2 \\ \text{subject to} \quad & y_i \cdot (\vec{\alpha}^T \vec{x}_i + \alpha_0) - 1 \geq 0 \qquad \forall i \end{aligned}$$

Soft Margin Problem

The Soft Margin SVM1.10 allows features to be on the wrong side of the decision boundary. So the classes are not linearly separable anymore. In this case the



optimization change to:

$$\begin{aligned} &\text{minimize } \frac{1}{2}||\vec{\alpha}||_2^2 + \mu \sum_i \xi_i \\ \text{subject to} \quad & -(y_i \cdot (\vec{\alpha}^T \vec{x}_i + \alpha_0) - 1 + \xi_i) \leq 0 \\ & -\xi_i \leq 0 \end{aligned}$$

ξ_i denotes misclassified features, so you minimizes like in the Hard Margin case and add as a constraint that the sum over the misclassified features $\mu \sum_i \xi_i$ should be also minimum.

parenthesis: KKT - conditions

The dual gap is Zero if the KKT- conditions are fullfilled:

1. Primal constrains:

$$\begin{aligned} f_i(\vec{x}) &\le 0, \quad i = 1,2,...,m \\ h_i(\vec{x}) &= 0, \quad i = 1,2,...,p \end{aligned}$$

2. Dual constraints: $\vec{\lambda} \ge 0$
3. Complementary slackness: $\lambda_i \cdot f_i(\vec{x}) = 0$. The most Important for SVMs
4. Gradient of Lagrangian is zero

if you derive L(the optimization Problem SVM-Hard Margin) with respect to $\vec{\alpha}, \alpha_0$

$$L(\vec{\alpha}, \alpha_0, \vec{\lambda}) = \frac{1}{2} \|\vec{\alpha}\|_2^2 - \sum_i \lambda_i \cdot (y_i \cdot (\vec{\alpha}\vec{x}_i + \alpha_0) - 1)$$

you get these two informations:

$$\vec{\alpha} = \sum_i \lambda_i y_i \vec{x}_i \qquad \text{and} \qquad -\sum_i \lambda_i y_i = 0$$

with this you get the dual problem:

$$\begin{aligned} \text{maximize} \quad & -\frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y_i y_j \cdot \langle \vec{x}_i, \vec{x}_j\rangle + \sum_i \lambda_i \\ \text{subject to:} \quad & \vec{\lambda} \ge 0 \\ & \textstyle\sum_i \lambda_i y_i = 0 \end{aligned}$$

The KKT-conditions have to be fulfilled for a zero duality gap!
Complementary slackness:

$$\forall i : \qquad \lambda_i(y_i \cdot (\vec{\alpha}^T\vec{x}_i + \alpha_0) - 1) = 0$$

if $\lambda_i > 0$ the second part $y_i(\vec{\alpha}^T\vec{x}_i + \alpha_0) - 1 = 0$ has to be Zero. And this is only true for the vectors on the Margin !
For all other feature Vectors the Lagrangian-multiplier is Zero!
new decision Boundary:

$$f(\vec{x}) = \underbrace{\Big(\sum_{i=1}^{m} \lambda_i y_i \vec{x}_i\Big)^T}_{\vec{\alpha}} \vec{x} + \alpha_0 \longrightarrow f(\vec{x}) = \sum_{i=1}^{m} \lambda_i y_i \langle \vec{x}_i, \vec{x}\rangle + \alpha_0$$

Feature Transform in the case of SVM

Both hard- and soft- margin SVM's generates linear decision Boundaries.
Use the same transformation Φ liike in the case of Logistic Regression with the advantage that in the decision boundary above we kicked out alpha, you can transform $\vec{x}$ with Φ you got a quadratic decision boundary (you have a linear one in a higher dimensional space where the SVM solves the problem!) **Difference between Roseblatt's Perceptron and SVM**

- SVM is a convex optimization problem (lagrange-Method -> unique solution)
- SVM maximizes the distance to a hyperplane
- Rosenblatt fits plane and uses only missclassified features (discrete problem)
- Perceptron is not able to classify patterns with classes that are not linear separable, but there is soft-margin SVM

1.11 Kernels

For Kernel functions the following holds:

$$k(\vec{x}, \vec{x}') = \langle \Phi(\vec{x}), \Phi(\vec{x}') \rangle$$

for any feature mapping Φ.

- Whenever you have a inner product you can use the Kernel-Trick $\langle \Phi(\vec{x}), \Phi(\vec{x}') \rangle$ to get non-linear decision Boundaries!
- you can pre-Compute a so called Kernel Matrix for a given set of feature vectors

$$K_{ij} = \langle \Phi(\vec{x}_i), \Phi(\vec{x}_j) \rangle \qquad \text{where ij denots the entrie in the Matrix}$$

- the Kernel-matrix is positive semi definite

an Example is the polynomial Kernel $k(\vec{x}, \vec{x}') = (\langle \Phi(\vec{x}), \Phi(\vec{x}') \rangle + 1)^d$. If you set $d = 2$ then you will get a quadratic decision Boundary.

You **don't have to know the feature transformation** Φ. You just have to know what the inner product is of the transformed features. And a combination of the computation of the inner product and the transformation **is** the Kernel. The Kernel function does both!

Kernel PCA

The usage is, that if the classes are not linearly separable, so projection on the first principal component will mix up the classes. With the kernel you have a transformation before the projection in a higher dimensioal space where the classes are linearly separable! you can rewrite the PCA such that you can use the Kernel trick !
The Key equation of the Kernel PCA is(K is the Kernel Matrix):

$$K^2 \cdot \vec{\alpha}_i = m \cdot \lambda_i K$$

with the eigenvalue-/eigenvector-Problem:

$$K\vec{\alpha}_i = m\lambda_i\vec{\alpha}_i$$

For PCA you need feature vectors with zero mean. You get the components of the centred Kernel-Matrix $\tilde{\mathbf{K}}$ based on the old entries without knowing Φ:

$$\tilde{\mathbf{K}}_{i,j} = \mathbf{K}_{i,j} - \frac{1}{m}\sum_{k=1}^{m}\mathbf{K}_{i,k} - \frac{1}{m}\sum_{k=1}^{m}\mathbf{K}_{k,j} + \frac{1}{m^2}\sum_{k,l=1}^{m}\mathbf{K}_{k,l}$$

Kernel PCA (and thus also classical PCA) can be computed by solving an eigenvalue problem for a $m \times m$-matrix where m is the number of training samples. The Kernel matrix depends not on the dimension of the feature Vectors !

1.12 Laplacian SVM

This is an extension to Kernel SVMs and is state of the art in research.
The Laplacian SMV works on partially labeled training sets:
Training data $\mathcal{S} = \mathcal{L} \cup \mathcal{U}$

$$\max_{\lambda \in \mathcal{R}^l} \sum_{i=1}^{l} \lambda_i - \frac{1}{2}\vec{\lambda}^T Q \vec{\lambda}$$

subject to $0 \leq \lambda_i \leq 1, i = 1..., l$

$$\sum_{i=1}^{l} \lambda_i y_i = 0$$

where Q is some really fancy shit !

labeled data: $\mathcal{L} = \{(\vec{x}_i, y_i), \quad i = 1, ..., l\}$

unlabeled data: $\mathcal{U} = \{\vec{x}_i, \quad i = \underbrace{l + 1, ..., m\}}_{u}$

1.13 Parameter Estimation Methods

- Maximum likelihood estimation
  - All observations are assumed to be mutually statistically independent
  - the observations are kept fixed
  - the log-likelihood function is optimized regarding the parameters

$$\hat{\vec{\theta}} = \arg\max_{\vec{\theta}} p(\vec{x}_1, \vec{x}_2, ..., \vec{x}_m; \vec{\theta}) = \arg\max_{\vec{\theta}} \prod_{i=1}^{m} p(\vec{x}_i; \vec{\theta}) = \arg\max_{\vec{\theta}} \sum_{i=1}^{m} \log p(\vec{x}_i; \vec{\theta})$$

- Maximum a-posteriori estimation
  - the probability density function of the parameters $p(\theta)$ to be estimated is known

$$\hat{\vec{\theta}} = \arg\max_{\vec{\theta}} p(\vec{\theta}|\vec{x}) = \arg\max_{\vec{\theta}} \log p(\vec{\theta}) + \log p(\vec{x}|\vec{\theta})$$

1.14 Expectation Maximization Algorithm

The expectation maximization method are developed to deal with

- high dimensional parameter spaces
- latent, hidden, incomplete data

Gaussian Mixture Models

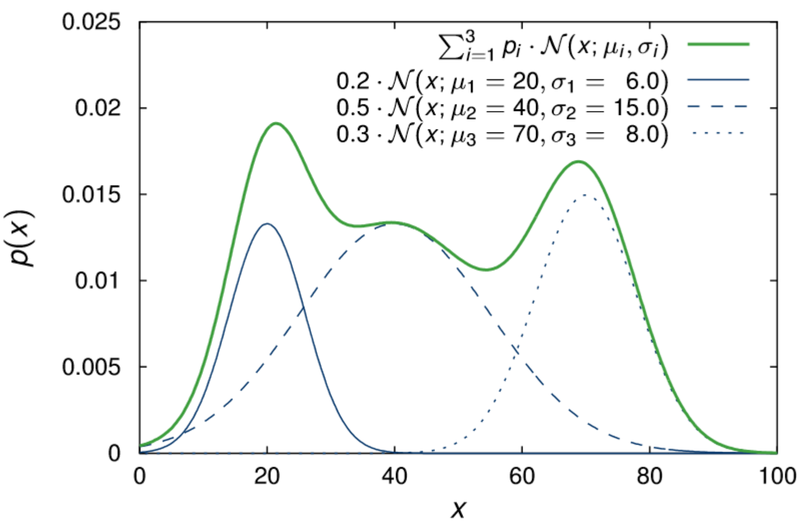
Given m feature vectors in an d-dimensional space, find a set of K multivariate Gaussian distribution that best represent the observations. GMMs are an example of classification by **unsupervised learning**:

- it is not known which feature vector are generated by which of the K Gaussian
- the desire output is, for each feature vector, an estimate of the probability that it is generated by distribution k

The parameter Estimation for GMM's is a nonlinear Problem!
You want:

$$p(k|\vec{x}) = \frac{p(k) \cdot \overbrace{p(\vec{x}|k)}^{\mathcal{N}(\vec{x};\vec{\mu}_k,\Sigma_k)}}{p(\vec{x})} \quad \text{with: } p(\vec{x}) = \sum_{k=1}^{K} p(k) \cdot p(\vec{x}|k)$$

Use the log-likelihood Estimation to estimate the parameters for each Gaussian:



μ_k the mean for the k-th Gaussian

Σ_k the covariance matrices of size $d \times d$ for the k-th Gaussian

p_k fraction of all features in component k, how many samples where actually generated by component k

Log-Likelihood:

$$L = \sum_{i=1}^{m} \log p(\vec{x}_i) \quad \text{with: } p(\vec{x}) = \sum_{k=1}^{K} p(k) \cdot p(\vec{x}|k)$$

Derive L you get can calculate the parameters like in the case of one Gaussian but in this case **weighted** by $p(k|\vec{x})$:

$$\hat{\vec{\mu}}_k = \frac{\sum_i p(\vec{x}_i) \cdot p(k|\vec{x}_i)}{\sum_i p(k|\vec{x}_i)}; \quad \hat{\Sigma}_k = \frac{\sum_i (\vec{x}_i - \hat{\vec{\mu}}_k)(\vec{x}_i - \hat{\vec{\mu}}_k)^T \cdot p(k|\vec{x}_i)}{\sum_i p(k|\vec{x}_i)}; \quad \hat{p_k} = \frac{1}{m} \sum_i p(k|\vec{x}_i)$$

This leads to an iterative scheme, initialize the parameters with Zero and start the EM(1.14):

$$\begin{tikzcd} & \hat{\mu}_k, \hat{\Sigma}_k, \hat{p}_k \arrow[dd, bend left=60, "\text{Expectation}"] & \\ & & \\ & p(k|\vec{x}) \arrow[uu, bend left=60, "\text{Maximization}"] & \end{tikzcd}$$

General Expectation Maximization Algorithm

Expectation Maximization is used when trying to model hidden information. We formulate the missing information principle as follows:

$$\text{observable information} = \text{complete information} - \text{hidden information}$$

The GMMs are such a case where you don't know which component generates which samples.
Write this in a more mathematical way:

- observable random variable X
- hidden random variable Y parameter set: $\vec{\theta}$

Key equation of the joint probability density of the events x and y is:

$$\underbrace{-\log p(x;\vec{\theta})}_{\text{observable-}} = \underbrace{-\log p(x,y;\vec{\theta})}_{\substack{\text{complete-}\\ \text{-information}}} - \underbrace{(-\log p(y|x;\vec{\theta}))}_{\text{hidden-}}$$
$$\log p(x;\hat{\vec{\theta}}^{(i+1)}) = \log p(x,y;\hat{\vec{\theta}}^{(i+1)}) - \log p(y;\hat{\vec{\theta}}^{(i+1)})$$

The key to find a solution for this is marginalization!

$$\int p(\vec{x},y;\vec{\theta})dy = p(x;\vec{\theta}) \tag{1.6}$$

Multiply the key equation above with $p(y|x;\hat{\vec{\theta}}^{(i)})$ and integrate over the hidden event y. You end up with the following formula (for the EM-part we design a iterative scheme for $\vec{\theta}$):

$$\underbrace{\log p(x;\hat{\vec{\theta}}^{(i+1)})}_{\text{log-likelihood}} = \underbrace{Q(\vec{\theta}^{(i)};\hat{\vec{\theta}}^{(i+1)})}_{\text{Kullback-Leiber statistics}} + H(\vec{\theta}^{(i)};\hat{\vec{\theta}}^{(i+1)}) \tag{1.7}$$

The H-Term is the Entropy,the Entropy will never decrease $\rightarrow H(\vec{\theta};\vec{\theta}') \geq H(\vec{\theta};\vec{\theta})$.
You **ignore** H in the EM- algorithm.
Instead of maximizing the log-likelihood function you maximize the Kullback-Leiber statistics Q iteratively while ignoring H !

1. Initialize $\hat{\theta}^{(0)}$
2. Set $i := -1$. Repeat
   a) Set $i := i + 1$

b) Expectation step:

$$\text{compute: } Q(\hat{\theta}^{(i+1)}; \hat{\theta}^{(i)})$$

c) Maximization step:

$$Q(\hat{\theta}^{(i+1)}; \hat{\theta}^{(i)}) \rightarrow \text{maximize w.r.t} \quad \hat{\theta}$$

until $\hat{\theta}^{(i+1)} = \hat{\theta}^{(i)}$.

3. Output: estimate $\hat{\theta} := \hat{\theta}^{(i)}$

some Facts:

- the maximum of Q is usually computed using zero crossings of the gradient
- The iteration scheme is numerically robust
- The iteration has constant memory requirements
- the EM-algorithm converge very slow
- it finds only local extremum $\rightarrow$ initialization have a heavy impact on the result
- EM-Algorithm is usually used in many constrained optimization Problems

1.15 Independent Component Analysis

Cocktail-Party Problem

Imagine two microphones in a room at different location which record the time signals $x_1(t), x_2(t)$. Each recorded signal is a weighted sum of two speakers $s_1(t), s_2(t)$:

$$x_1(t) = a_{11}s_1(t) + a_{12}s_2(t)$$
$$x_2(t) = a_{21}s_1(t) + a_{22}s_2(t)$$

where the parameters a_{ij} depends on the distance of the microphones to the speakers. Without knowing a_{ij} it's not easy to solve these linear equations.
The core idea to solve this problem is to use information about the statistical properties of the signals $s_i(t)$ to estimate a_{ij}. It is sufficient to assume that the $s_i(t)$ are **statistically independent** at each time point t. Rewrite the time series into

n linear mixture observations $x_1, ..., x_n$. Each mixture x_i as well as each component s_j are random variables:

$$x_i = \sum_{j=1}^{m} a_{ij} s_j, \qquad i = 1, ..., n \tag{1.8}$$

in matrix notation:

$$\vec{x} = A\vec{s} \tag{1.9}$$

the original Signals 1.1(c) (which you dont know!). the Signals we have and you want

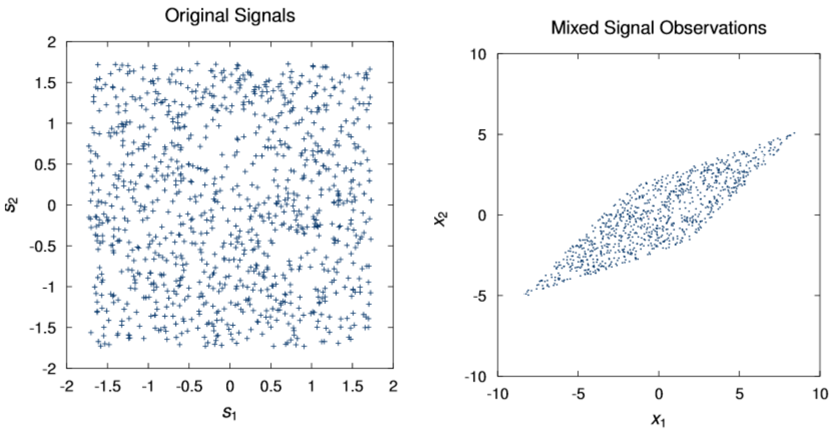


Abbildung 1.1: Orignial- and Observed-Signal

to transform to the Original 1.1(d).

Pre Step 1 - Centering

Step 2 needs de-meaned Data !

Pre Step 2 - Whitening Transform

First step is decorrelation!
For the decorrelation youse the approach you know from the LDA.
$\Longrightarrow$ enforce a Identity Matrix as the covariance Matrix (as done in the LDA)! Compute $\frac{1}{n}\sum_i \vec{x}_i\vec{x}_i^T$ to get the covariance Matrix.
Hints:
The Whitening transform is not unique(you can see in 1.2, its rotated)

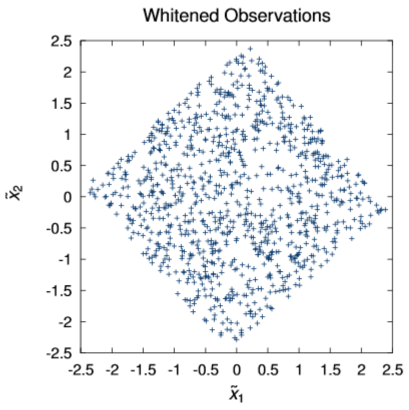


Abbildung 1.2: Signal after whitening transform

Basic ICA

If you knew A, you could compute its inverse A^{-1} to obtain the independent components:

$$\vec{s} = A^{-1}\vec{x}$$

(Hint: A is whitening transformed : $D^{-\frac{1}{2}}U^T A$ the sign of A is still not unique!)
That would lead you to s_i being a linear combination of x_i with a weight vector $\vec{w}$, which is a row of $\mathbf{A}$:

$$s_i \overset{?}{=} y = \vec{w}^T\vec{x}$$

so y equals one of the independent components if $\vec{w}$ is one of row of $\mathbf{A}^{-1}$.
You can rewrite this:

$$y = \vec{w}^T\vec{x} = \vec{w}^T \underbrace{A\vec{s}}_{\vec{x}} = \vec{z}^T\vec{s}; \qquad \text{where } \vec{z} = A^T\vec{w}$$

The Key principle of ICA is:

Maximizing the non-Gaussianity of $\vec{w}^T\vec{x}$ results in the independent components!

Because the Gaussian is the distribution with the highest entropy, therefore it is the least informative pdf!(One Component is allowed to be normal distributed)

ICA Estimation Algorithm

With all this knowledge you can build an algorithm:

1. Apply centering transform

2. Apply whitening transform
3. Set $i := 1$ and repeat
   - Take a random vector $\vec{w}_i$
   - Maximize non-Gaussianity of $\vec{w}_i^T$ subject to $\|\vec{w}_i\|_2 = 1$ and $\vec{w}_j^T \vec{w}_i$ (where $j < i$)
   - Increase i

   until $i > n$ (where n is the number of independent components)
4. Use $W = \left(\vec{w}_1^T, \vec{w}_2^T, \dots, \vec{w}_n^T\right)$ to compute $\vec{s}$
5. Output: independent components $\vec{s}$

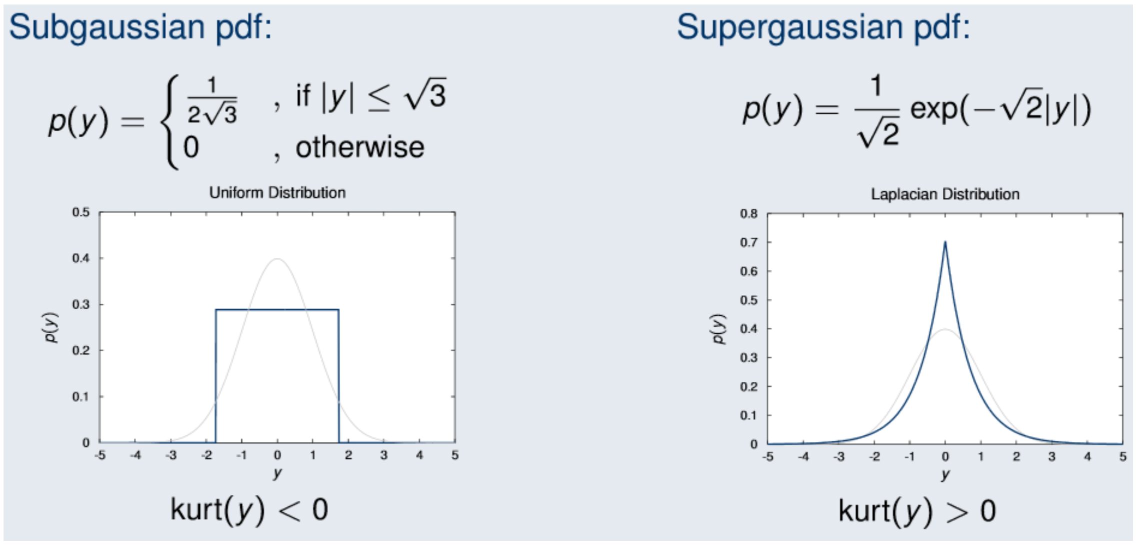
You need a measurement for non-Gaussianity to do this.

Measures of Non-Gaussianity

Kurtosis

The kurtosis of a Gaussian is Zero

$$\text{kurt}(y) = 0$$



Subgaussian pdf:

$$p(y) = \begin{cases} \frac{1}{2\sqrt{3}} & , \text{ if } |y| \leq \sqrt{3} \\ 0 & , \text{ otherwise} \end{cases}$$

$$\text{kurt}(y) < 0$$

Supergaussian pdf:

$$p(y) = \frac{1}{\sqrt{2}} \exp(-\sqrt{2}|y|)$$

$$\text{kurt}(y) > 0$$

Negentropy

Negentropy measures the difference to a Gaussian random variable.

$$J(y) = H(y_{\text{Gauss}}) - H(y) \qquad \text{where } \Sigma_{y_{\text{Gauss}}} = \Sigma_y$$

In theory, negentropy is an optimal statistica estimator of non-Gaussianity. But Computing the negentropy from a measures set of samples requires the estimation of the pdf.
Instead, there are approximations for negentropy.

Mutual Information

measure statistical dependency between two random variables directly using *mutual information*

minimize the Mutual Information to compute the direction of the highest non-Gaussianity.
Negentropy and Mutual Information are equivalent under certain conditions!

1.16 Model Assessment

No Free Lunch - Theorem

The no free lunch theorem states that

the sum over all cost functions given two algorithms is equal !

i.e. an algorithm might perform well regarding a specific cost function, but no algorithm will perform well according to all possible cost functions.(1.3)
The consequences of the Theorem are:

- there is no general best classifier !
- if an algorithm achieves superior results on some problems, it must pay with inferiority on other problems

Off-training set error

- used to compare the classification performance of algorithms
- specifies the error on samples that are **not** elements of the training set

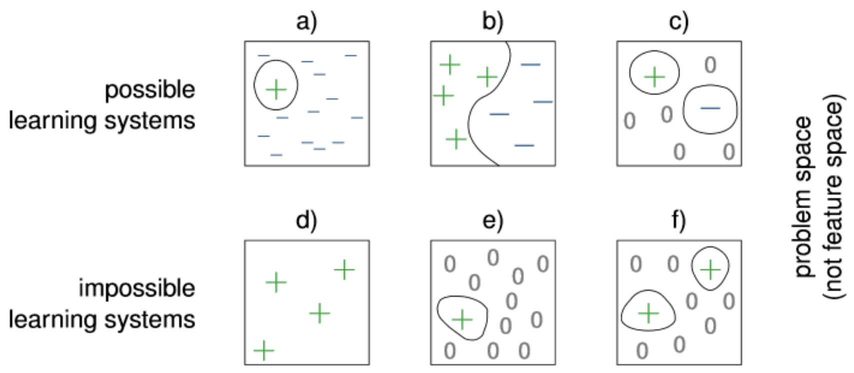

a)
b)
c)
d)
e)
f)
possible learning systems
impossible learning systems
problem space (not feature space)

Abbildung 1.3: Each square represents all possible classification problems.+/− indicates better/worse generalization than the average

- trade-off between training and test set size! (bigger training Data are good for the training but then you have a smaller test Dataset)

Bias and Variance

The bias-variance relation allows you to assess the quality of a lerning algorithm in terms of the alignment to the classification problem

Bias The bias measures the accuracy of the match. In other words Bias is a systematic error:

high bias means poor match

Variance The variance measures the precision of specificity for the match. In other words variance is how things jump back and forth:

high variance implies a weak match

You can reduce both, Bias and Variance, if you use as much prior Information as possible!

- in regression the bias-variance relation is additive in $(\text{bias})^2$ and variance
- For classification the relation is multiplicative and non-linear (low variance is important for accurate classification)

Estimating and Comparing Classifier

Its about computation of Bias and Variance for learning Algorithms that we applied to a new problem with unknown distribution

$$\text{Resampling methods try to estimate the bias and variance}$$
$$\text{of } \phi_n(x) \text{ using subsamples from a Dataset}$$
$$\text{e.g. } \phi(x) = \frac{1}{n}\sum_i \vec{x}_i = \bar{\vec{x}} \rightarrow \text{how robust is the estimator of the mean?}$$

Jackknife

Jackknife Computes the bias and the Variance on the Dataset $X_{(i)} = (x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$ is the set without the i-th element. And does that n times.
Jackknife is a specialized Bootstrap.

Bootstrap

$$\text{A bootstrap data set is created by randomly selecting } n \text{ points}$$
$$\text{from the sample set with replacement}$$

This selection is independently repeated B times.

Cross-Validation

In cross-validation, the training samples are split into two disjoint parts:

- the first set is the training set used for the traditional training
- the second set is the test set used to estimate the classification error
- in a second step, both sets are swapped
- by that, the classification error can be estimated on the complete data set
- yet, training and test set are always disjoint

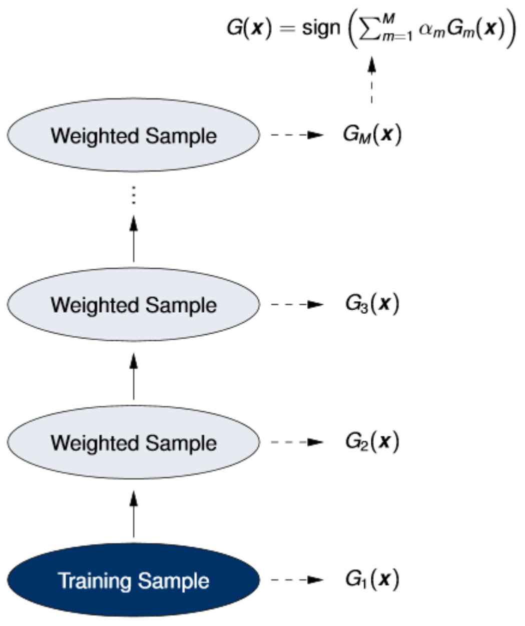
An m-fold cross-validation splits the data into m disjoint sets of size $\frac{n}{m}$, where one set is used as test set and the other $m-1$ sets are used for training. Cross-validation degrades to the jackknife for $m = n$.

1.17 AdaBoost

The core Idea is to combine many weak classifiers.

A weak classifier is one whose error rate is only slightly better than random guessing.

After the first classification, the missclassified features get a higher priority ("Weighted Sampleïn 1.17): The Algorithm-schema is:



1. Initialize weights $w_i = \frac{1}{N}$
2. Set $m := 1$, repeat
   - Fit classifier $G_m(\vec{x})$ to training data using $\vec{w}$
   - Compute classification error
   - Compute classifier weights
   - Compute new sample weights
   - Increase m

   until $m = M$
3. Output: $G(x) = \text{sgn}\left(\sum_{i=1}^{M} \alpha_m G_m(\vec{x})\right)$