Artificial Intelligence

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Thorsten Schmidt

Abteilung für Mathematische Stochastik

www.stochastik.uni-freiburg.de
thorsten.schmidt@stochastik.uni-freiburg.de
SS 2017

Motivation
Overview
A hierarchy
Machine learning examples
Introduction
Basics
Supervised learning
Unsupervised learning
Semi-supervised learning
Reinforcement learning
Machine Learning Basics
Generalized linear models
Regularization
LASSO
Logistic regression
Maximum-likelihood
Support vector machines
The statistical classification problem
Support Vector Classifier
The kernel trick
A short excursion to convex optimization
The kernel trick
Reproducing kernel Hilbert spaces
Kernel examples
MNIST - data: an example
Support vector machines (continued)
Multiple classes
Virtual Support Vector Machines
Adaptive basis function models
CART
Hierarchical mixtures of experts

Some preliminaries
Structured Probabilistic Models
Stochastic Gradient Descent

Deep Learning
Gradient-based learning
Rectified linear units

Backpropagation

Regularization

Dynamic Approximate Programming
Introduction

Markov decision problems

Approximate dynamic programming
The basic idea
Q-Learning

Infinite-Horizon Problems
Value iteration
Policy iteration

Approximating the value function for fixed π
Temporal Differences

Bayesian Optimization

Literature (incomplete, but growing):

- I. Goodfellow, Y. Bengio und A. Courville (2016). **Deep Learning**. http://www.deeplearningbook.org. MIT Press
- D. Barber (2012). **Bayesian Reasoning and Machine Learning**. Cambridge University Press
- R. S. Sutton und A. G. Barto (1998). **Reinforcement Learning : An Introduction**. MIT Press
- G. James u. a. (2014). **An Introduction to Statistical Learning: With Applications in R**. Springer Publishing Company, Incorporated. ISBN: 1461471370, 9781461471370
- T. Hastie, R. Tibshirani und J. Friedman (2009). **The Elements of Statistical Learning**. Springer Series in Statistics. Springer New York Inc. URL: https://statweb.stanford.edu/~tibs/ElemStatLearn/
- K. P. Murphy (2012). **Machine Learning: A Probabilistic Perspective**. MIT Press
- CRAN Task View: Machine Learning, available at https://cran.r-project.org/web/views/MachineLearning.html
- UCI ML Repository: http://archive.ics.uci.edu/ml/ (371 datasets)
- Warren B Powell (2011). **Approximate Dynamic Programming: Solving the curses of dimensionality**. Bd. 703. John Wiley & Sons
- A nice resourse is https://github.com/aikorea/awesome-rl

Motivation

- Artificial Intelligence includes machine learning as one exciting special case
- Machine Learning is nowadays used at many places (Google, Amazon, etc.)
- It is a great job opportunity ! It needs maths and probability !
- Many applications are surprisingly successful (speech / face recognition) and currently people are seeking further applications
- Here we want to learn about the foundations, discuss implications and what can be done by ML and what not
- The lecture is an open forum for discussions and will be developed during the semester. Slides will be available online, one day ahead. The exercises will include computational projects, in particular towards the end.

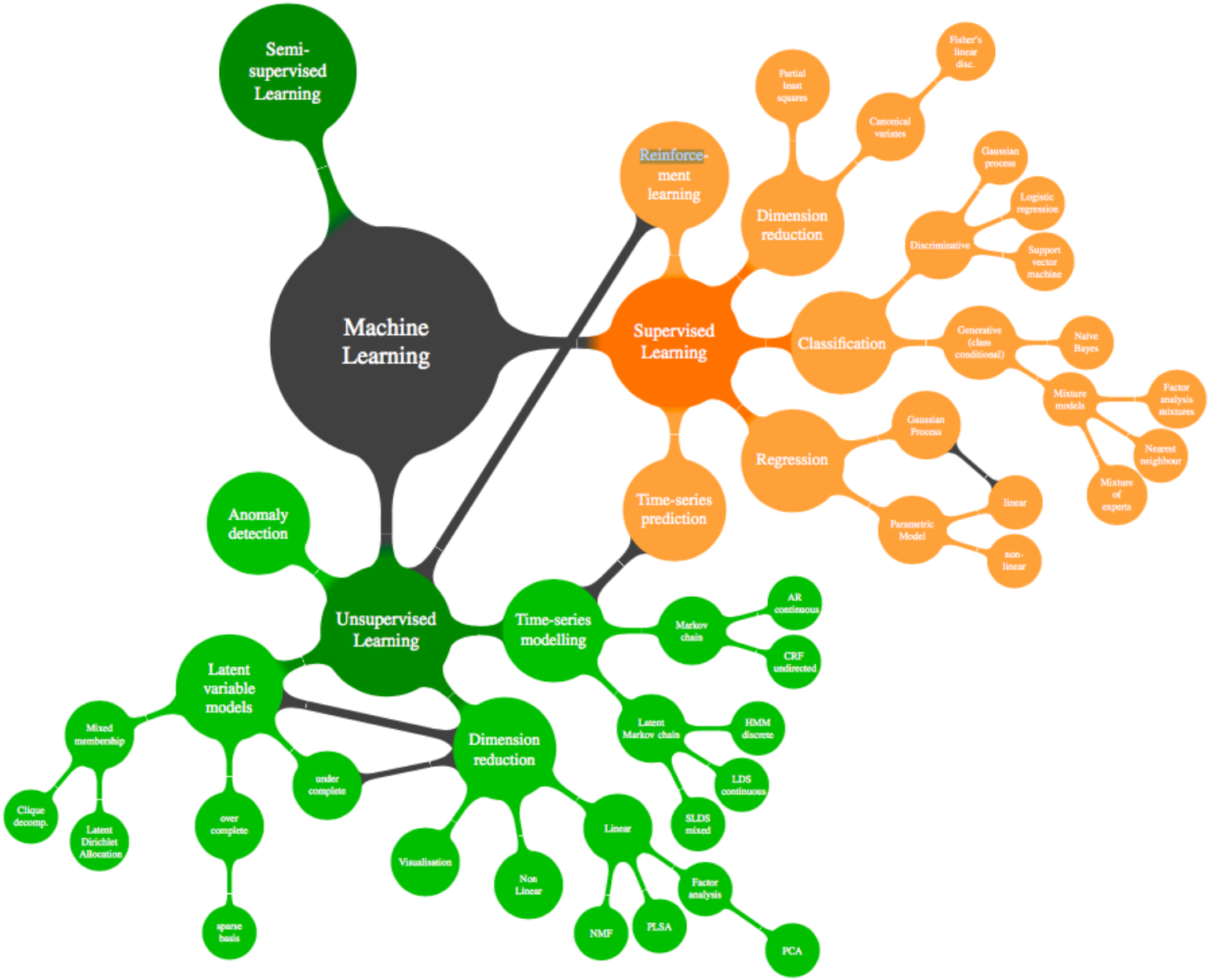
Overview[1]

- Artificial intelligence is the field where computers solve problems.
- It is easy for a computer to solve tasks which can be described formally (Chess, Tic-Tac-Toe). The challenge is to solve a tasks which are hard to describe formally (but are easy for humans: walk, drive a car, speak, recognize people ...)
- The solution is to allow computers to learn from experience and to understand the world by a hierarchy of concepts, each concept defined in terms of its relation to simpler concepts.
- A fixed knowledge-base would be somehow limiting such that we are interested in such attempts where the systems acquire their own knowledge, which we call **Machine Learning**.

[1] This introduction follows closely Goodfellow et.al. (2016).

- First examples of machine learning are **logistic regression** or **naive Bayes** $\rightarrow$ standard statistical procedures (Cesarean delivery / Recognition of Spam, more examples to follow)
- Problems become simpler with a nice representation. Of course it would be nice if the system itself could find such a representation, which we call **representation learning**.
- An example is the so-called **auto-encoder**. This is a combination of an encoder and a decoder. The encoder converts the input to a certain representation and the decoder converts it back again, such that the result has nice properties.
- Speech for example might be influenced by many factors of variation (age, sex, origin, ...) and it needs nearly human understanding to disentangle the variation from the content we are interested in.
- **Deep Learning** solves this problem by introducing hierarchical representations.

- This leads to the following hiearchy:
- AI $\rightarrow$ machine learning $\rightarrow$ representation learning $\rightarrow$ deep learning.



Source: Barber (2012).

Examples of Machine Learning

Some of the most prominent examples:

- LeCun et.al.[2] recognition of handwritten digits. The MNIST Database[3] provides 60.000 samples for testing algorithms.
- The Viola & Jones face recognition,[4]. This path-breaking work proposed a procedure to combine existing tools with machine-learning algorithms. One key is the use of approx. 5000 learning pictures to train the routine. We will revisit this procedure shortly.

[2] Y. LeCun u. a. (1998). „Gradient-based learning applied to document recognition“. In: **Proceedings of the IEEE** 86.11, S. 2278–2324.

[3] `http://yann.lecun.com/exdb/mnist/`

[4] P. Viola und M. Jones (2001). „Robust Real-time Object Detection“. In: **International Journal of Computer Vision**. Bd. 4. 34–47.

- Speech recognition has long been a difficult problem for computers (first works date to the 50's) and only recently been solved with high computer power. It may seem surprising, that mathematical tools are at the core of these solutions. Let us quote Hinton et.al.[5]

  > *Most current speech recognition systems use hidden Markov models (HMMs) to deal with the temporal variability of speech and Gaussian mixture models (GMMs) to deter- mine how well each state of each HMM fits a frame or a short window of frames of coefficients that repre- sents the acoustic input. (...)*
  > *Deep neural networks (DNNs) that have many hidden layers and are trained using new methods have been shown to outperform GMMs on a vari- ety of speech recognition benchmarks, sometimes by a large margin*

  So, one of our tasks will be to develop a little bit of mathematical tools which we will need later. Most notably, some of the mathematical parts can be replaced by deep learning, which will be of high interest to us.

[5] Geoffrey Hinton u. a. (2012). „Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: **IEEE Signal Processing Magazine** 29.6, S. 82–97.

Types of machine learning:

- Supervised learning: The data consists of datapoints and associated labels, i.e. we start from the dataset

$$(x_i, y_i)_{i \in I}.$$

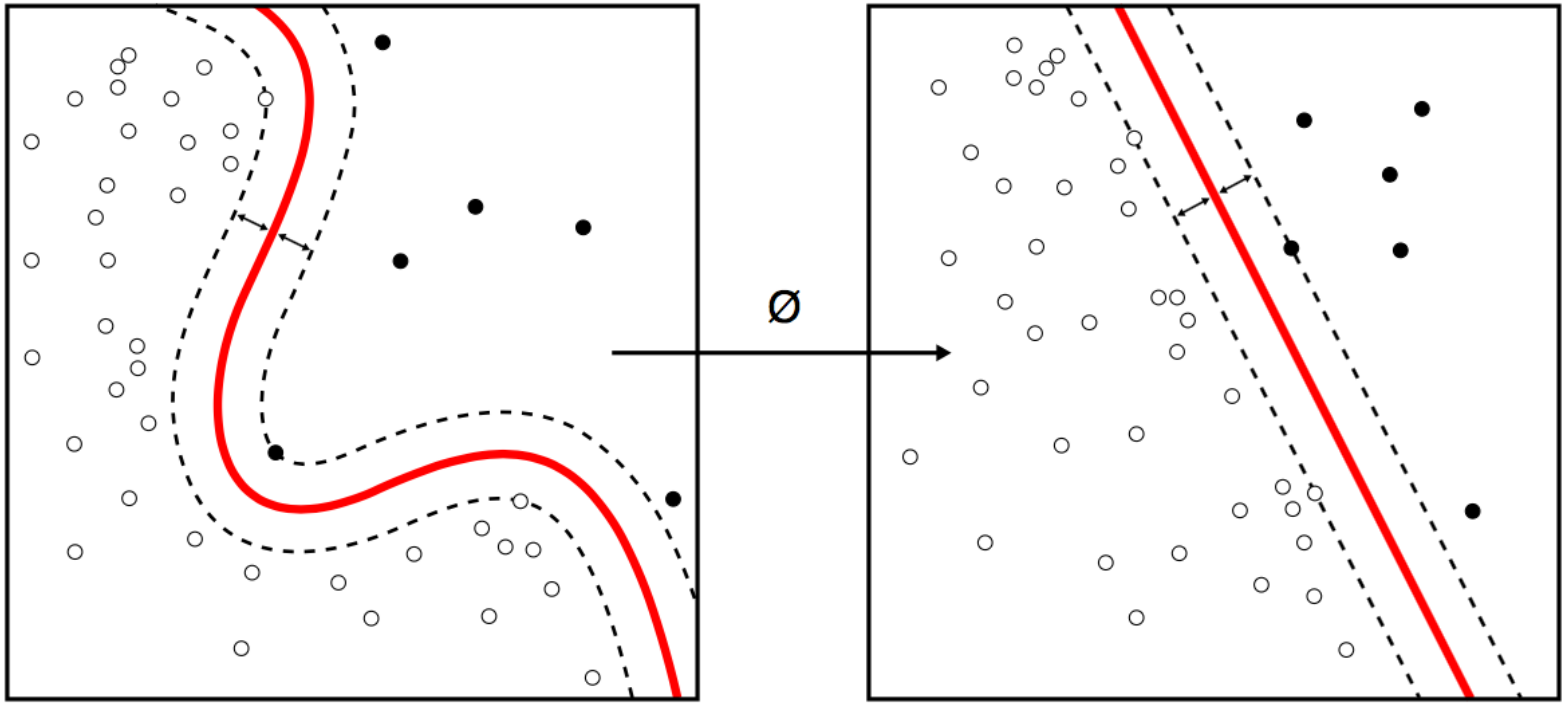
  We give some examples:

- **Image recognition** (face recognition) where the images come with labels, i.e. cats / dogs or the person to which the image is associated to.
- **Spam filter** the training set contains emails together with the label spam / no spam.
- **Speech recognition** here sample speech files comes together with the content of the sentences. It is clear, that some sort of grammar understanding helps to break up the sentences into smaller pieces, i.e. words.

- Unsupervised learning: In this case the data just comes at it is, i.e.

$$(x_i)_{i\in I}$$

and one goal would be to identify a certain structure from the data itself. In this sense the machine learning algorithm shall itself find a characteristics which divides the data into suitable subsets.

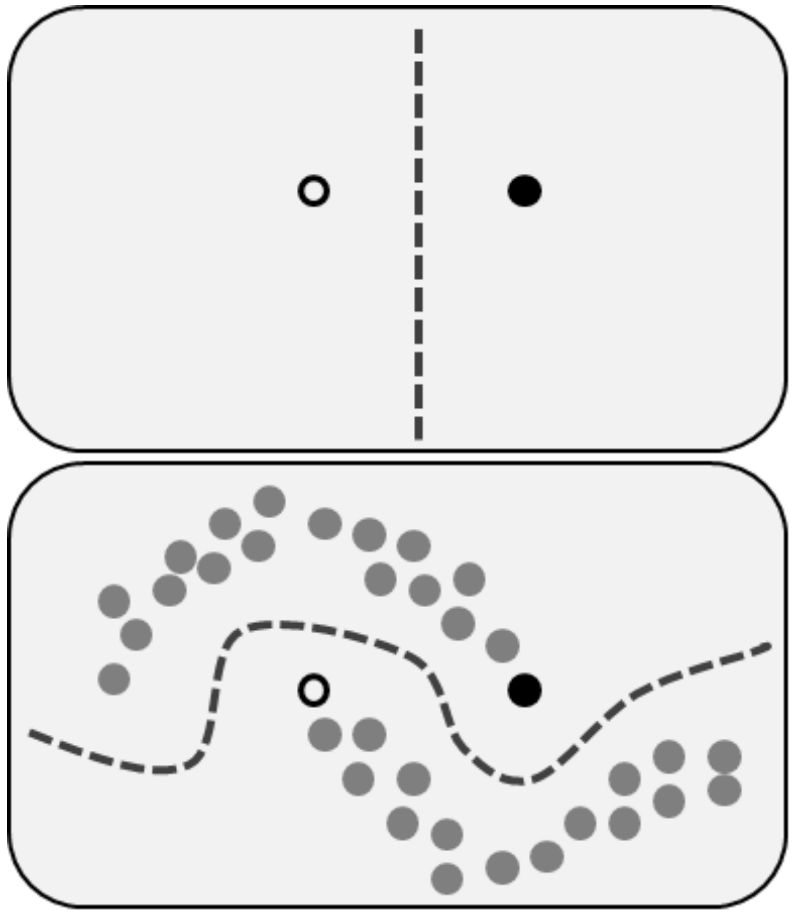


Picture by: Alisneaky, svg version by User:Zirguezi - Own work,
CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=47868867

Some examples

- Analysis of genomic data
- Density estimation
- Clustering
- Principal component analysis

- Semi-supervised learning: only a few data are labelled and many are unlabelled.
- Labelling typically is quite expensive and the additional use of unlabelled data might improve the performance. However, some assumptions need to be made, such that this procedure works through.



Picture by: Techerin - Own work,
CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=19514958

Reinforcement learning

is quite different from the above examples.

- First: time matters, the problem depends on time ! Observations accumulate over time.
- There is no supervisor but a reward signal measuring the quality of the decision.
- The approach utilizes a probabilistic framework: Markov decision processes.
- Examples are: drive a car, optimally manage a portfolio ...

- In a nutshell, we proceed iteratively through time.
- At time t, we *observe* X_t, get a reward $U(X_t)$ and are able to make a decision D_t which influences the state at time $t+1$, X_{t+1}.
- A *policy* describes the decision given the state. It can be stochastic or deterministic.
- While initially the environment is unknown, the system gathers information through its interactions with the environment and improves its policy.

A quite related area is Statistical Learning. This new area of statistics is quite related to machine learning and we will study a number of relevant problems.

Definition

A computer program learns from experience E with respect to tasks T, if its performance P improves with experience E.

This quite vague definition allows us to develop some intuition about the situation.

- **Experience** is given by an increasing sequence of observations, for example $X_1, X_2, \dots, X_t$ could represent the information at time t. This is typically decoded in a **filtration**: a filtration is an increasing sequence of sub-σ-fields $(\mathcal{F}_t)_{t\in\mathcal{T}}$.
- The performance is often measures in terms of an **utility function**. For example the utility at time t could be given by $U(X_t)$ with an function U. U could of course depend on more variables. One could also look for the accumulated utility

$$\sum_{t=1}^{T} U(X_t).$$

One very simple learning algorithm is linear regression, a classical statistical concept. Here it arises as an example of **supervised learning**.

Example (Linear Regression)

Suppose we oberseve pairs $(x_i, y_i)_{i=1,\dots,n}$ and want to predict y on basis of x. **Linear** regression requires

$$\hat{y}(x) = \beta x$$

with some weight $\beta \in \mathbb{R}$. We specify a loss function[6]

$$\text{RSS}(\beta) := \sum_{i=1}^{n} (y_i - \hat{y}(x_i))^2$$

and minimize over β.

One could choose $-$MSE as utility function. So how does the system **learn**?

[6] Given by the Residual Sum of Squares here.

The system learns by maximizing the utility, i.e. minimizing the MSE for each n. And additional data will lead to a better prediciton. We will later see that this is in a certain sense indeed optimal.

We use the first-oder condition to derive the solution letting $\boldsymbol{x} = (x_1, \dots, x_n)$ and similar for $\boldsymbol{y}$,

$$\begin{aligned} & 0 = \partial_\beta (\boldsymbol{y} - \beta \boldsymbol{x})^2 = \partial_\beta (\boldsymbol{y}^2 - 2\boldsymbol{y}^\top \beta \boldsymbol{x} + \beta^2 \boldsymbol{x}^\top \boldsymbol{x}) \\ \Leftrightarrow \beta \quad & 0 = -2\boldsymbol{x}^\top \boldsymbol{y} + 2\beta \boldsymbol{x}^\top \boldsymbol{x} \end{aligned}$$

such that we obtain

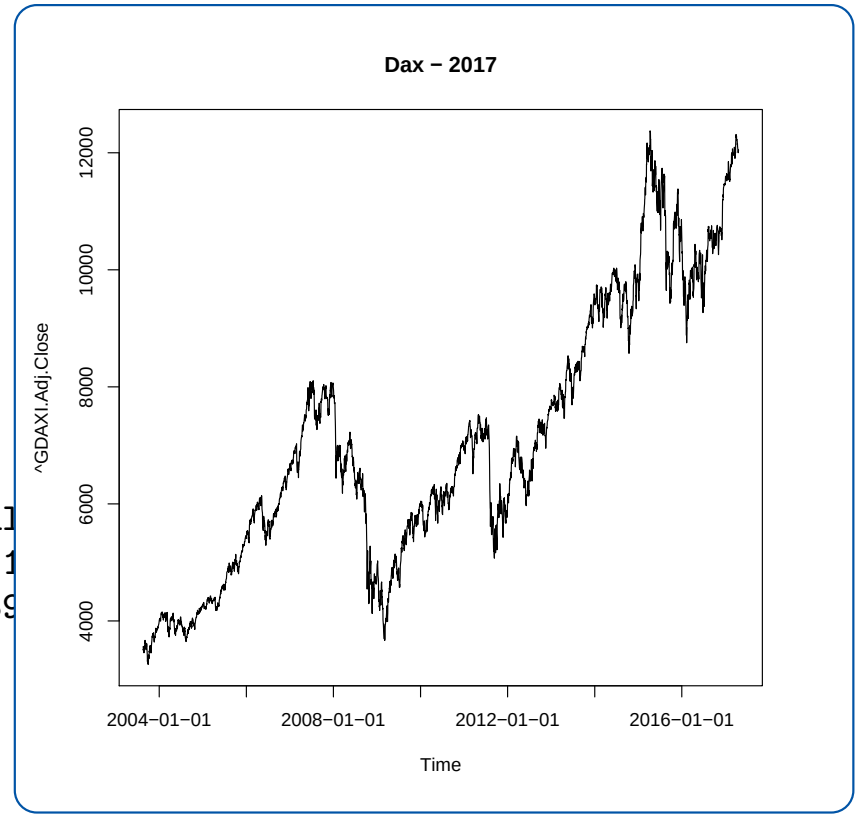
$$\hat{\beta} = (\boldsymbol{x}^\top \boldsymbol{x})^{-1} \boldsymbol{x}^\top \boldsymbol{y}.$$

Note that typically one considers affine functions of x without mentioning, i.e. one looks at functions $y = \alpha + \beta x$. This can simply be achieved with the linear approach by augmenting $\boldsymbol{x}$ by an additional entry 1.

- Of course many generalizations are possible:
- To higher dimensions: consider data vectors $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, $i = 1, \ldots, n$,
- To nonlinear functions: include $x_i^1, \ldots, x_i^p$ into the covariates
- and many more.

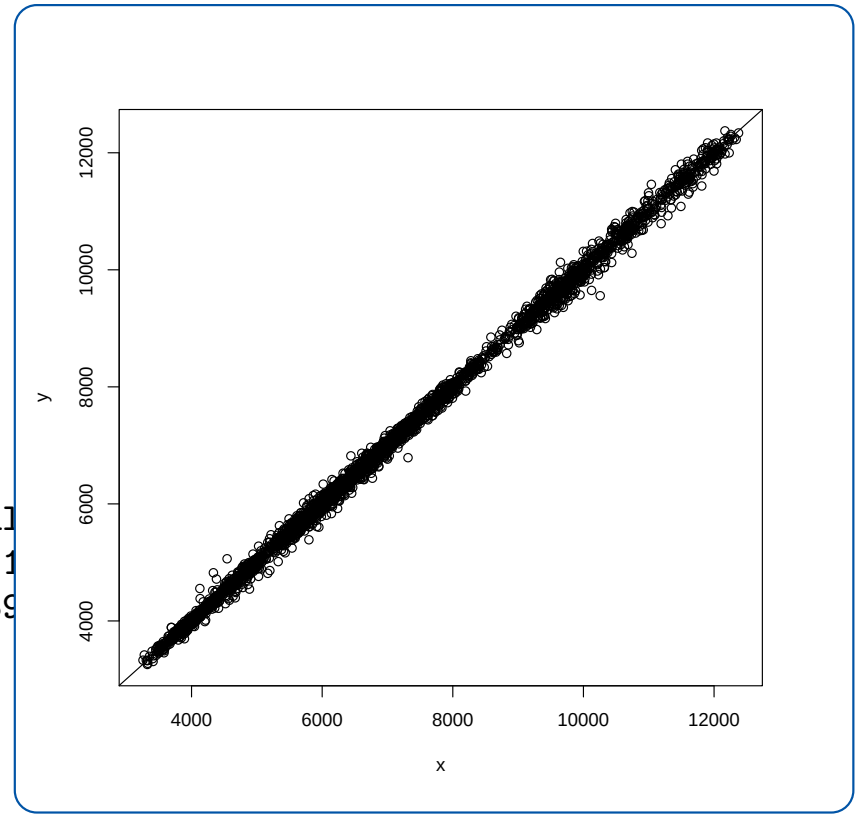
Let us consider a linear regression in R.

```
library (fImport)
stockdata <- yahooSeries(c("^GDAXI"),nDaysBack=5000)[,c("^GDAXI.Adj.Close")]
plot(stockdata)

N=length(stockdata)
# prepare for linear regression
x = stockdata[1:N-1]
y = stockdata[2:N]

plot(x,y)

Regression = lm (y~x)
summary (Regression)
abline (Regression)

# Coefficients:
#             Estimate Std. Error  t val
# (Intercept) 6.0077820  5.1533966    1
# x           0.9994964  0.0006941 1439
```



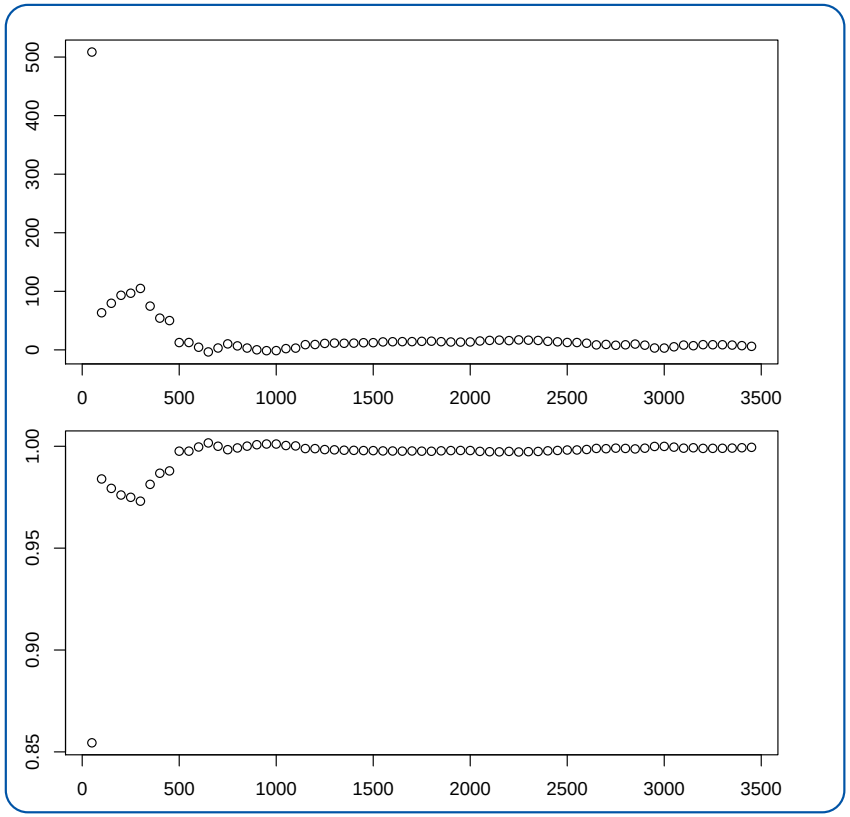
Let us consider a linear regression in R.

```
library (fImport)
stockdata <- yahooSeries(c("^GDAXI"),nDaysBack=5000)[,c("^GDAXI.Adj.Close")]
plot(stockdata)

N=length(stockdata)
# prepare for linear regression
x = stockdata[1:N-1]
y = stockdata[2:N]

plot(x,y)

Regression = lm (y~x)
summary (Regression)
abline (Regression)

# Coefficients:
#             Estimate Std. Error  t val
# (Intercept) 6.0077820  5.1533966    1
# x           0.9994964  0.0006941 1439
```



Now we consider the learning effect:

```
n=round(N/50)+1
ab = array(rep(0,2*n),dim = c(n,2))
j = 50; i=1
while (j < N-1) {
  Regression = lm(y[1:j]~x[1:j])
  ab [i,] = Regression$coefficients
  i=i+1; j=j+50  }
i=i-1

par ( mfrow = c(2,1),mar=c(2,2.1,1,1))
plot((1:i)*50,ab[1:i,1])
plot((1:i)*50,ab[1:i,2])
```



Could we improve this ? Suggestions ?

What is the difference to Statistics ?

In a statistical approach we start with a **parametric model**:

$$Y_i = \alpha + \beta x_i + \varepsilon_i, \quad i = 1, \dots, n$$

and assume that $\varepsilon_1, \dots, \varepsilon_n$ have a certain structure (for example, i.i.d. and $\mathcal{N}(0, \sigma^2)$). The one can derive (see, e.g. Czado & Schmidt (2011)) **optimal estimators** for α and β. One can also relax the assumptions and gets weaker results.

So what ? What are the advantages of the statistical approach ?

One particular outcome is that we are able to provide **confidence intervals**, **predictive intervals** and **test** hypothesises.

Generalized Linear Models

We already saw that transforming the input variables suitable might be helpful. This is the idea of a generalized linear model (GLM), see Casella & Berger (2002).

Definition

A GLM consists of three components:

1. Response variables (random) $Y_1, \dots, Y_n$,
2. a systematic component of the form $\alpha + \boldsymbol{\beta}^\top \boldsymbol{x}_i$, $i = 1, \dots, n$,
3. a link function g satisfying

$$\mathbb{E}[Y_i] = g(\alpha + \boldsymbol{\beta} \boldsymbol{x}_i), \quad i = 1, \dots, n.$$

Regularization of multiple linear regression

- One problem in practice is parsimony of a linear regression: suppose you have many covariates and you want to include only those which are relevant.
- It would be possible to iteratively throw out those parameters which are not significant. This procedure, however is not optimal. Many others have been proposed.
- We concentrate on **continuous** subset selection methods: it is better to introduce a penalty for including two many parameters, which we call regularization. This is moreover a standard procedure for ill-posed problems. We will consider a famous example: the **LASSO** introduced in R. Tibshirani (1996). „Regression Shrinkage and Selection via the Lasso“. In: **Journal of the Royal Statistical Society. Series B (Methodological)** 58.1, S. 267–288.

LASSO

- The **least absolute shrinkage and selection operator** minimizes the following function

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \left\{ \frac{1}{n} \parallel \boldsymbol{Y} - \boldsymbol{x}\boldsymbol{\beta} \parallel_2^2 + \lambda \parallel \boldsymbol{\beta} \parallel_1 \right\}.$$

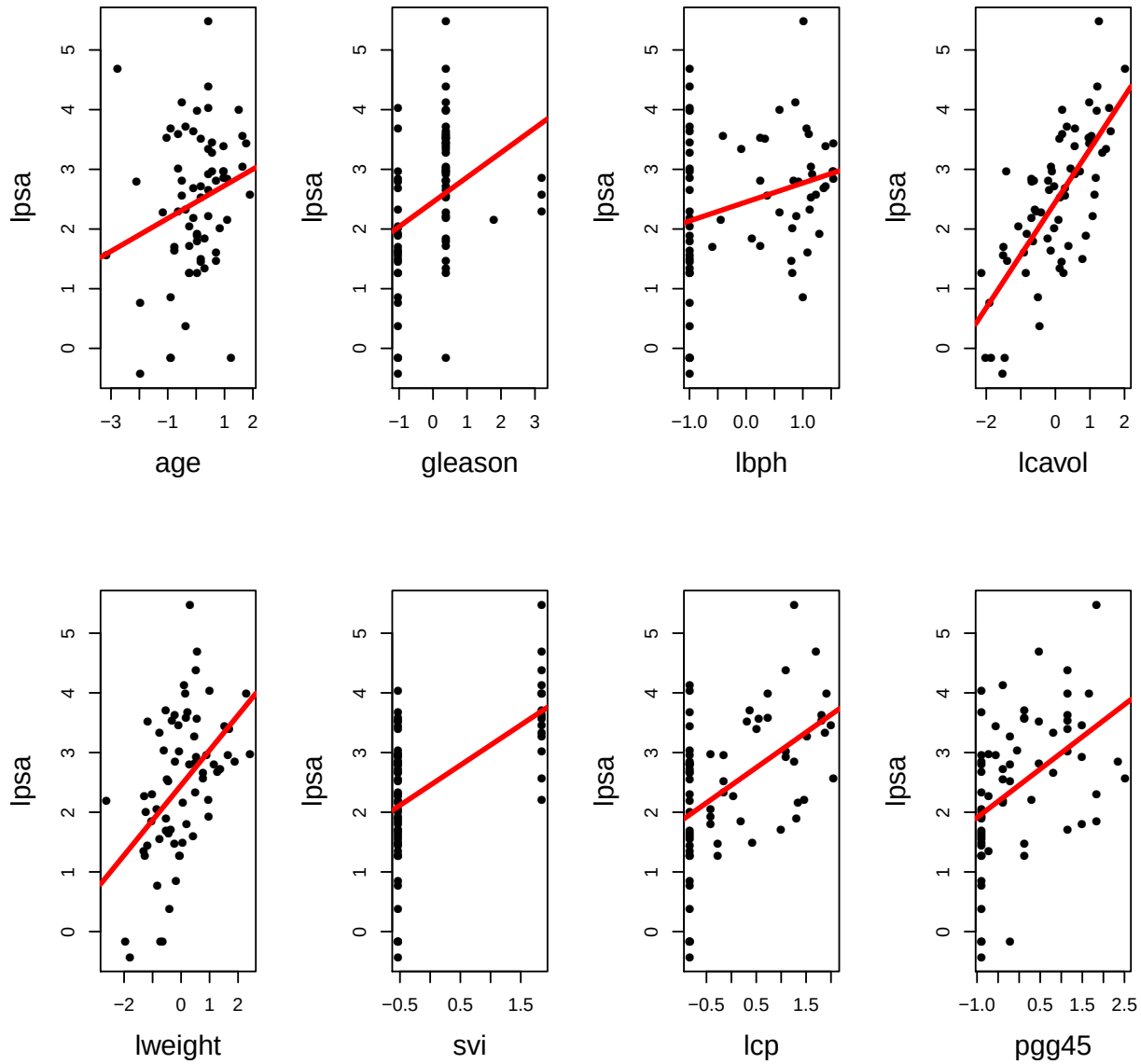
  The parameter λ has to be chosen and allows to vary the level of regularization. Clearly this model prefers to set non-significant parameters to zero.

- Let us illustrate the lasso with an example taken from Chris Franck, `http://www.lisa.stat.vt.edu/?q=node/5969`. The data stems from Stamey et.al.[7].

- The data describes clinical measures from 97 men about to undergo radical prostatectomy. It is of interest to estimate the relation between the clinical measures and the prostate specific antigen (measures are: lcavol - log (cancer volume), lweight - log(prostate weight volume), age, lbph - log (benign prostatic hyperplasia), svi - seminal vesicle invasion, lcp - log(capsular penetration), Gleason (score), ppg45 - percent Gleason scores 4 or 5, $Y =$lpsa - log(prostate specific antigen))

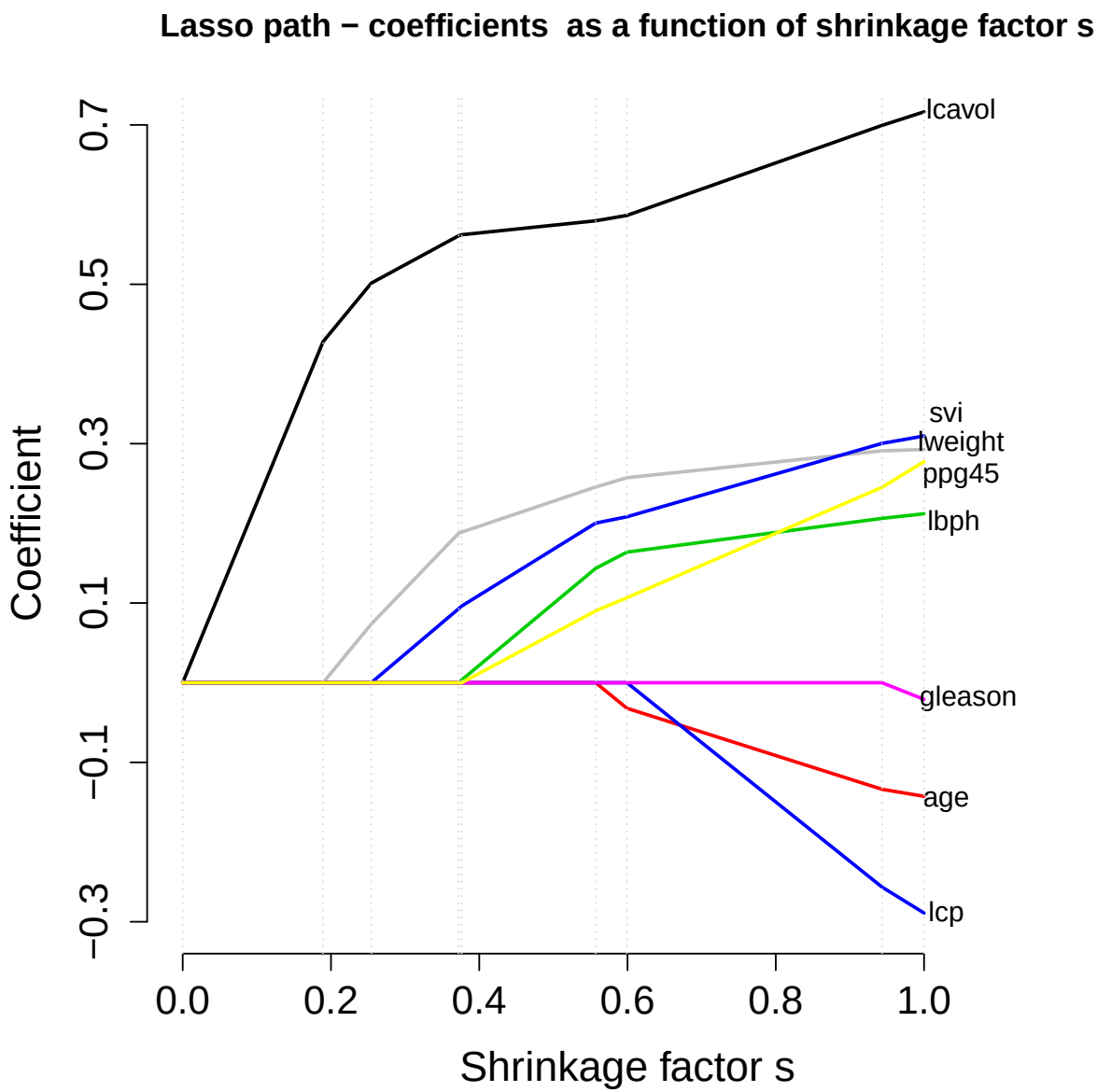
[7] T. A. Stamey u. a. (1989). „Prostate specific antigen in the diagnosis and treatment of adenocarcinoma of the prostate. II. Radical prostatectomy treated patients.“ In: **The Journal of urology** 141.5, S. 1076–1083.

We start by examining bi-variate regressions.

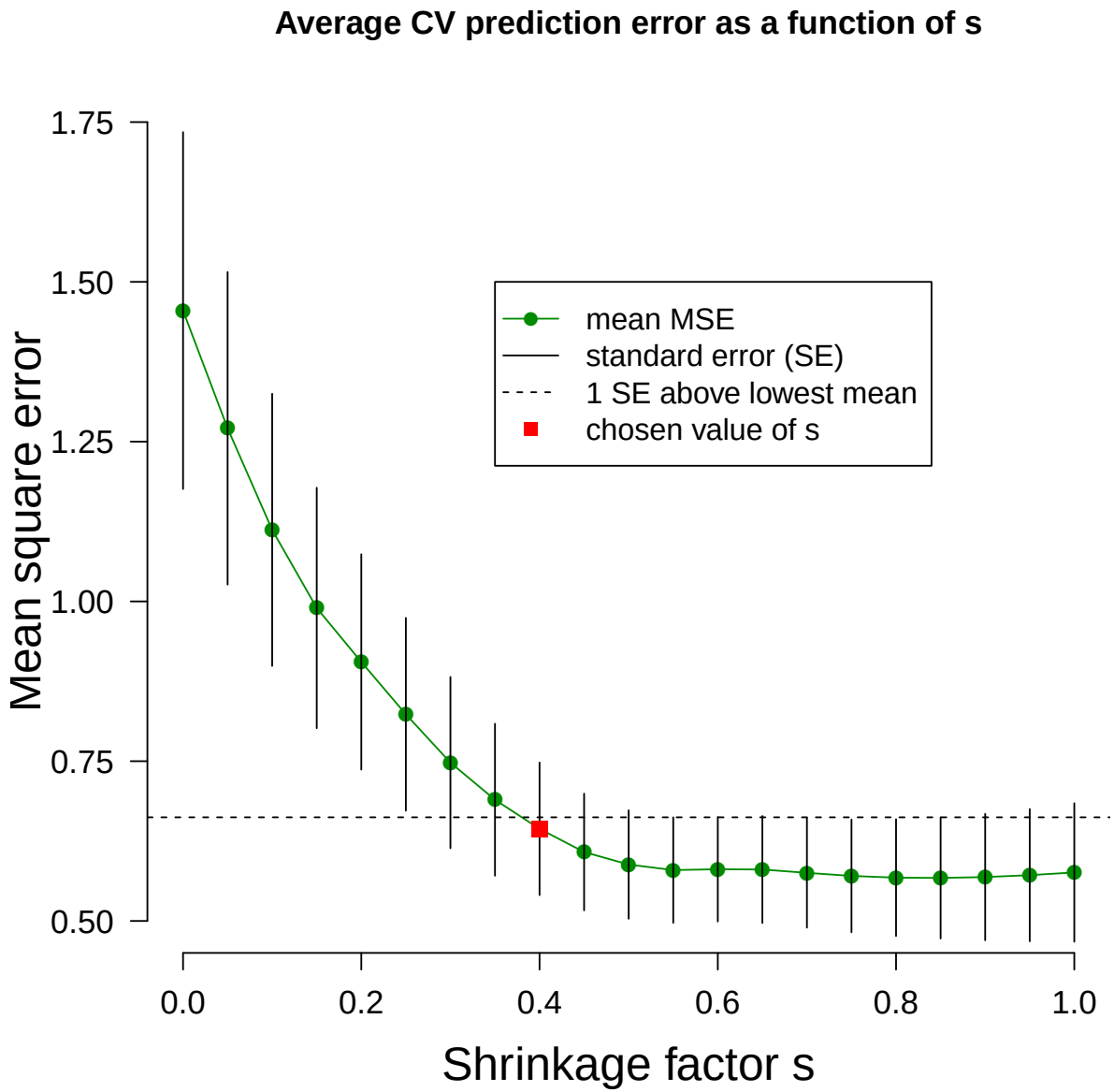


- It is obvious that some variables have fewer impact and some others seem to be more important. The question is how to effectively select those.
- We illustrate how cross-validation may be used in this case. This means we separate the data into a training set and a validation set. The tuning parameter λ is chosen based on the training set and validated on the validation set.
- We use a 10-fold cross validation, ie. the set is split into 10 pieces. Iteratively, each piece is chosen as the validation set while the remaining 9 sets are used to estimate the model.

This is the so-called lasso path. The shrinkage factor is antiproportional to λ.



This is the cross-validation result. A rule of thumb is to select that value of s that is within 1 standard error of the lowest value.



- We see that the optimal choice of λ is far from trivial. Alternative approaches are at hand, compare the recent results by Johannes Lederer and coauthors, J. Lederer und C. Müller (2014). „Don't Fall for Tuning Parameters: Tuning-Free Variable Selection in High Dimensions With the TREX". In: **ArXiv e-prints**. eprint: 1404.0541 (stat.ME).

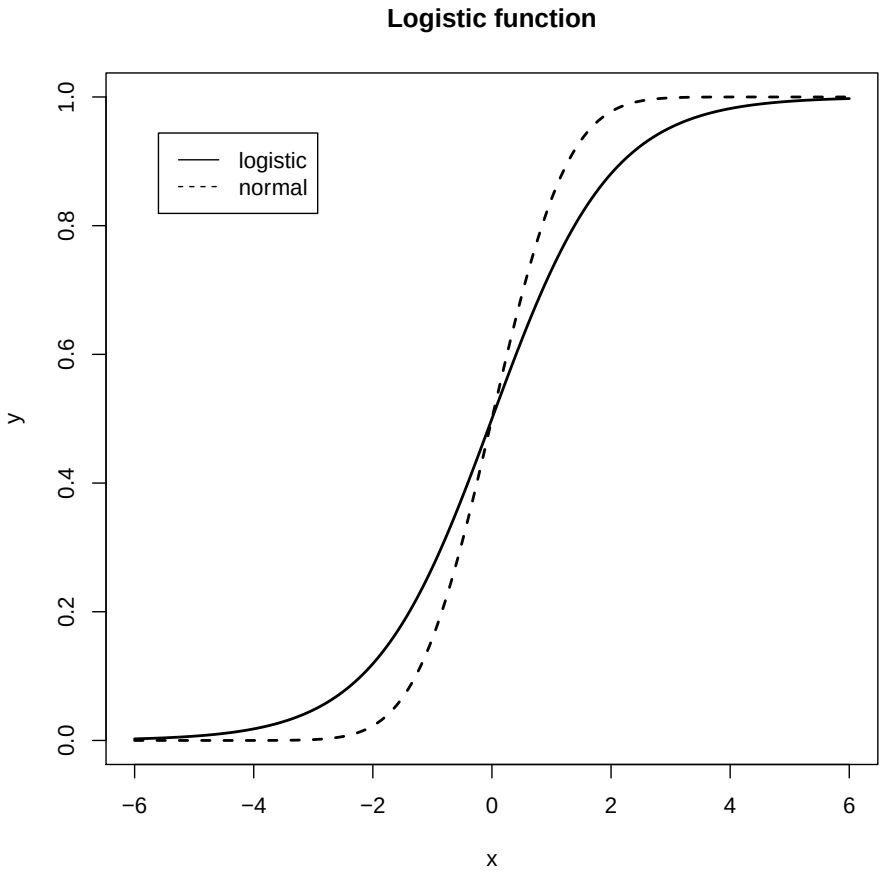
Logistic regression

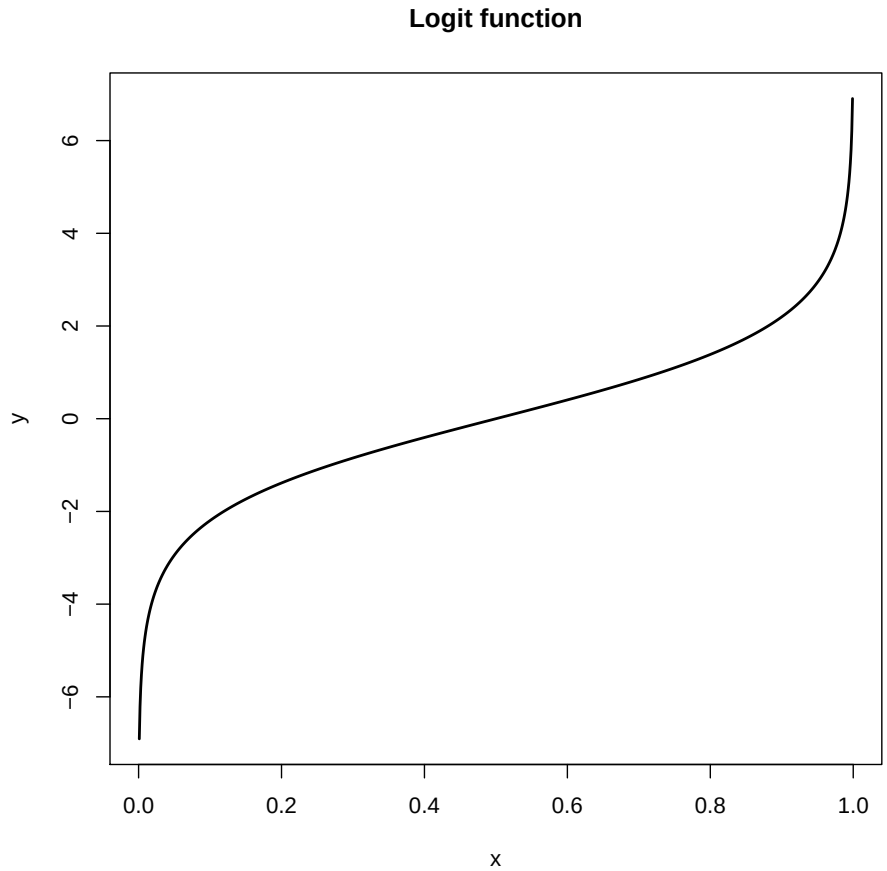
One important regression approach for classification is logistic regression. In this case, the response is always binary. One therefore needs to transform the whole real line to $[0,1]$ and two approaches are common: first, via the logistic function

$$\sigma(x) = \frac{e^x}{1+e^x},$$

(leading to $g = \sigma^{-1}$, the so-called **logit** function) by a cumulative distribution function (when this is Φ - standard normal - this approach is called **probit** model).

The most common estimation method used is **maximum-likelihood**. We take a small detour towards this exciting statistical concept going back to Sir Ronald Fisher.


Logistic function
logistic
normal
y
x
0.0
0.2
0.4
0.6
0.8
1.0
−6
−4
−2
0
2
4
6


Logit function
y
x
−6
−4
−2
0
2
4
6
0.0
0.2
0.4
0.6
0.8
1.0

Maximum-likelihood

- A **statistical model** is given by a family of probability measures $(P_\theta)_{\theta\in\Theta}$ on a common measurable space $(\Omega,\mathscr{F})$. It is typically called **parametric**, if Θ is of finite dimension.
- The **likelihood**-function for the observation E is given by

$$L(\theta) = P_\theta(E)$$

If $P_\theta(E) = 0$ for all $\theta \in \Theta$ one proceeds via the density: assume $P_\theta \ll P^*$ for all $\theta \in \Theta$ and denote the densities by $f_\theta := {}^{dP_\theta}/_{dP^*}$. Then, for the observation x,

$$L(\theta) = f_\theta(x).$$

- This looks complicated, but is in most cases quite simple: consider i.i.d. random variables $X_1,\dots,X_n$ with common density f_θ. Then P^* is clearly the Lebesgue-measure. Due to the i.i.d.-property,

$$L(\theta) = \prod_{i=1}^{n} f_\theta(x_i).$$

Definition

Any maximizer $\hat{\theta}$ of the likelihood-function is called maximum-likelihood estimator for the model $(P_\theta)_{\theta\in\Theta}$.

In the above example, we need to maximize $\prod_{i=1}^n f_\theta(x_i)$, which is typically infeasible. One therefore considers the log-likelihood function

$$\ell(\theta) := \ln L(\theta)$$

which is often much easier to maximize. Typically one can apply first-order conditions or needs to solve numerically.

Example (ML for the normal distribution)

Consider $X_i \sim \mathcal{N}(\mu, 1)$. Then the density is

$$f_\theta(x) = \frac{1}{\sqrt{2\pi}} \exp\Big(-\frac{1}{2}(x-\mu)^2\Big).$$

We obtain the log-likelihood function

$$l(\theta) = \text{const.} - \frac{1}{2}\sum_{i=1}^{n}(x_i-\mu)^2.$$

The first derivative is

$$\partial_\mu l(\theta) = \sum_{i=1}^{n} x_i - n\mu \stackrel{!}{=} 0$$

and we obtain the maximum-likelihood estimator (second derivative is < 0)

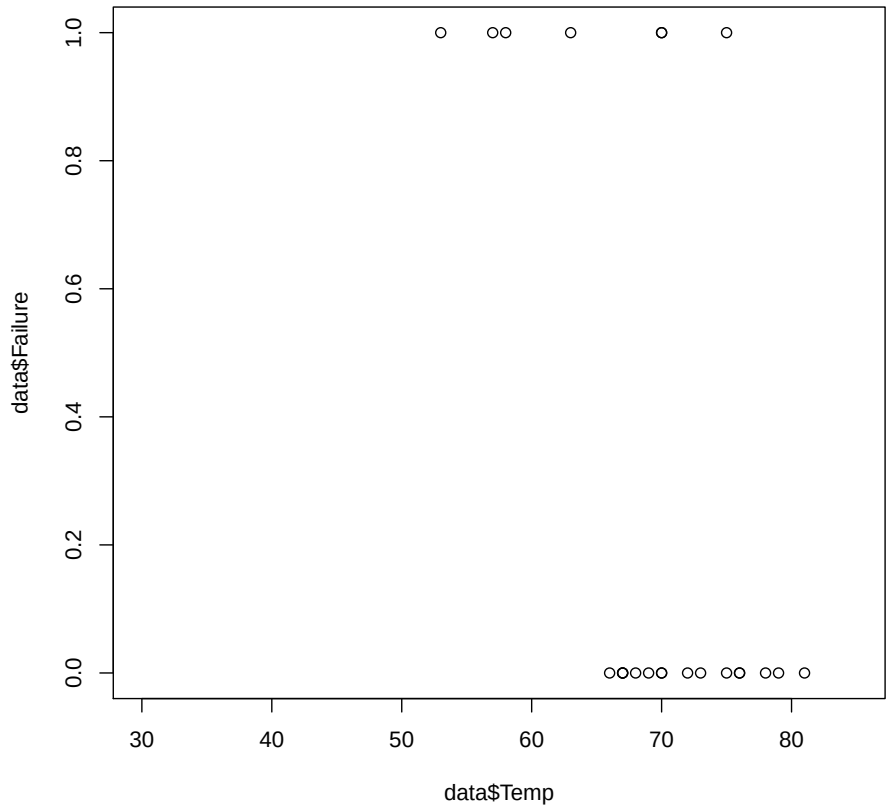
$$\hat{\mu} = \bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}.$$

Exercise: compute the ML estimator for σ ! Read Czado & Schmidt (2011) on ML-estimation and further estimation procedures.

Back to logistic regression. We look at the by now infamous Challenger O-ring data set (taken from Caslla & Berger (2002))

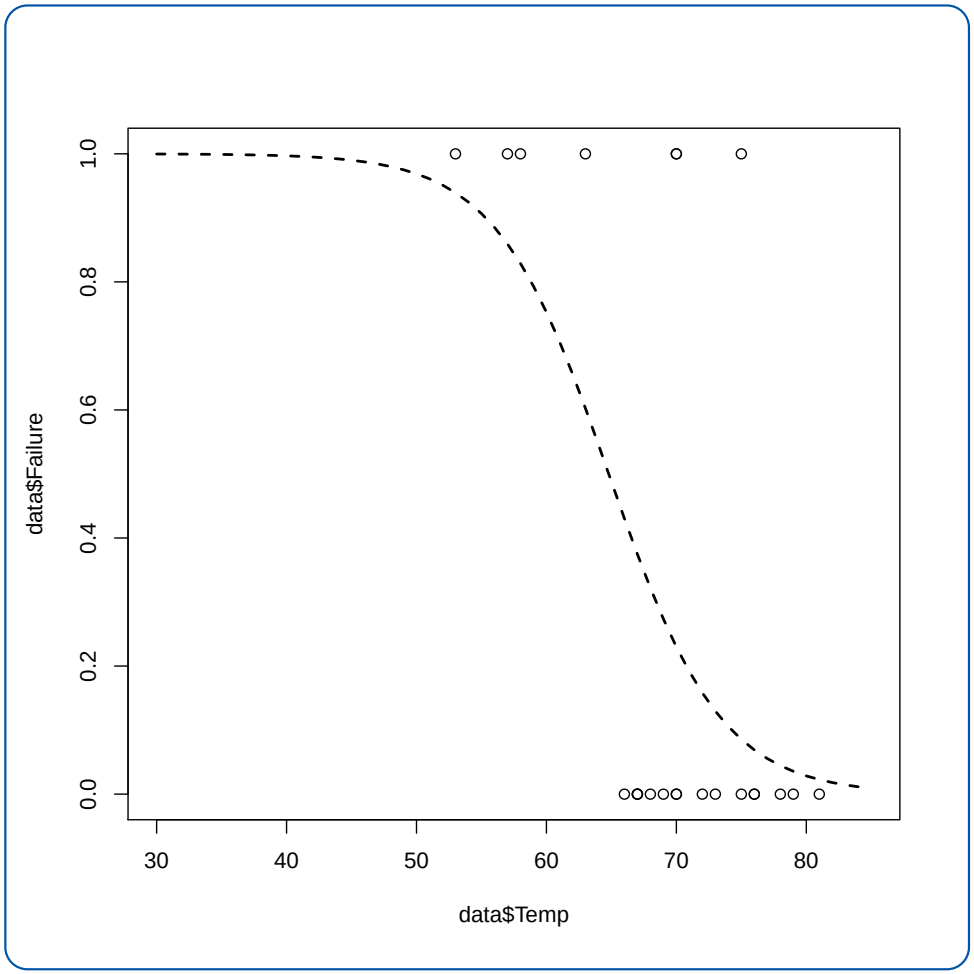
1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
53	57	58	63	66	67	67	67	68	69	70	70	70	70	72	73	75	75	76	76

The table reports failures with associated temperature.



```
library(gdata,quietly=TRUE,verbose=FALSE, warn.conflicts=FALSE)  # for reading xls
data = read.xls("ChallengerData.xls") # Taken From Casella & Berger (2002)
plot (data$Temp, data$Failure,xlim=c(30,85))

summary(out.int <- glm(Failure ~ Temp, family=binomial , data = data))

a= out.int$coefficients[1]
b= out.int$coefficients[2]
x=seq(30,85,by=1)
lines(x,exp(a+b*x)/(1+exp(a+b*x)))

x=31; exp(a+b*x)/(1+exp(a+b*x))
```

The estimated probability
for a failure at 31° is 0.9996088.

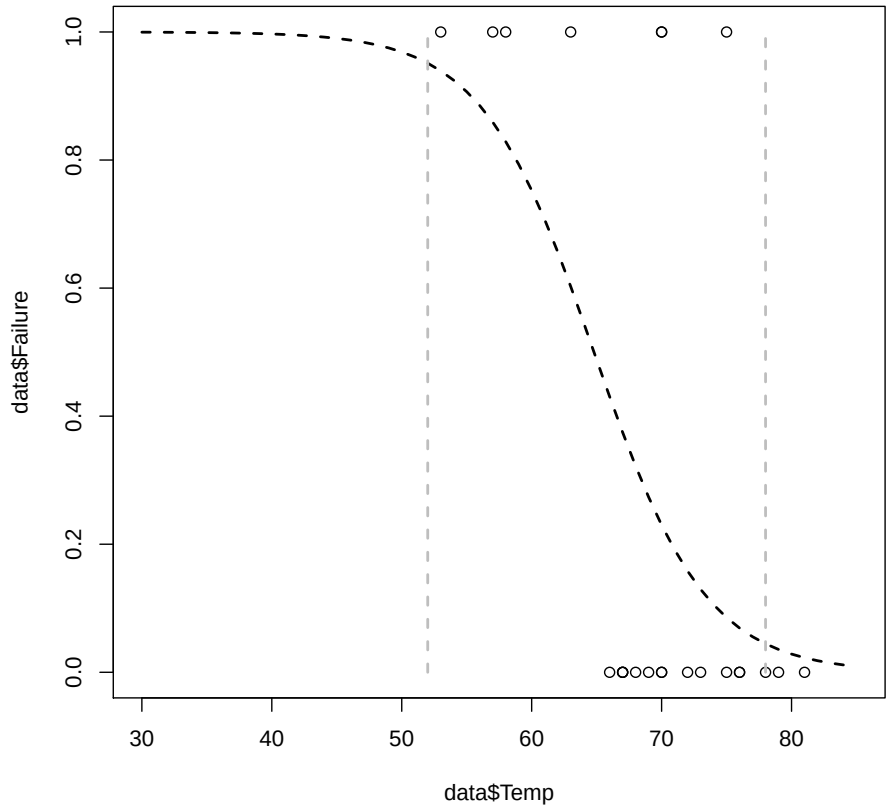


- Logistic regression naturally classifies the data into two fields: the ones with probability above 0.5, where we would optimally decide for outcome one and the ones with probability below 0.5, where we would decide for outcome 0.
- Hence, we obtain a **decision boundary**, given by the hyperplane

$$\alpha + \boldsymbol{\beta}\boldsymbol{x} = 0.$$

- If the decision boundary separates the two groups, then the data is called **linearly separable.** Note that this is can not be achieved in the Challenger dataset.
- Note that the logistic regression also provides probabilities of false decisions: at the boundary this is 50/50, but further out the probability of a false decision decrease. **Significant** decisions requires the probability of a false decision to be below a significance level, e.g. $\alpha = 0.05$ or $\alpha = 0.01$.

With significance level $\alpha = 0.05$ obtained decision boundaries.



Load the R example[8] from the homepage and revisit the above steps. Try your own examples.

[8] Called LogisticRegression.R

- The likelihood-function has to be maximized numerically.
- A first-order iterative scheme is the **gradient-descent** algorithm. Look this algorithm up and recall its properties and functionality.

Support vector machines

- The first example of a tool we visit but which is not typical for classical statistics is **Support Vector Machines**.
- For the introduction we mainly follow Hastie et. al. (2009) and Steinwart & Covel[9].

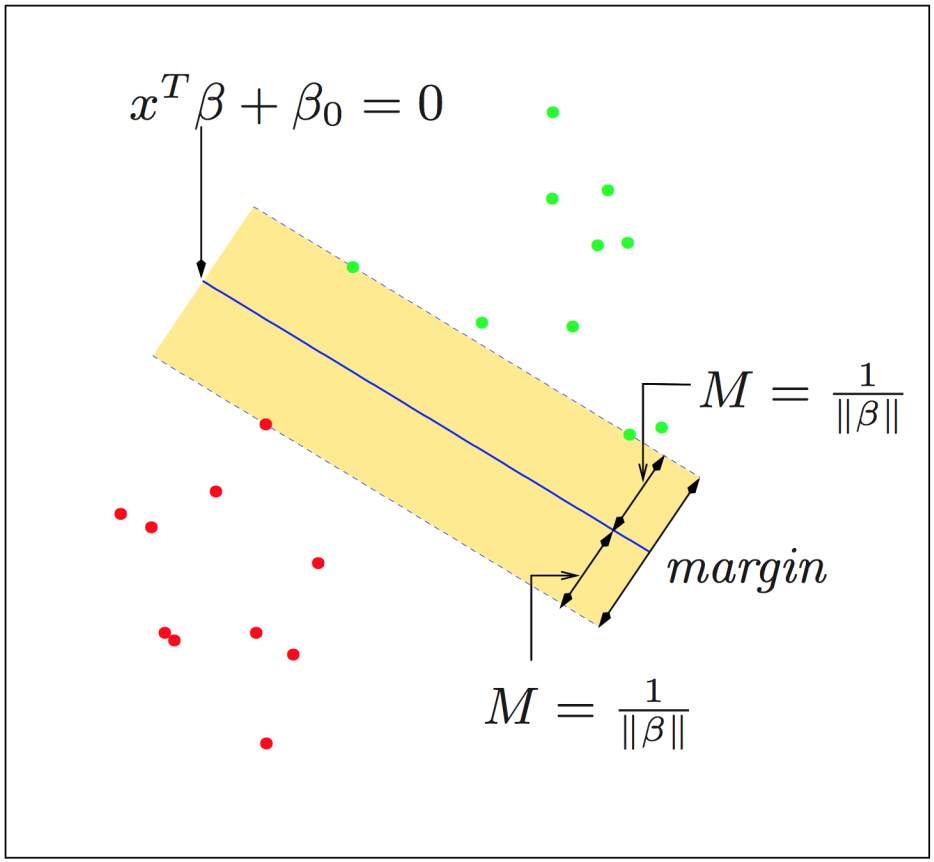
[9] I. Steinwart und C. Scovel (2007). „Fast rates for support vector machines using Gaussian kernels“. In: **Ann. Statist.** 35.2, S. 575–607.

- We start by formally introducing the **statistical classification problem**.
- We have a finite training set
$$T = ((x_1, y_1), \dots, (x_n, y_n)) \in (X \times Y)^n,$$
where $X \subset \mathbb{R}^d$ and $Y = \{-1, 1\}$.
- The standard **batch model** assumes that the samples $(x_i, y_i)_{1 \le i \le n}$ are i.i.d. according to an unknown probability measure P on $X \times Y$. Furthermore, a new sample (x, y) is drawn from P independently of T.
- A **classifier** $\mathscr{C}$ assigns to every T a measurable function $f = f_T : X \to \mathbb{R}$.
- The prediction of $\mathscr{C}$ for y is
$$\operatorname{sign} f(x)$$
with the convention $\operatorname{sign}(0) := 1$.
- We measure the quality of the classification f by the **classification risk**
$$\mathscr{R}(f) := P(\{(x, y) : \operatorname{sign} f(x) \neq y\}).$$

- Clearly, it is the goal to achieve the smallest possible risk, the so-called **Bayes risk**

$$\mathscr{R}_P := \inf\{\mathscr{R}(f) | f : X \to \mathbb{R} \text{ measurable}\}.$$

- A function which attains this level is called a **Bayes decision function**.
- Let us start with an illustrative introduction to SVM (Pictures taken from Hastie et.al. (2009))



Consider the hyperplane described by $x^\top \beta + \beta_0 = 0$, with $\| \beta \| = 1$ and the classification $\mathscr{G}$:

$$\text{sign}(x^\top \beta + \beta_0).$$

The maximal margin is obtained by the following optimization problem

$$\max_{\beta, \beta_0 : \|\beta\| = 1} M$$

subject to $y_i(x_i^\top \beta + \beta_0) \geq M, \ i = 1, \ldots, n$

- Such classifiers, computing a linear combination of the input and returning the sign were called **perceptrons** in the late 1950s (Rosenblatt, 1958) and set the foundations for later models of neural networks in the 80s and the 90s.
- We reformulate this criterion as follows: first, we get rid of $\| \beta \| = 1$ by considering

$$\frac{1}{\| \beta' \|} y_i (x_i^\top \beta' + \beta_0') \geq M$$

or, equivalently

$$y_i (x_i^\top \beta' + \beta_0') \geq M \| \beta' \|.$$

- With β', β_0' satisfying these equations, any (positive) multiple will also satisfy these, we rescale to $\| \beta' \| = M^{-1}$ and arrive at

$$\begin{aligned} &\min_{\beta, \beta_0} \frac{1}{2} \| \beta \|^2 \\ &\text{subject to } y_i (x_i^\top \beta + \beta_0) \geq 1, \ i = 1, \dots, n. \end{aligned} \tag{1}$$

- This is a convex optimization problem and can be solved via the classical Karush-Kuhn-Tucker conditions.
- It should be noted that the solution does only depend on a small amount of the data, and hence has a certain kind of robustness. On the other side, it will possibly not be optimal under additional information on the underlying distribution.

Non-linearly separable data

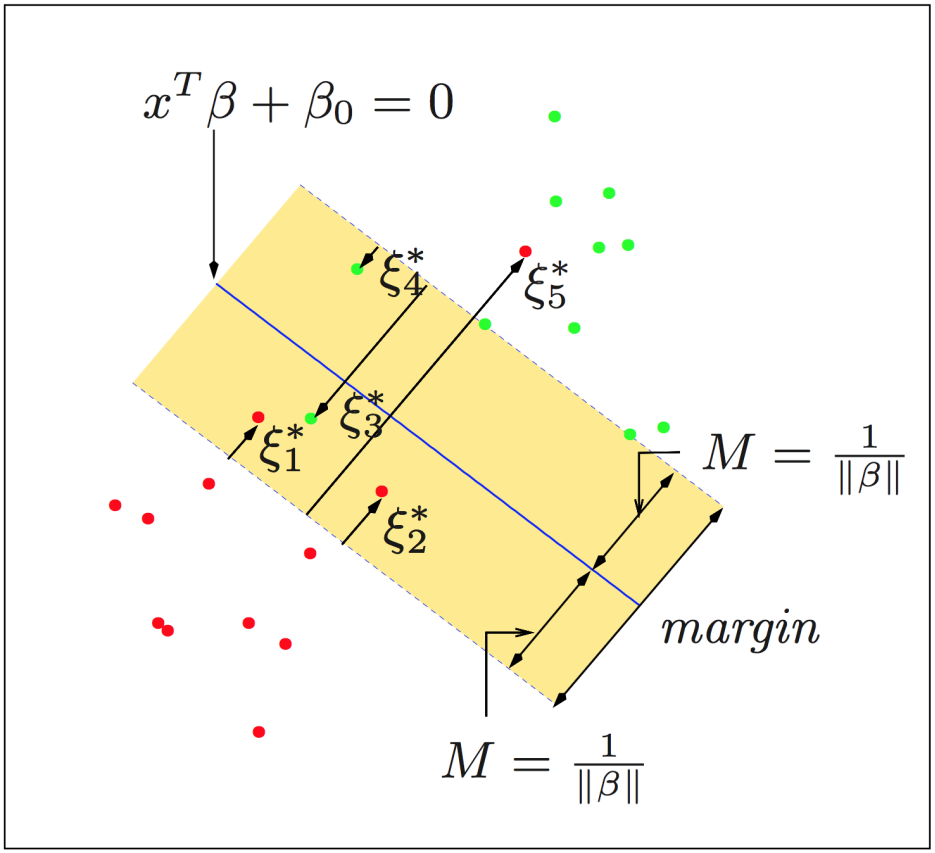
- When the data does not separate fully we will allow some points to be on the wrong side.
- In this regard, define the **slack variables** $\xi_1, \dots, \xi_n$ and consider

$$y_i(x_i^\top \beta + \beta_0) \geq M - \xi_i \tag{2}$$

or

$$y_i(x_i^\top \beta + \beta_0) \geq M(1 - \xi_i) \tag{3}$$

  with $\xi_i \geq 0$, $\sum \xi_i \leq K$ with a constant K.
- The first conditions seems more natural, while the second choice measures the overlap in relative distance, which chances with the width of the margin, M. However, (2) leads to a non-convex optimization problem. The second problem is convex and is the ”standard” support vector classifier.



The case for data which is not fully linearly separable.
Misclassification occurs when $\xi_i > 1$.

- Summarizing we arrive at the following minimization problem (again choosing $\| \beta \| = M^{-1}$) for the **support vector classifier**.

$$\min \| \beta \| \text{ subject to } \begin{cases} y_i(x_i^\top \beta + \beta_0) \geq (1 - \xi_i), \forall i \\ \xi_i \geq 0, \sum \xi_i \leq K. \end{cases}$$

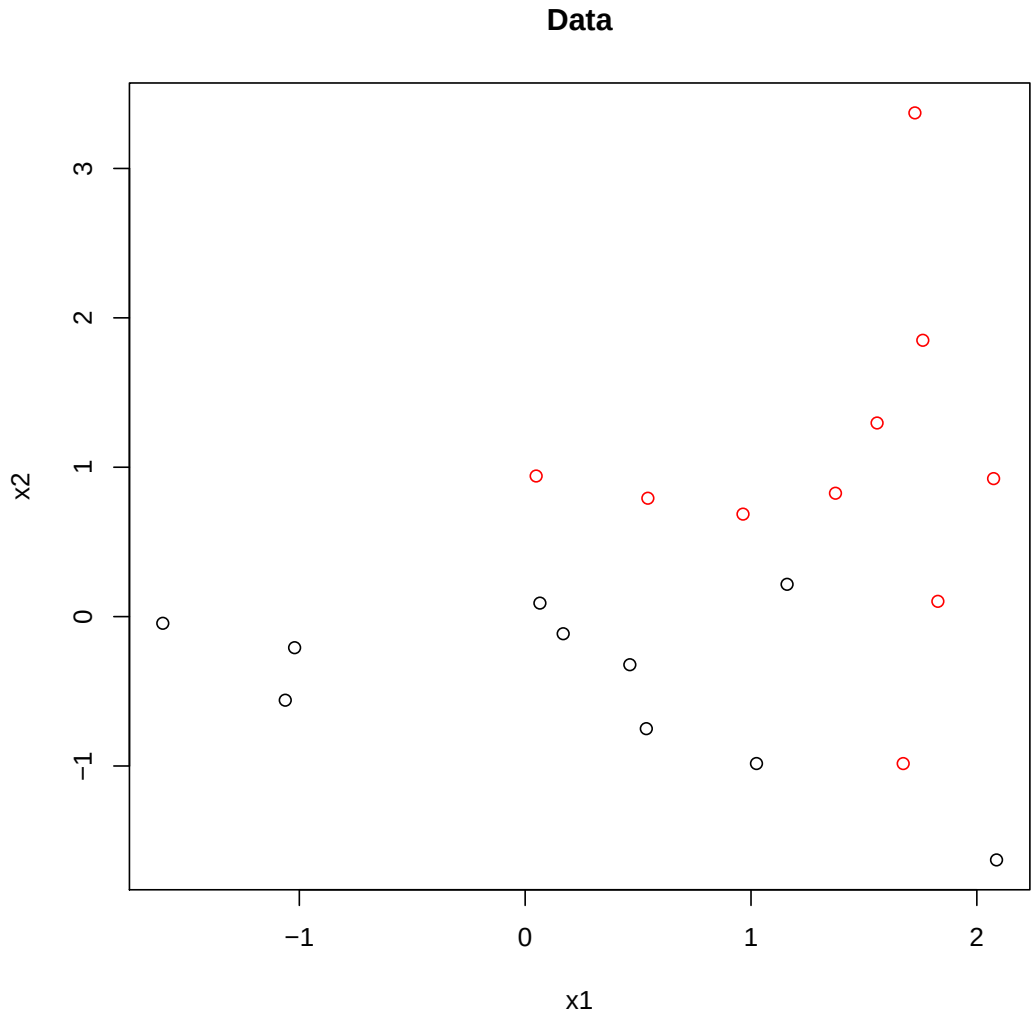
- For a detailed description we will revisit the Properties from Jungnickel (2014) in the following. Beforehand we illustrate the SVM with a small R example.

A toy example

The classical statistical example is to consider two groups of normal distributions with different mean.

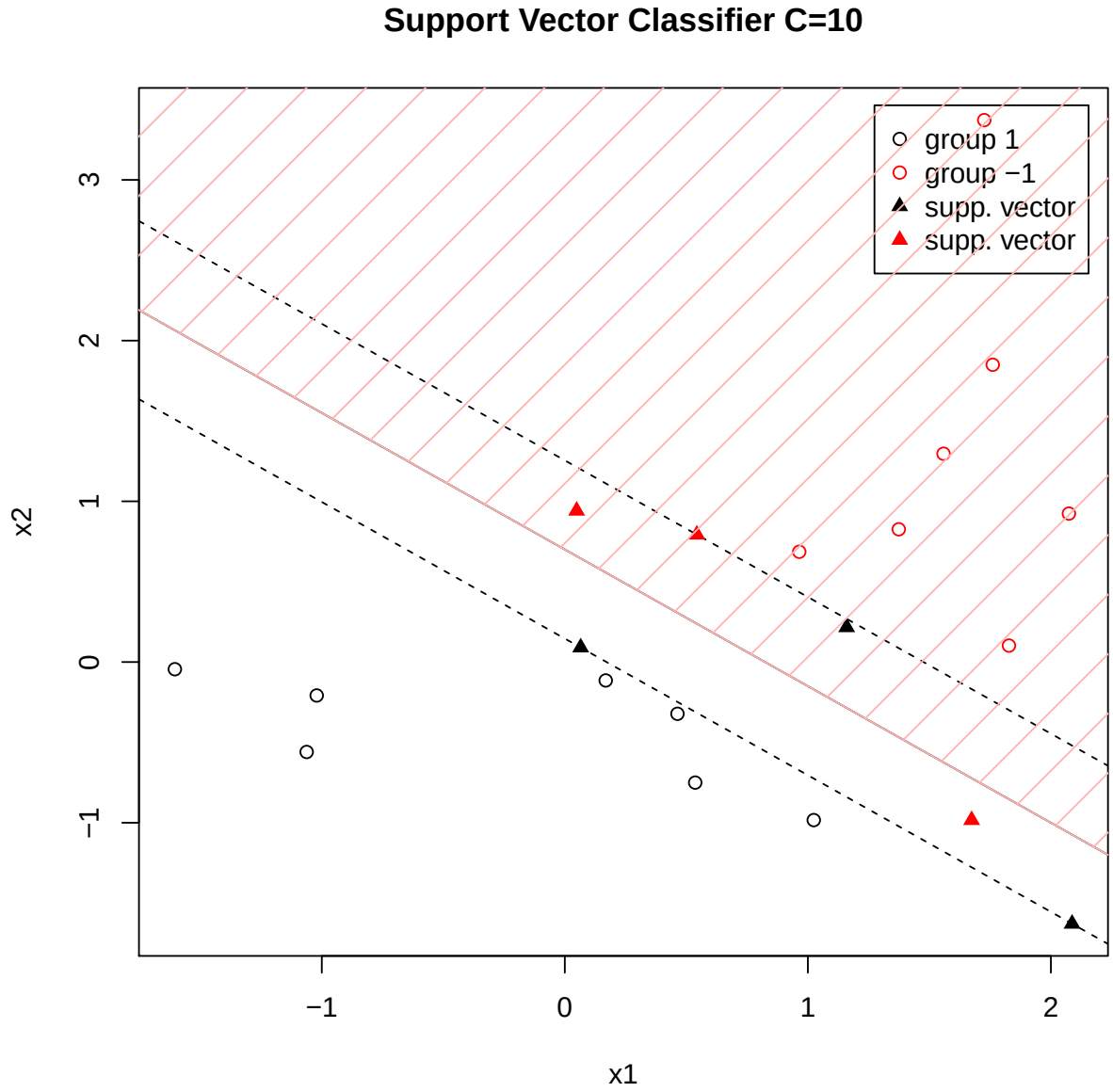
- Consider (X_i, Y_i) which are standard normal in group one and normal with mean $(1, 1)$ but same (identity) covariance matrix.
- The example is taken from Chapter 9 of James et al. (2013)

```
library(e1071)
%  N=20
%  x=matrix(rnorm(N*2), ncol=2)   # data - x corresponds to 2-dimensional data
%  y=c(rep(-1,N/2), rep(1,N/2))      # data - y has the classifier
%  x[y==1,]=x[y==1,] + 1           # We shift the mean for the second class by (1,1)
%  plot(x, col=(3-y))
```

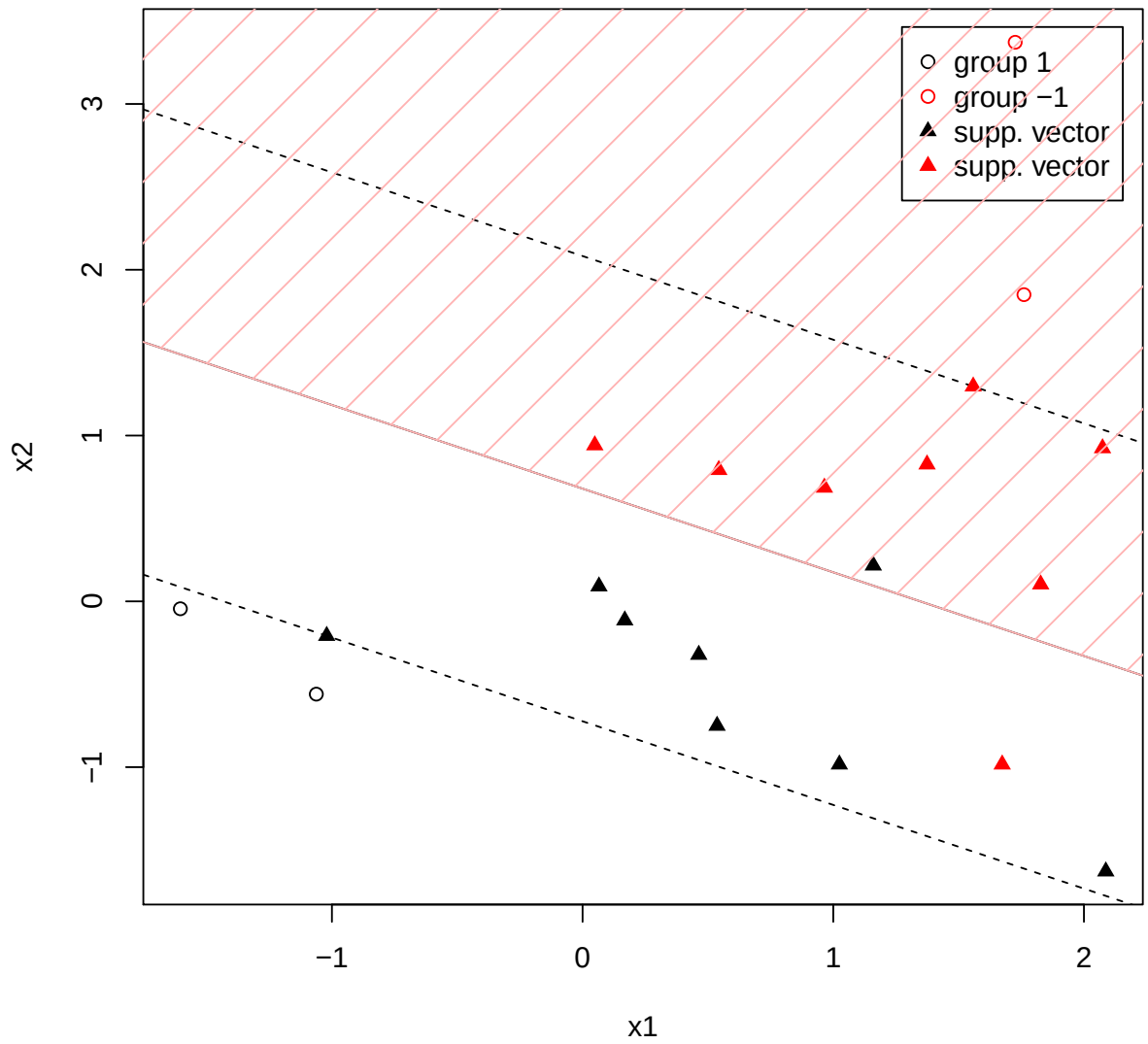


```
dat = data.frame(x=x, y=as.factor(y))
svmfit = svm( y ~ ., data=dat, kernel="linear", cost=10, scale=FALSE)

svmfit$index         # Identities of the support vectors
summary(svmfit)      # Summary
plot(svmfit,dat,color = terrain.colors) # Plot
```

- This small code fits a linear SVM with cost factor 10 to the data.
- Its output is the number and identities of the support vectors (unfortunately not the classification rule)



The summary plot: red / black decode the classes and the crosses are the support vectors. Note that we have four misclassifications.



A smaller parameter C leads to more misclassfications. What could be an optimal choice ?

The library provides a connection to cross-validation for the choice of parameters.

```
tune.out=tune(svm,y ~.,data=dat,kernel="linear",ranges=list(
      cost=c(0.001, 0.01, 0.1, 1,5,10,100)))
summary(tune.out)

#Parameter tuning of ‘svm’:
#- sampling method: 10-fold cross validation
#- best parameters:
# cost
# 0.1
#- Detailed performance results:
#   cost error dispersion
#1 1e-03  0.65  0.3374743
#2 1e-02  0.65  0.3374743
#3 1e-01  0.05  0.1581139
#4 1e+00  0.10  0.2108185
#5 5e+00  0.15  0.2415229
#6 1e+01  0.15  0.2415229
#7 1e+02  0.15  0.2415229
```

The kernel trick

- It seems quite restrictive to consider only linear classification rules. We have already seen that in linear regression we were able to overcome this problem by a suitable transformation of the data. This can also be achieved here and is often called **the kernel trick**.
- We first give a rather informative introduction and thereafter discuss the mathematical properties.

A short excursion to convex optimization

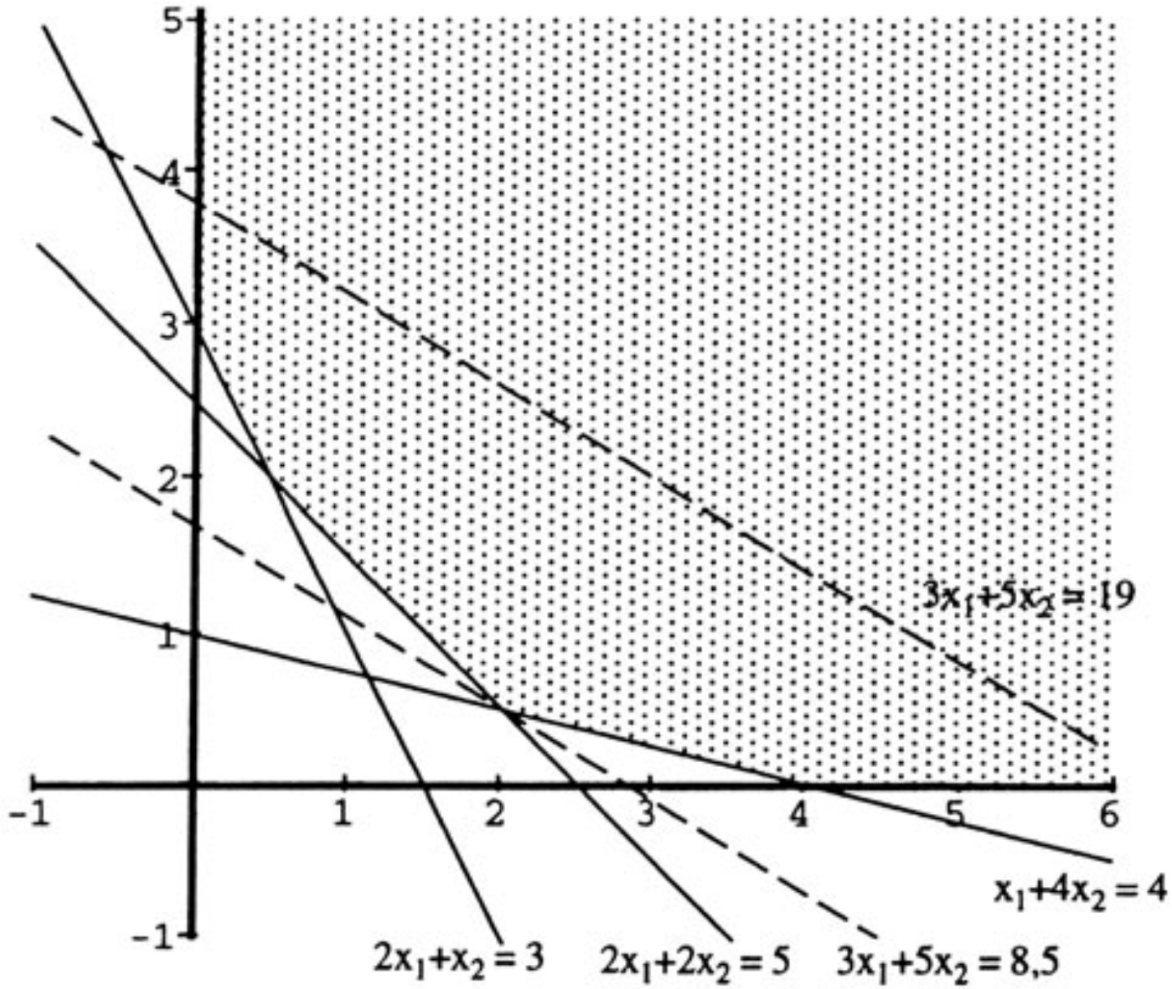
We follow Jungnickel[10]. Consider the **linear optimization problem**, also called **linear programm**

$$\begin{aligned} \min \quad & 3x_1 + 5x_2 \\ \text{subject to} \quad & 2x_1 + x_2 \geq 3 && (1) \\ & 2x_1 + 2x_2 \geq 5 && (2) \\ & x_1 + 4x_2 \geq 4 && (3) \\ & x_1 \geq 0 && (4) \\ & x_2 \geq 0 && (5) \end{aligned}$$

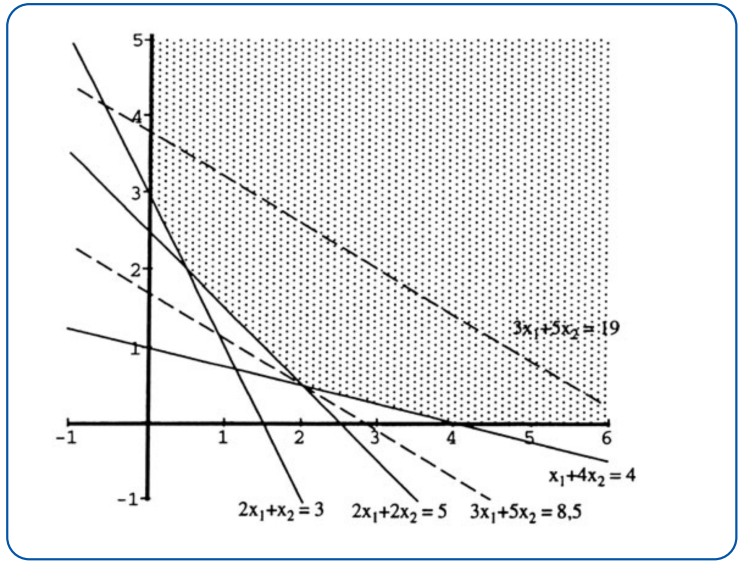
We illustrate the setting in the following picture. Besides this, we plot the target function.

$$3x_1 + 5x_2 = 19.$$

[10] D. Jungnickel (2014). **Optimierungsmethoden: Eine Einführung**. Springer-Verlag.



The optimal solution can be found by moving $3x_1 + 5x_2 = 19$ towards the origin $(0, 0)$.



- We realize that the optimal solution will be on one of the intersection points (this important observation leads to the **Simplex algorithm**)
- The optimal solution is $x^* = (2, 0.5)$ representing $3x_1 + 5x_2 = 8.5$
- As we obtained this geometrically, are we able to prove that this solution is optimal ?

$$
\begin{aligned}
\min \quad & 3x_1 + 5x_2 \\
\text{subject to} \quad & 2x_1 + x_2 \geq 3 && (1)\\
& 2x_1 + 2x_2 \geq 5 && (2)\\
& x_1 + 4x_2 \geq 4 && (3)\\
& x_1 \geq 0 && (4)\\
& x_2 \geq 0 && (5)
\end{aligned}
$$

- Observe, that such an equation might not always have a finite solution (eg. if one of the coefficients is negative)
- Any of the boundary conditions can be multiplied with positive constants. The boundary conditions can also be added. We could, for example look at $1.5\cdot(2)$ and obtain

$$3x_1 + 3x_2 \geq 7.5$$

- As, moreover $x_2 \geq 0$, we obtain necessarily $3x_1 + 5x_2 \geq 7.5$
- Can we do better ? Are we able to reach, e.g. 8.5 ??

The dual problem

$$
\begin{aligned}
\min \quad & 3x_1 + 5x_2 & \\
\text{subject to} \quad & 2x_1 + x_2 \geq 3 & (1)\\
& 2x_1 + 2x_2 \geq 5 & (2)\\
& x_1 + 4x_2 \geq 4 & (3)\\
& x_1 \geq 0 & (4)\\
& x_2 \geq 0 & (5)
\end{aligned}
$$

- Multiplying the inequalities with y_i, we arrive at the following scheme: let us find multipliers $y_1,\ y_2,\ y_3 \geq 0$, such that

$$
\begin{aligned}
2y_1 + 2y_2 + y_3 &\leq 3\\
y_1 + 2y_2 + 4y_2 &\leq 5\\
y_1, y_2, y_3 &\geq 0.
\end{aligned}
$$

- The r.h.s. should be **maximized**, to be as close to x^* as possible, such that the objective function is

$$\max \quad 3y_1 + 5y_2 + 4y_3.$$

- This is the so-called dual problem. We can easily verify that $x^* = (2, 0.5)$ verifies the dual problem, hence is optimal.

Arguing similar, but more general we arrive at the following result.

Proposition (Weak duality)

Consider $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$ and the two problems

$$\min c^\top x, \quad \text{subject to} \quad Ax \geq b, x \geq 0 \tag{P}$$

$$\max y^\top b, \quad \text{subject to} \quad y^\top A \leq c^\top, y \geq 0. \tag{D}$$

Then, for any x_0 satisfying the constraints in (P) and y_0 the ones in (D) it holds that

$$y_0^\top b \leq c^\top x_0.$$

The proof is fairly easy:

$$y_0^\top b \leq y_0^\top (Ax_0) = (\overrightarrow{y_0^\top} A) x_0 \leq c^\top x_0.$$

It can even be shown that in this case $y^{*\top} = c^\top x^*$, see Satz 2.9.17 in Jungnickel (2014).

Convex optimization

Actually we are interested in convex optimization problems. More precisely, we consider the following **minimization problem (MP)**

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & g(x) \leq 0 \\ & h(x) = 0, \end{aligned} \tag{MP}$$

where $x \in X \subset \mathbb{R}^n$ and $g : X \to \mathbb{R}^m$, $h : X \to \mathbb{R}^p$. We denote by S the set where the constraints in (MP) are satisfied. Define the **Lagrange function**

$$\ell(x, \lambda, \mu) := f(x) + \lambda^\top h(x) + \mu^\top g(x).$$

Define the **dual function** $\theta(\lambda, \mu) := \inf\{\ell(x, \lambda, \mu) : x \in X\}$. Then the **dual problem** to (MP) is

$$\begin{aligned} \max \quad & \theta(\lambda, \mu) \\ \text{subject to} \quad & \mu \geq 0. \end{aligned} \tag{DP}$$

Again, we have strong duality under additional assumptions, see Satz 6.5.10 in Jungnickel (2014).

Proposition (Strong duality)

Let X be a convex set and $f : X \to \mathbb{R}$, $g : X \to \mathbb{R}^p$ be convex functions and $h : X \to \mathbb{R}^m$ be an affine function.
If there exists $x_0 \in X$, s.t. $g(x_0) < 0$, $h(x_0) = 0$ and $0 \in \text{int } h(X)$, then

$$\sup\{\theta(\lambda, \mu) : \mu \geq 0\} = \inf\{f(x) : x \in S\}.$$

The following necessary conditions are found in Satz 2.9.9. in Jungnickel (2014).

It is also very useful to obtain necessary conditions, however in the simpler optimization problem where we only have inequality constraints. Set $I(x) = \{i : g_i(x) = 0\}$. We consider

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & g(x) \leq 0, \quad x \in X. \end{aligned} \tag{MPIC}$$

Proposition (Karush-Kuhn-Tucker conditions)

Let X be open and x an admissible point and g_i differentiable in x for all $i \in I(x)$ and continuous for $i \notin I(x)$. Moreover, let

$$\nabla g_i(x), i \in I(x) \text{ be linearly independent.} \tag{LICQ}$$

If x is a local minimum, then there exist μ_i, $i \in I(x)$, s.t.

$$\nabla f(x) + \sum_{i \in I(X)} \mu_i \nabla g_i(x) = 0.$$

If g_i are differentiable for all i, we obtain

$$\nabla f(x) + \sum_i \mu_i \nabla g_i(x) = 0,$$

$$\mu_i \geq 0, \text{ and } \mu_i g_i(x) = 0, \quad i = 1, \ldots, p.$$

Back to the kernel trick

We are interested in the following convex problem:

$$\begin{aligned} \min_{\beta,\beta_0,\xi} \quad & \frac{1}{2} \parallel \beta \parallel^2 + C \sum_{i=1}^{N} \xi_i \\ \text{subject to} \quad & y_i(x_i^\top \beta + \beta_0) \geq (1-\xi_i), \forall i \\ & \xi_i \geq 0. \end{aligned} \tag{MP}$$

The Lagrange function is

$$\frac{1}{2} \parallel \beta \parallel^2 + C \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i (y_i(x_i^\top \beta + \beta_0) - (1-\xi_i)) - \sum_{i=1}^{N} \mu_i \xi_i$$

with positivity constraints $\alpha, \mu, \xi \geq 0$.

The Lagrange function is

$$\frac{1}{2} \parallel \beta \parallel^2 + C \sum_{i=1}^{N} \xi_i - \sum_{i=1}^{N} \alpha_i (y_i (x_i^\top \beta + \beta_0) - (1 - \xi_i)) - \sum_{i=1}^{N} \mu_i \xi_i.$$

We see that this function is differentiable in β, β_0 and ξ. Setting the respective derivatives to zero, we obtain

$$\beta = \sum_{i=1}^{N} \alpha_i y_i x_i, \quad 0 = \sum_{i=1}^{N} \alpha_i y_i, \quad \alpha_i = C - \mu_i. \tag{4}$$

Inserting this into the Lagrangian we obtain the Wolfe dual objective function

$$\ell_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^\top x_j,$$

giving a lower bound of the objective function of (MP).

$$\nabla f(x) + \sum_i \mu_i \nabla g_i(x) = 0,$$

$$\mu_i \geq 0, \text{ and } \mu_i g_i(x) = 0, \quad i = 1, \ldots, p,$$

x admissible.

In addition, we obtain the conditions

$$\begin{aligned} \alpha_i \left(y_i (x_i^\top \beta + \beta_0) - (1 - \xi_i) \right) &= 0, \\ \mu_i \xi_i &= 0, \\ y_i (x_i^\top \beta + \beta_0) - (1 - \xi_i) &\geq 0. \end{aligned}$$

- We can also make a further observation: from (4), we see that the solution for β has the form

$$\beta^* = \sum_{i=1}^{N} \alpha_i^* y_i x_i,$$

with nonzero coefficients α_i^* for those observations i which lie on the margin (the **support vectors**).

- β_0^* can be computed by the average of all solutions and we arrive at the decision function

$$\hat{G}(x) = \text{sign}[x^\top \beta^* + \beta_0^*].$$

- An important observation is that the optimization problem now only depends on the scalar product

$$x_i^\top x_j$$

which opens the scenery for reproducing kernel Hilbert spaces.

Reproducing kernel Hilbert spaces

For details on this subject, consider C. Berg, J. P. R. Christensen und P. Ressel (1984). **Harmonic analysis on semigroups**. Springer-Verlag.

Definition

Let X be a nonemtpy set. A function $K : X \times X \to \mathbb{C}$ is called a **kernel**, if

$$\sum_{i,j=1}^{n} c_i \bar{c}_j K(x_i, x_j) \geq 0$$

for all $n \in \mathbb{N}$, $\{x_1, \ldots, x_n\} \subset X$ and $\{c_1, \ldots, c_n\} \subset \mathbb{C}$. If we replace $\mathbb{C}$ by $\mathbb{R}$, we call the kernel real-valued.

Example

Let $\phi : X \to \mathbb{C}$ be arbitrary. The $K(x_1, x_2) := \phi(x_1)\phi(x_2)$ is positive definite:

$$\sum_{i,j=1}^{n} c_i \bar{c}_j K(x_i, x_j) = \Big| \sum_{i=1}^{n} c_i f(x_i) \Big|^2 \geq 0.$$

There are also negative definite kernels (which is not of interest here), and more precisely one calls the above kernel a positive definite kernel.

Now consider such a kernel K and the linear subspace $H_0 \subset \mathbb{C}^X$ generated by the functions

$$\{K_x : x \in X\},$$

where $K_x(y) = K(x,y)$.
We introduce a (well-defined) scalar product for $f = \sum a_i K_{x_i}$, $g = \sum b_j K_{x_j}$ by

$$\langle f, g \rangle := \sum_{i,j} a_i \bar{b}_j K(x_i, x_j).$$

Then, H_0 is a pre-Hilbert space and its completion H is a Hilbert space.

Most importantly, the scalar product has the **reproducing property**

$$\langle f, K_x \rangle = \sum_i a_i K(x_i, x) = f(x)$$

for all $f \in H_0$ and $x \in X$. This is why the space H is called **reproducing kernel Hilbert space**.

Mercer's theorem allows to obtain a different view on the kernel K. Associate the integral operator

$$T_K f := \int_X K(.,x) f(x) d\nu(x)$$

to the kernel K (in $L^2(X)$). The the spectracl decomposition of T_K yields

$$K(x_1,x_2) = \sum_n \lambda_n \phi_n(x_1) \bar{\phi}_n(x_2). \tag{5}$$

The RKHS is given by

$$H = \{f \in L_2(X) : \sum_n \lambda_n^{-1} \langle f, \phi_n \rangle^2 < \infty\}.$$

This yields in turn the representation

$$f(x) = \sum_n a_n \phi_n(x).$$

On the other side, any kernel satisfying (5) is non-negative definite and thus leads to a RKHS.

Summarizing, we obtain the following:

- A RKHS allows a representation
$$f(x) = \sum_n a_n \phi_n(x)$$
with an associated kernel K satisfying
$$K(x_1, x_2) = \sum_n \lambda_n \phi_n(x_1) \bar{\phi}_n(x_2).$$
- One can think of transforming x to $\phi_n(x)$ such that
$$\{(\phi_n(x))_{n=1}^{\infty} : x \in X\}$$
is called the feature space. This can in principle be an infinite dimensional space.

Kernel examples

From T. Evgeniou, M. Pontil und T. Poggio (2000). „Regularization networks and support vector machines“. In: **Advances in computational mathematics** 13.1, S. 1–50 we cite the following list of kernels:

Table 1

Some possible kernel functions. The first four are radial kernels. The multiquadric and thin plate splines are positive semidefinite and thus require an extension of the simple RKHS theory of this paper. The last three kernels were proposed by Vapnik [96], originally for SVM. The last two kernels are one-dimensional: multidimensional kernels can be built by tensor products of one-dimensional ones. The functions B_n are piecewise polynomials of degree n, whose exact definition can be found in [85].

Kernel function	Regularization Network
$K(\mathbf{x}-\mathbf{y}) = \exp(-\|\mathbf{x}-\mathbf{y}\|^2)$	Gaussian RBF
$K(\mathbf{x}-\mathbf{y}) = (\|\mathbf{x}-\mathbf{y}\|^2 + c^2)^{-1/2}$	Inverse multiquadric
$K(\mathbf{x}-\mathbf{y}) = (\|\mathbf{x}-\mathbf{y}\|^2 + c^2)^{1/2}$	Multiquadric
$K(\mathbf{x}-\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|^{2n+1}$	Thin plate splines
$K(\mathbf{x}-\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|^{2n} \ln(\|\mathbf{x}-\mathbf{y}\|)$	
$K(\mathbf{x},\mathbf{y}) = \tanh(\mathbf{x}\cdot\mathbf{y}-\theta)$	(only for some values of θ) Multi-Layer Perceptron
$K(\mathbf{x},\mathbf{y}) = (1+\mathbf{x}\cdot\mathbf{y})^d$	Polynomial of degree d
$K(x,y) = B_{2n+1}(x-y)$	B-splines
$K(x,y) = \dfrac{\sin(d+1/2)(x-y)}{\sin((x-y)/2)}$	Trigonometric polynomial of degree d

- The choice of the kernel should be adapted to the question and influences the performance massively !
- Most important are probably: Gaussian (RBF), polynomial and splines. But there are many other choices, for example in speech recognition (where one uses **string kernels**...)

MNIST dataset with SVM

Lets look at an example: our aim is to classify handwritten digits with SVM.

- We will use (parts of) the MNIST[11] dataset, available at `http://yann.lecun.com/exdb/mnist/`.
- This includes 60.000 images of handwritten digits (0-9) with classification. ($\rightarrow$ wich learning ?)
- Lets get an impression of the first one hundred pictures:

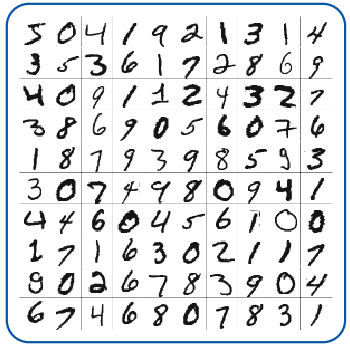
[11] Modified National Institute of Standards and Technology database.

First 100 digits from the MNIST dataset.

```
# Taken from https://gist.github.com/brendano/39760
# Copyright under the MIT licens:
# I hereby license it as follows. This is the MIT license.
# Copyright 2008, Brendan O'Connor
# Permission is hereby granted, free of charge, to any person obtaining a copy of this softwar
# and associated documentation files (the "Software"), to deal in the Software without restric
# including without limitation the rights to use, copy, modify, merge, publish, distribute,
# sublicense, and/or sell copies of the Software, and to permit persons to whom the Software i
# furnished to do so, subject to the following conditions: The above copyright notice and this
# permission notice shall be included in all copies or substantial portions of the Software.
```

```
library(e1071)

load_mnist <- function() {
  load_image_file <- function(filename) {
    ret = list()
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    ret$n = readBin(f,'integer',n=1,size=4,endian='big')
    nrow = readBin(f,'integer',n=1,size=4,endian='big')
    ncol = readBin(f,'integer',n=1,size=4,endian='big')
    x = readBin(f,'integer',n=ret$n*nrow*ncol,size=1,signed=F)
    ret$x = matrix(x, ncol=nrow*ncol, byrow=T)
    close(f)
    ret
  }
  load_label_file <- function(filename) {
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    n = readBin(f,'integer',n=1,size=4,endian='big')
    y = readBin(f,'integer',n=n,size=1,signed=F)
    close(f)
    y
  }
  train <<- load_image_file('train-images-idx3-ubyte')     # Images
  test <<- load_image_file('t10k-images-idx3-ubyte')

  train$y <<- load_label_file('train-labels-idx1-ubyte')   # Labels
  test$y <<- load_label_file('t10k-labels-idx1-ubyte')  }

show_digit <- function(arr784, col=gray(12:1/12), ...) {
  image(matrix(arr784, nrow=28)[,28:1], col=col, ...)}
```

```
setwd("somepath/MNIST")
load_mnist()

# We plot the first 100 images
par(mfrow=c(10,10),mai=c(0,0,0,0))
for (i in 1:100) {
  show_digit(train$x[i,],col=gray(12:1/12),xaxt="n", yaxt="n")
}
```



```
N=2000
dat=data.frame(x=train$x[1:N,],y=as.factor(train$y[1:N]))
svmfit = svm (y ~., data=dat,   method="class", kernel="linear", cost=10, scale=FALSE)
```

- This is the SVM-routine. Note that we chose linear and cost factor $C = 10$.
- Additionally, we only use 2000 samples for training (otherwise it takes quite long. 10.000 is feasiable...)

```
#prediction
testdat = data.frame(x=test$x[1:100,])
pred = predict(svmfit,testdat)

table(pred,test$y[1:100])
sprintf("Error rate: %f",1-sum(pred == test$y[1:100])/100)

# This gives the following result:


pred  0  1  2  3  4  5  6  7  8  9
   0  8  0  0  0  0  0  0  0  0  0
   1  0 14  0  0  0  0  0  0  0  0
   2  0  0  8  0  0  1  2  0  0  0
   3  0  0  0 11  0  0  0  0  0  0
   4  0  0  0  0 14  0  0  0  0  0
   5  0  0  0  0  0  6  0  0  0  0
   6  0  0  0  0  0  0  8  0  0  0
   7  0  0  0  0  0  0  0 14  0  0
   8  0  0  0  0  0  0  0  0  2  0
   9  0  0  0  0  0  0  0  1  0 11

"Error rate: 0.040000"
```

Which is already remarkable. Which of the images are misclassified ?

The multi-class problem

Actually, in the previous example we did have **10** classes and not only 1!

- The method is actually not designed to cope with more than one classes.
- Some heuristic methods have been proposed to overcome this deficiency: **one-vs-one** or **one-vs-all** comparison.

One-versus-one Classification

Think you have K classes and data points $(x_i, y_i)_{i=1}^n$. Then you divide in the $K(K-1)/2$ classification problems (x_i, z_i^{kl}),

$$z_i^k = \begin{cases} 1 & \text{if } y_i = k, \\ -1 & \text{if } y_i = l. \end{cases}$$

The decision rule is to assign those class which appears most often in the single SVMs.

One-versus-all Classification

Think you have K classes and data points $(x_i, y_i)_{i=1}^n$. Then you divide in the K classification problems (x_i, z_i^k),

$$z_i^k = \begin{cases} 1 & \text{if } y_i = k, \\ -1 & \text{otherwise.} \end{cases}$$

Denote the estimated vectors in these SVMs by $\beta_1, \ldots, beta_k$ the result of the fits. For a test observation x we choose the class k for which $\beta_k x$ is largest.

As is obvious, both methods have their difficulties in multi-class classification.

Both approaches try to find a measure for the quality of the estimation. One also would like to find something like a probability relating to the fit and a common approach is to compute

$$\sigma(a\hat{\beta}x+b)$$

where σ is the sigmoid function and a and b are estimated by maximum-likelihood. As commented in Murphy (2012), a. a. O., page 504, this has difficulties in general (there is no reason why $\hat{\beta}x$ leads to a probability).

Virtual Support Vector Machines

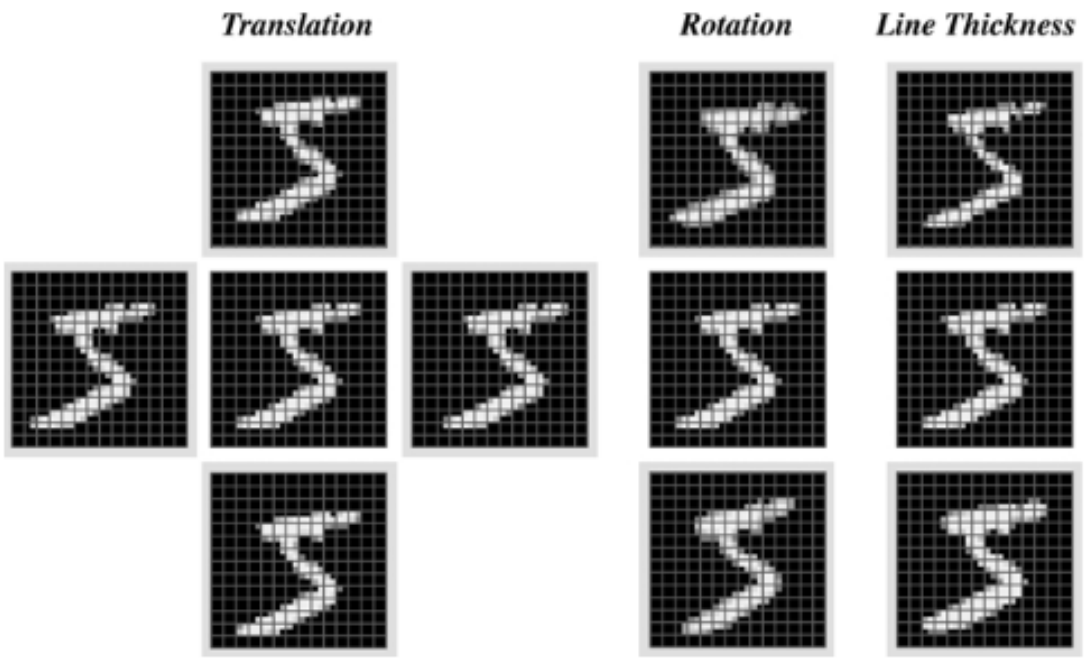
- The record-holding SVM on the MNIST database uses a **Virtual** SVM, see D. Decoste und B. Schölkopf (2002). „Training invariant support vector machines“. In: **Machine learning** 46.1, S. 161–190.
- We cite: ”Practical experience has shown that in order to obtain the best possible performance, prior knowledge about invariances of a classification problem at hand ought to be incorporated into the training procedure”
- Indeed, currently we did not take this into account at all, but simply trained the data. So how could this be done ?

- One could try to find optimal kernel functions just tailor-made to the problem.
- One could generate **virtual** examples from the training set by applying some standard transformations.
- A combination of these approaches.

For the MNIST digits one can use for example translation, rotation or in-/decrease in line thickness.

The consequence is that much more support vectors are found, suggesting that the methods improves the estimation. The optimal choice with error rate of **0.56%** uses a polynomial kernel of degree 9

$$K(x,y) = \frac{1}{512}(x \cdot y + 1)^9.$$



Source: Decoste & Schölkopf (2002)

Table 1. Comparison of Support Vector sets and performance for training on the original database and training on the generated Virtual Support Vectors. In both training runs, we used polynomial classifier of degree 3.

Classifir trained on	Size	Av. no. of SVs	Test error
Full training set	7291	274	4.0%
Overall SV set	1677	268	4.1%
Virtual SV set	8385	686	3.2%
Virtual patterns from full DB	36455	719	3.4%

Virtual Support Vectors were generated by simply shifting the images by one pixel in the four principal directions. Adding the unchanged Support Vectors, this leads to a training set of the second classifier which has five times the size of the first classifier's overall Support Vector set (i.e. the union of the 10 Support Vector sets of the binary classifiers, of size 1677—note that due to some overlap, this is smaller than the sum of the ten support set sizes). Note that training on virtual patterns generated from *all* training examples does not lead to better results han in the Virtual SV case; moreover, although the training set in this case is much larger, it hardly leads to more SVs.

Source: Decoste & Schölkopf (2002)

Recall that number 9 was misclassified (6 instead of 5)

```
test=attr(pred,"decision.values")[9,]
sign(test)

5/0 5/4 5/1 5/9 5/2 5/3 5/6 5/7 5/8 (...)
 -1   1   1   1  -1   1  -1   1   1  (...)
```

While 6 gets all ones.

We try an alternative statistic (now for 20.000 data):

```
testnorm=test/max(test)
labels1=c("5/0","5/1","5/2","5/3","5/4","5/6","5/7","5/8","5/9")
labels2=c("0/6","1/6","2/6","3/6","4/6","5/6","6/7","6/8","9/6")

> testnorm[labels1]
       5/0         5/1         5/2         5/3         5/4         5/6         5/7         5/8         5/9
 0.04257473  0.21475787  0.01911578  0.77776091  0.03457997 -0.28087561  0.08408199  0.25468632  0.15987355
> factors*testnorm[labels2]
       0/6        1/6        2/6        3/6        4/6        5/6        6/7        6/8        9/6
0.08472909 0.26445613 0.37370402 0.19366219 0.16418205 0.28087561 0.15263579 0.29240588 0.06712401
```

Adaptive basis function models

- As we saw in the previous chapter, a common classification approach is to consider a prediction of the form
$$f(x) = \beta\phi(x)$$
where ϕ is determined via a kernel, such that $\phi(v) = (\kappa(v,x_1),\dots(\kappa(v,x_n))$ with data points $x_1,\dots,x_n$.
- The question is how to obtain an optimal kernel and how to estimate the associated parameters.
- An alternative is to learn ϕ directly from the data ! This is done in the so-called ABMs (adaptive basis function models): the starting point is to study
$$f(x) = w_0 + \sum_{m=1}^{M} w_m\phi_m(x)$$
with weights $w_0,\dots,w_M$ and **basis functions** $\phi_1,\dots,\phi_M$.

- A typical approach is a parametric specification of the kernel function, i.e.

$$\phi_m(x) = \phi(x, v_m)$$

where $v_1, \ldots, v_M$ are the parameters of the kernel. The entire parameter set is denoted by $\theta := (w_0, \ldots, w_M, v_1, \ldots, v_M)$.

CART

- The first example will be **classification and regression trees** (CART).
- A **classification tree** has as an output classes and a **regression tree** gives real numbers instead.
- The idea is to partition the data suitably. Most commonly are half-planes i.e. we use as classification rules

$$\mathbb{1}_{\{x \leq t\}}$$

and various combinations of thereof. The classficiation in this case is the partition given by

$$\{x \in \mathbb{R} : x \leq t\} \quad \text{versus} \quad \{x \in \mathbb{R} : x > t\}$$

and leads to a **binomial tree**.

We study the famous example of Titanic passenger data. In fact this dataset is included in the R package `rpart.plot` and contains 1046 datapoints with observations on the passenger class, survival, sex, age, sibsp (number of spouses or siblings aboard), parch (number of parents or children aboard).

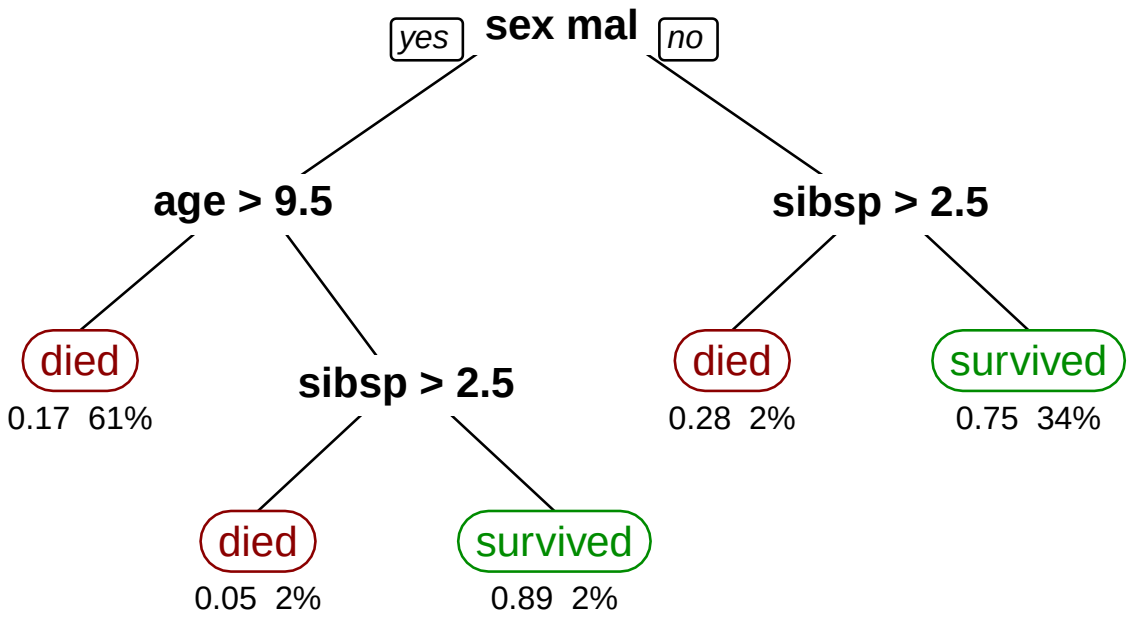
```
> data(ptitanic)
> summary(ptitanic)
 pclass        survived       sex           age              sibsp           parch
 1st:323   died    :809   female:466   Min.   : 0.1667   Min.   :0.0000   Min.   :0.000
 2nd:277   survived:500   male  :843   1st Qu.:21.0000   1st Qu.:0.0000   1st Qu.:0.000
 3rd:709                              Median :28.0000   Median :0.0000   Median :0.000
                                      Mean   :29.8811   Mean   :0.4989   Mean   :0.385
                                      3rd Qu.:39.0000   3rd Qu.:1.0000   3rd Qu.:0.000
                                      Max.   :80.0000   Max.   :8.0000   Max.   :9.000
                                      NA's   :263
```

A classification tree can be obtained as follows:

```
library(rpart)
library(rpart.plot)
data(ptitanic)

fit = rpart(survived ~  sex + age + sibsp, data=ptitanic, method="class")
cols <- c("darkred", "green4")[fit$frame$yval]
prp(fit,tweak=1.4 , extra=106, under=TRUE, ge=" > ", eq=" ", col=cols)
```

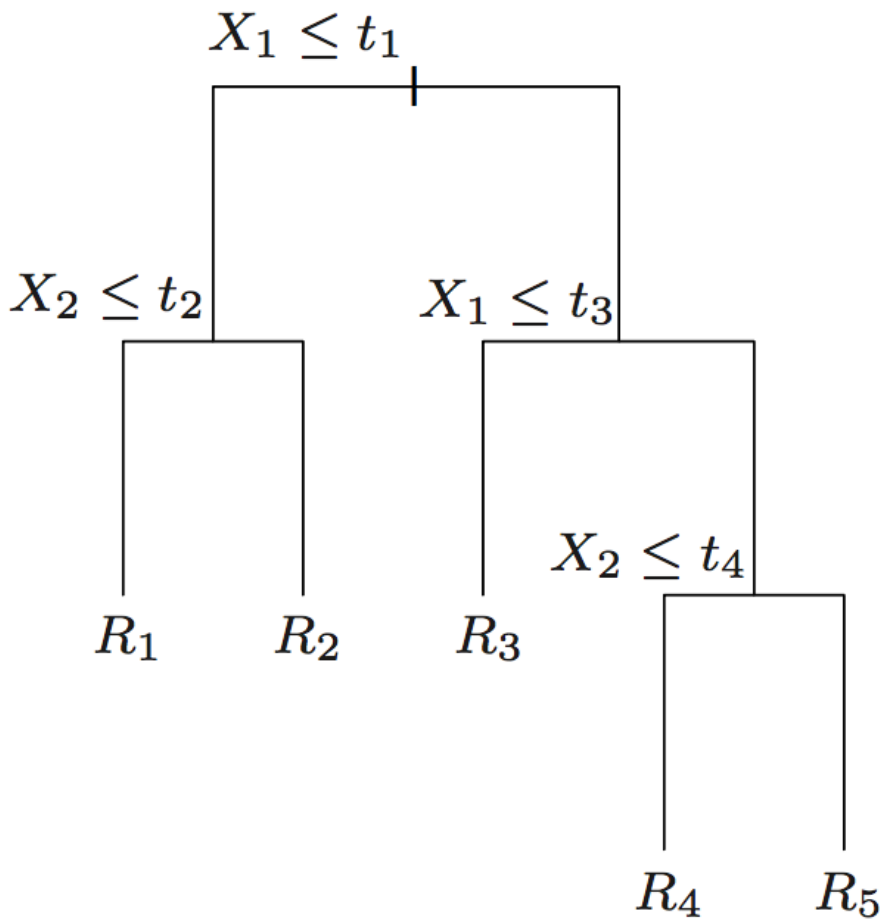
The plot is inspired by the graphic from Stephen Milborrow[12].
The titanic dataset is also part of a Kaggle competition and there is a very nice blog by Trevor Stephens[13] one how to fit this dataset. In particular, it is interesting how important some simple engineering techniques are (exploiting what really is in your data)! This is, however, not applicable here because no names are provided in the titantic dataset.

[12] See `https://commons.wikimedia.org/wiki/File:CART_tree_titanic_survivors.png`
[13] `http://trevorstephens.com/kaggle-titanic-tutorial/getting-started-with-r/`

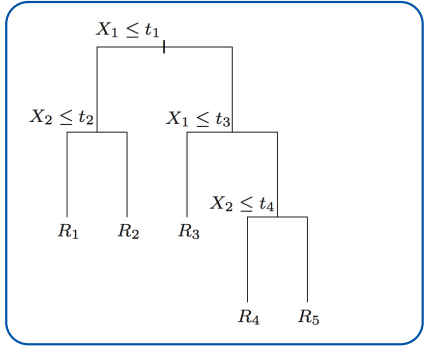


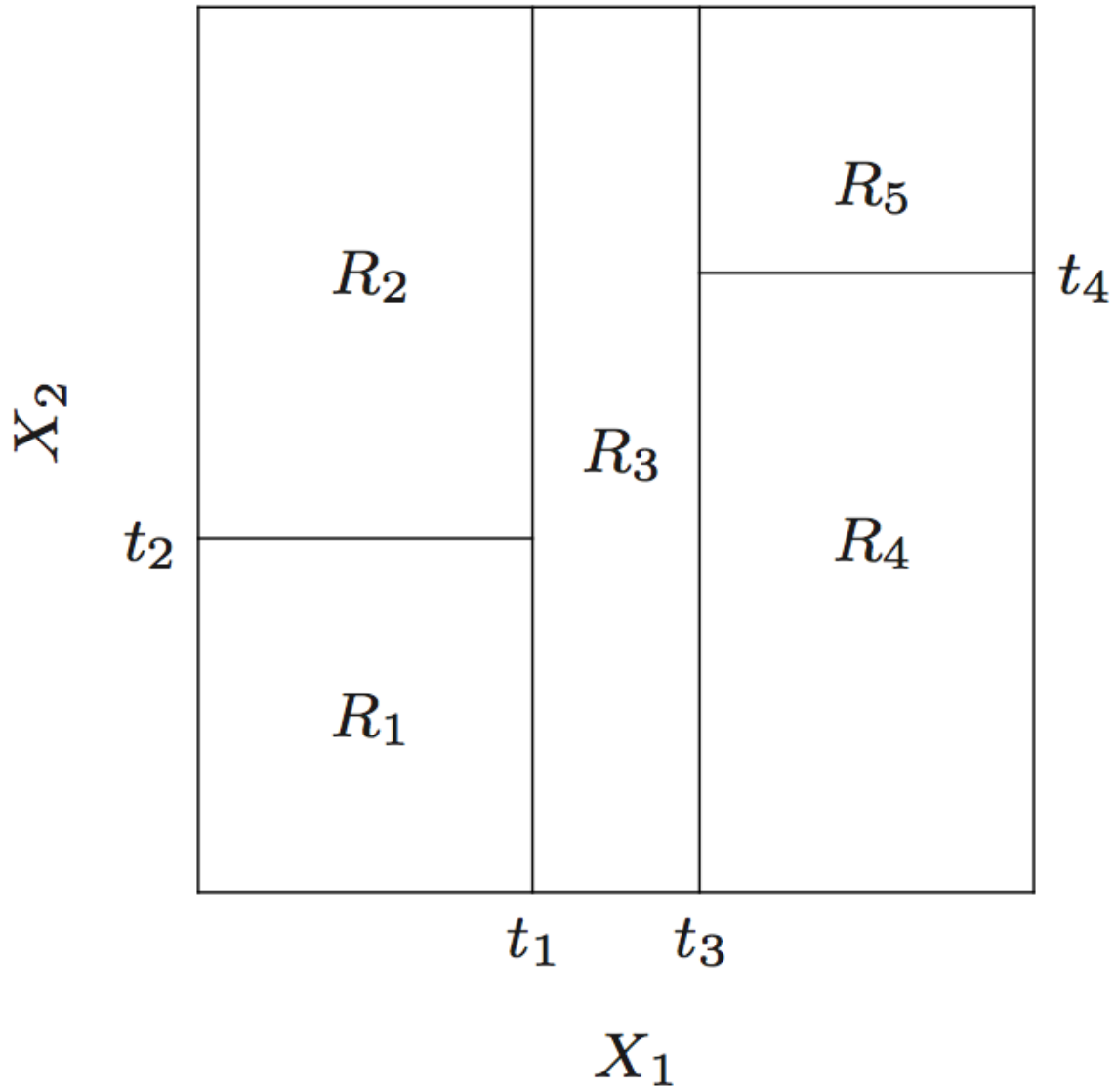
This is an astonishing example of ”women and children first”, as clearly spotted on the data.

Let us get back to theory and study a precise 2d example of an **regression tree**, Figures taken from Hastie et.al. (2009).

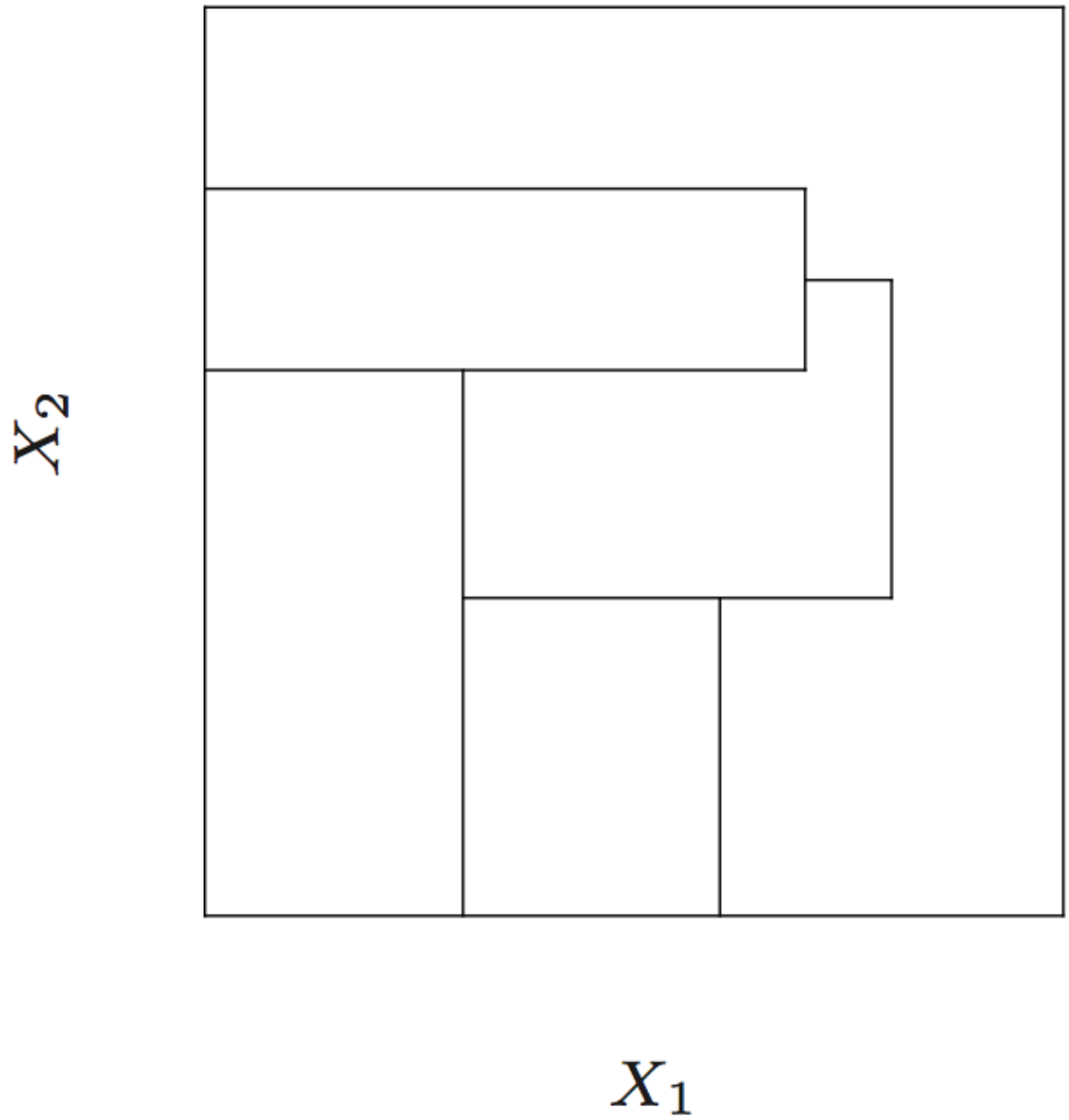


The associated partition is





Not all rectangular partitions can be reached by such a graph.



- Summarizing, we arrive at a model of the form

$$f(x) = \sum_{m=1}^{M} w_m \mathbb{1}_{\{x \in R_m\}}$$

  which fits in the ABM framework with $\phi(x, v_m) = \mathbb{1}_{\{x \in R_m\}}$.
- Note that the binary tree is recursive, which leads to easy and fast implementations. It moreover hast a high degree of interpretability.
- If we have data points $x_1, \ldots, x_n$ it is clear that a possible split will always be done at a data point (and not in between - why?). Thus the data automatically implies a maximum of possible regions $R_1, \ldots, R_M$. The main point is how to choose in a clever way a good performing sub-partition. We denote the possible split points in dimension j by $\mathcal{T}_j$, $j = 1, \ldots, d$.

- E.g. if we have data points $1, 3, -1, 3$ we obtain $\mathcal{T}_1 = \{-1, 1, 3\}$.
- Finding the pest partition is NP-complete, see Hyafil and Rivest (1976)[14]. Hence one needs to find efficient heuristics to construct nearly-optimal trees.
- We will study a **locally optimal approach**, but there are also other attempts, for example the evolutionary algorithm used in the `R`-package 'evtree'.
- Recall that we have (x_i, y_i), $i = 1, \dots, N$ data points at hand with x_i being d-dimensional, say. Suppose we have a partition into regions $R_1, \dots, R_M$ and we model

$$f(x) = \sum_{m=1}^{M} w_m \mathbb{1}_{\{x \in R_m\}}$$

(as above) and choose to minimize $\sum (y_i - f(x_i))^2$. Then clearly, the optimal $\hat{w}_m$ are given by the local averages

$$\hat{c}_m \propto \sum_{i=1}^{N} \mathbb{1}_{\{x_i \in R_m\}}.$$

[14] L. Hyafil und R. L. Rivest (1976). „Constructing optimal binary decision trees is NP-complete“. In: **Information processing letters** 5.1, S. 15–17.

- The next step is the splitting. We split one dimension in two pieces, i.e. consider the partition

$$R_1(j,s) := \{x \in \mathbb{R}^D : x_j \leq s\}, \quad R_2(j,s) := \{x \in \mathbb{R}^D : x_j > s\}$$

of $\mathbb{R}^D$, leading to the minimization problem

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right].$$

- As already remarked, $\hat{c}_i$, $i = 1,2$ can easily be found by averaging over the entries in R_1 (and R_2, respectively). Moreover, s can also be find very fast by browsing through $\mathscr{T}_j$, rendering the determination of (j^*, s^*) feasible.
- It remains to find a good **stopping criterion**.

- On first sight, one could guess that we run the algorithm until a new split does not increase the fitting quality substantially. However, the problem with trees is more complicated and a split at one step might only increase the fit in a small step but later will lead to just the right classification.
- Hence, on typically grows a **large** tree T_0 first, which is achieved by stopping at a certain minimal node size.
- Then, the large tree is **pruned**, and we describe **cost-complexity pruning**: denote the nodes of the tree T_0 by $R_1, \dots, R_M$ and let

$$N_m = |\{x_i \in R_m\}|$$

$$\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i$$

$$Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2.$$

The cost-complexity criterion is given by

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|.$$

- The tuning parameter α assesses the tradeoff between tree size and goodness of fit. It is typically estimated via crossvalidation leading to the optimal parameter $\hat{\alpha}$.
- For each α, there is a unique smallest subtree T_α minimizing $C_\alpha(T)$. This tree can be found by successively collapsing the internal nodes producing the smallest per-node increase in the cost-complexity criterion and proceeding until we obtain the single-node tree. In this finite sequence of subtrees T_α is contained (see Hastie et. al. p. 308 for comments and links to the literature).

- **Classification trees** can be treated similarly, but we would typically use a different cost function: consider the case of K classes $1,\dots,K$ and denote by

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} \mathbb{1}_{\{y_i = k\}}$$

  the fraction of observations in node R_m which have class k.

- The class estimator in node R_m is given by the class which has most members in this node, i.e.

$$\hat{k}_m = \arg\max_k \hat{p}_{mk}.$$

- Typical measures are misclassification error, the Gini index, or even entropy.

Be reminded that while being easy to interpret, trees do not come without disadvantages: they are unstable, the solution is typically not globally optimal and they are not smooth. For further discussions we refer to Hastie e.a. (2009)

Hierarchical mixtures of experts

- An alternative to the hard split is to put probabilities on each nodes. It is common to call the terminal nodes experts and we therefore arrive at a mixture of experts. Total probabilities are computed with Bayes' formula, compare Section 9.5 in Hastie e.a. (2009)

In the following lecture we will consider random forests, bagging and boosting of CARTs.

Structured Probabilistic Models

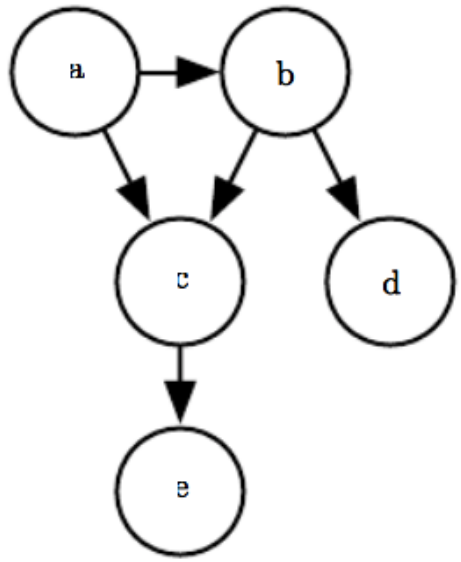
Machine Learning often involves high-dimensional probability distributions and numerous interaction between the dimensions need to be specified. One possibility to achieve this are structured probabilistic models and we discuss two possibilities.

1. Directed graphical models describe the **conditional factorization** v density is factorized as follows:

$$p(\boldsymbol{x}) = \prod_i p(x_i | x_j : j \in J_i)$$

with subsets $J_1, J_2, \ldots$ of the index set of $\boldsymbol{x}$.
The example (left[15]) describes the density

$$p(a,b,c,d,e) = p(a)p(b|a)p(c|a,b)p(d|b)p(e|c).$$



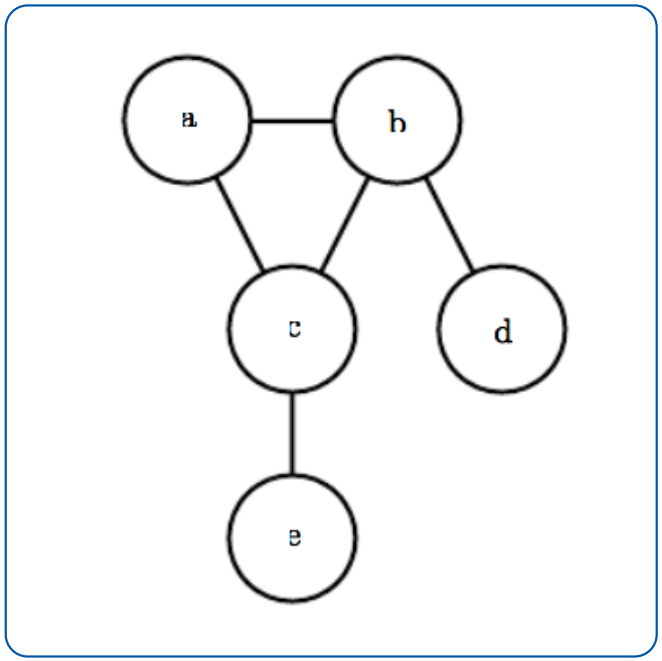
[15] Taken from Goodfellow e.a.(2016), Figure 3.7.

2 Undirected graphical models describe the **unconditional factorization** via undirected graphs. The density is factorized as follows:

$$p(\boldsymbol{x}) = \prod_i p(x_j : j \in J_i)$$

with subsets $J_1, J_2, \ldots$ of the index set of $\boldsymbol{x}$.
The example (left[16]) describes the density

$$p(a,b,c,d,e) \propto \phi^1(a,b,c)\phi^2(b,d)\phi^3(c,e).$$



[16] Taken from Goodfellow e.a.(2016), Figure 3.8.

Stochastic Gradient Descent

- Most of the algorithms we saw had to solve an optimization problem. A standard algorithm to solve these problems is the **gradient descent algorithm** ($\rightarrow$ blackboard - proposed by A. Cauchy in 1847).
- However, if the training set is too large, this becomes computationally very expensive: consider a cost functional of the type

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(x^i, y^i, \theta)$$

  with some differentiable loss function L.
- Gradient descent requires the computation of

$$\partial_\theta = \frac{1}{m} \sum_{i=1}^{m} \partial_\theta L x^i, y^i, \theta),$$

  leading to a computational cost of $O(m)$.
- As can be seen above, the gradient is actually an average - as for estimating an expectation it might be feasible to consider a small subsample of $\{1, \ldots, m\}$ leading to approximately the same result: this is the **stochastic gradient descent** approach.

- Select uniformly a subsample $\{i_1, \dots, i_n\} \subset \{1, \dots, m\}$ and compute the vector

$$g(\theta) := \frac{1}{n} \sum_{k=1}^{n} \partial_\theta L x^{i_k}, y^{i_k}, \theta).$$

- The algorithm computes the n-th approximation of the minimum θ_* by

$$\theta^n = \theta^{n-1} - \varepsilon g(\theta^{n-1})$$

  with stepsize (or learning rate) ε.

- It is quite surprising that a variant of the gradient descent arises here. Typically, gradient descent is a slow and quite unreliable procedure (see Wikipedia for some examples and R code). However, the stochastic gradient descent is very successfully applied in machine learning - it often finds a solution close to a minimum quick enough (see Goodfellow e.a. (2016) Section 5.9 for further comments).

Deep Learning

- "Deep"learning contrasts ßhallow"learning: such algorithms are for example linear regression, SVMs...: they have an input layer and an output layer. We have experienced the kernel trick: inputs may be transformed once before application of the algorithm.
- In deep learning there are one ore more **hidden layers** between input and output. Intuitively, at each layer we take the input, make a transformation and generate the output for the next layer.
- More formally, this corresponds to iteratively applied functions: a deep network is of the form

$$f(x) = f^n \circ \cdots \circ f^1(x) = f^n(f^{n-1}(\cdots f^2(f^1(x))\cdots)).$$

- f^k is called the k-the layer of the network.

A bit of the history

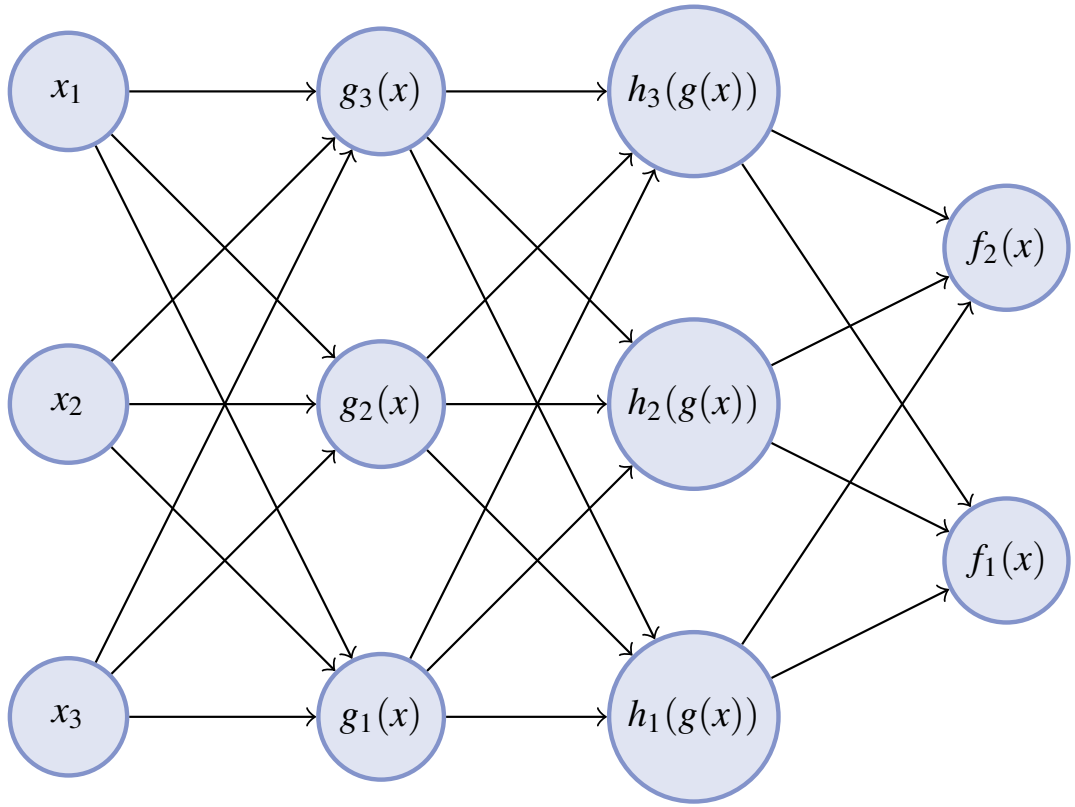
- The terminology of deep learning stems from early research on artifical intelligence and we dive shortly into this exciting subject. Two aspects were important at those times: to be inspired by the human brain and, on the other side, to try to understand the brain better through the construction of similar algorithms. Nowadays, we are more pragmatic and generalize the earlier ideas in several respects.
- A **neuron**[17] takes several inputs, say $x_1, \dots, x_n$ and gives

$$\mathbb{1}_{\{\sum_{i=1}^n w_i x_i > \theta\}}$$

  as an output - $w_i \in \mathbb{R}$ are several weights and $\theta \in \mathbb{R}$ is a threshold. This description of a neuron was given 1943 by W. McCulloch and W. Pitts.
- It was the idea of F. Rosenblatt in 1958 to introduce a simple neural net, called **perceptron** (after perception) which takes (possibly several) neurons as inputs and generates a more complex decision mechanism.

[17] See the wikipedia article on perceptrons.

Feed-forward neural networks



- Networks of this type are called either **feed-forward neural networks** or **multi-layer perceptrons** (when the nodes are actually neurons).
- $x_1, \ldots, x_n$ constitute the input layer. They give the input to all connected neurons in the **first layer**. There are 3 **hidden units** in our case and **two hidden layers**. The output is, as previously $f(x) = h(g(x))$.

- One problem which can not be achieved by a single layer perceptron is learning XOR ($\rightarrow$ Exercise).

The universal approximation theorem

- One important property of feed-forward neural networks is, that even in the single-layer case they can approximate arbitrary functions very well.
- The result is the so-called universal approximation theorem proved in Kurt Hornik (1991). „Approximation capabilities of multilayer feedforward networks“. In: **Neural networks** 4.2, S. 251–257.
- We study the mathematical details of this results.

- We consider special classes of feed-forward neural networks, which can be thought of a small generalization of multi-layer perceptrons: in each step, a neuron transforms the input vector x in an affine form to $a^\top x+b$ and sends the output $\phi(a^\top x+b)$. The outputs are weighted and summed up by each connected neuron.
- If there is only one hidden layer and only one output unit, we arrive at the output

$$\sum_{i=1}^{n} c_i \phi(a_i^\top x+b_i).$$

- Hence, the functions implemented by such a network with n hidden units is

$$\mathcal{N}^{(n)} = \mathcal{N}^{(n)}(\phi) = \left\{h : \mathbb{R}^d \to \mathbb{R} : h(x) = \sum_{i=1}^{n} c_i \phi(a_i^\top x+b_i)\right\}$$

  and for an arbitrary large number of units we set $\mathcal{N}(\phi) = \cup_n \mathcal{N}^{(n)}$.
- We consider functions in the $L^p(\mu)$-space with a finite measure μ. This are measurable functions $f : \mathbb{R}^d \to \mathbb{R}$, such that

$$\|f\|_p := \left(\int |f(x)|^p \mu(dx)\right)^{1/p} < \infty.$$

- A subset S of L^p is called **dense**, if for every $f \in L^p$ and $\varepsilon > 0$ there is a function $g \in S$, such that $\|f - g\|_p < \varepsilon$.

Theorem (Hornik (1991))

If ϕ is bounded and not constant, then $\mathcal{N}(\phi)$ is dense in $L^p(\mu)$ for any finite measure μ on $\mathbb{R}^d$.

This result also holds on the Banach space $C(K)$, K compact, with respect to the sup-norm. Further results are found in Hornik (1991).

The proof

- We will not discuss all the details of the proof, but have a look at certain components.
- First, observe that $\mathcal{N}$ is a **linear** subspace of $L^p(\mu)$ (elements are bounded!)
- If $\mathcal{N}$ is **not** dense, then the Hahn-Banach theorem yields the existence of a (non-zero) continuos linear function Λ such that Λ vanishes on $\mathcal{N}$. The goal is to construct a contradiction by this.
- Currently, Λ seems not to be so tractable, but duality of Hilbert spaces actually gives a very good description of such functionals. In particular, in our case we know that

$$\Lambda f = \int f g \mu(dx)$$

with some $g \in L^q(\mu)$ and $q = p/(p-1)$.

- Now we can write

$$\Lambda f = \int f d\mu'$$

with (by Hölders inequality) some finite (but possibly signed) measure μ'.

- As Λ vanishes on $\mathcal{N}$,

$$\int \phi(a^\top x + b)\mu'(dx) = 0$$

for all $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$.

We have

$$\int \phi(a^\top x + b)\mu'(dx) = 0 \tag{6}$$

for all $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$. It is clear that this can not hold for any function ϕ. Hornik was able to show, that if ϕ is bounded and not constant, then (6) will not hold for any finite signed measure μ'.

- A first step is the transformation

  $$\int \phi(a^\top x + b)\mu'(dx) = \int \phi(t + b)\mu_a(dt)$$

  with the projection measure $\mu_a(B) = \mu'(x \in \mathbb{R}^d : a\top x \in B)$.
- One proceeds further and arrives at

  $$\int \phi(t)h(\alpha t + \beta)dt.$$

  Now one can apply Fourier transform and arrives at $\mu_a = 0$ for all $a \in \mathbb{R}^d$.

Rate of convergence

- It was moreover shown in Andrew R Barron (1994). „Approximation and estimation bounds for artificial neural networks“. In: **Machine Learning** 14.1, S. 115–133 that the mean integrated squared error between (single-layer) network and target function is bounded by

$$O(1/n) + O(nd/N) \log N.$$

- Here, n is the number of nodes, d is the input dimension and N is the number of training observations.
- However, the result only considers a single layer and the analysis is therefore limited.

Gradient-based learning

- A remarkable difference between the neural networks and previously seen algorithms is that neural networks have typically **non-convex** target functions.
- Therefore, gradient methods (possibly stochastic ones) come into play in contrast to linear-quadratic sovlers we saw before. Let us study some common cost functions.

Learning with maximum-likelihood

- Once we have a probabilistic model, we automatically have a log-likelihood function which can solve as loss function. The advantage of this approach is that we do not need to specify an additional criterion.

- What kind of distance to we minimize when we look at maximum-likelihood ?
- Although more complicated schemes may be possible, think of i.i.d. observations from the density (or probability function) $p(x,\theta)$. Then, the log-likelihood is

$$l(x,\theta) = \sum_{i=1}^{n} \log p(x^i,\theta).$$

- Of course, this can be viewed as

$$l(x,\theta) = m\frac{1}{m}\sum_{i=1}^{n} \log p(x^i,\theta) = mE_x^n[\log p(X,\theta)]$$

  where E_x^n is the empirical distribution at the observation $x = (x^1,\dots,x^n)$.
- Then, we actually minimize the Kullback-Leibler divergence[18] given by

$$D_{KL}(p_x^n, p_\theta) := E_x^n[\log p_x^n(X) - \log p_\theta(X)]$$

  where p^n is the epmirical distribution at the observation x and p is our model density (note that the first term does not depend on θ).

[18] $D_{KL}(P,Q) = E_P[\log \frac{dP}{dQ}(X)]$.

Conditional maximum-likelihood

- For **supervised learning** we additionally have supervision data, such that we actually observe (x^i, y^i), $i = 1, \dots, n$. Conditional maximum likelihood then maximizes (in the i.i.d.-situation)

$$\sum_{i=1}^{n} \log p(y^i | x^i, \theta).$$

  Here, $p(y|x, \theta)$ is the density (or probability function) of y given x and parameter θ.

- Hence, a possible **cost function** for supervised learning is given by

$$J(\theta) = -\sum_{i=1}^{n} \log p(y^i | x^i, \theta)$$

Rectified linear units

- Inspired from perceptrons, our hidden units will typically produce an output of the form
$$\phi(a^\top x + b).$$
This generalizes the neuron where $\phi(y) = \mathbb{1}_{\{y>\theta\}}$ and it is an important question which ϕ is most suited ? Typically one would think of sigmoids, probability transforms or logit link functions. It is quite surprising, that the most applied **activation function** is the **rectified linear unit**
$$\phi(y) = \max\{0, y\}.$$
- However, there are good reasons for this: in this case, ϕ is piecewise linear and the second derivative vanishes (a.s.). Hence the gradient direction is far more useful than in other cases !

Some generalizations are available:

- $$\max\{0, y_i\} + \alpha_i \min\{0, y_i\}, \qquad i = 1, \ldots, d.$$
- A **leaky ReLU** fixes a small α while a parametric ReLU learns α during the procedure.
- A further treatment are maxout units (skipped here - see Goodfellow, Chapter 6.3.1.)

Forward propagation

- In a feedforward neural network to produce an output $\hat{y}$ from an input x information flows forward through the network
- This is called forward propagation
- During training, forward propagation produces a scalar cost $J(\theta)$

Forward propagation algorithm for a typical deep neural net

- Require: Network depth, l
- Require: $W^{(i)}$, $i \in \{1,...,l\}$, the weight matrices of the model
- Require: $b^{(i)}$, $i \in \{1,...,l\}$, the bias parameters of the model
- Require: x, the input to process
- Require: y, the target output
- set $h^{(0)} = x$
- for $k = 1,\ldots,l$ do:
  - $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  - $h^{(k)} = f(a^{(k)})$
- at the end of the loop set:
- $\hat{y} = h^{(l)}$
- $J(\theta) = L(\hat{y}, y) + \lambda \Omega(\theta)$,where θ is $(W^{(i)}, b^{(i)})$ $i \in \{1,...,l\}$

Backpropagation

- The back-propagation algorithm allows the information from the cost to flow backwards through the network, in order to compute the gradient

- The term back-propagation is not the whole learning algorithm
- Back-propagation is only a method to compute the gradient
- Another algorithm, e.g. stochastic gradient descent, is used to perform learning using this gradient.

- Computing an analytical expression for the gradient is straightforward
- Numerically evaluating such an expression can be computationally expensive
- The back-propagation algorithm does so using a simple and inexpensive procedure, that relates to the chain rule.

$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$

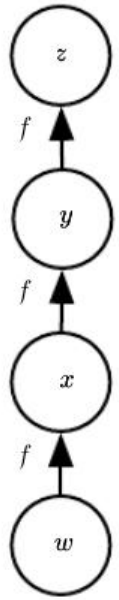


Figure from Goodfellow 2016

Backward propagation algorithm for a typical deep neural net

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, y)$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

end for

this is Algorithm 6.4 in Goodfellow 2016

- The gradients on weights and biases can be used for a stochastic gradient update
- Symbol-to-number differentiation (Torch, Caffe): Use a set of numerical values for the inputs and return a set of numerical values describing the gradient at those input values
- Symbol-to-symbol differentiation (Theano,Tensorflow): Add additional nodes to the graph that provide a symbolic description of the desired derivatives.
- Because the derivatives are just another computational graph, it is possible to run back-propagation again, to obtain higher derivatives.

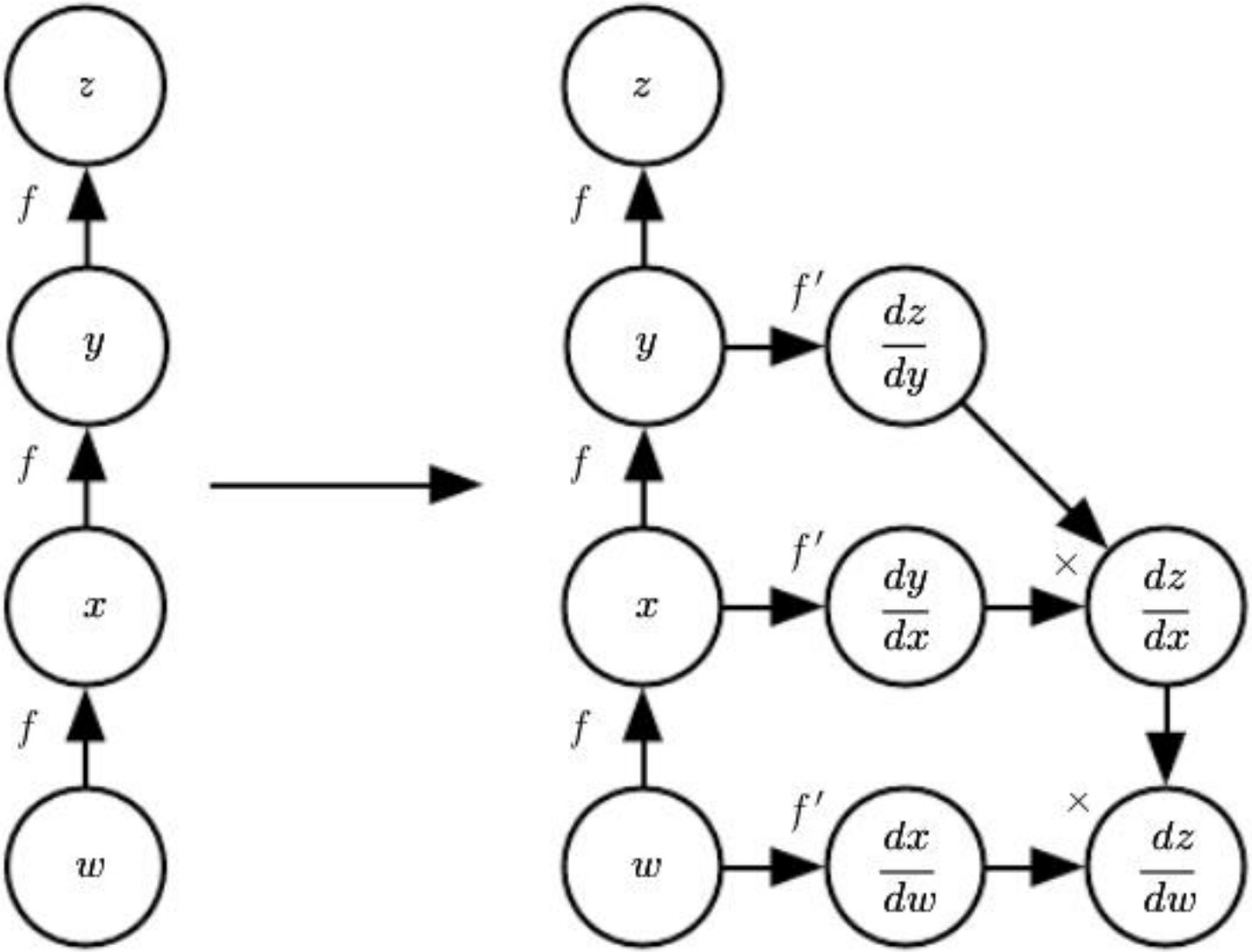


Figure from Goodfellow 2016

Regularization in Neural Networks

- regularization is a way to overcome underfitting, overfitting issues by trading variance of the prediction error against bias.
- $\mathbb{E}[L(\hat{y},y)] = \text{Irreducible Error} + \text{Bias}^2 + \text{Variance}$ (excercise)
- regularization is a modification to a learning algorithm that is intended to reduce its generalization error but not its training error.
- we have already seen bagging as a regularization method
- In the context of deep learning, most regularization strategies are based on regularizing estimators, by adding a parameter norm penalty $\Omega(\theta)$ to J

$$J(\theta;X,y) + \lambda\Omega(\theta)$$

weight decay

- weight decay refers to the L^2 penalty.
- also known as ridge regression
- if we do not punish the bias b the objective function for weight decay is given by

$$\tilde{J}(w;X,y) = \frac{\lambda}{2} w^T w + J(w;X,y)$$

- this means in a single gradient update step the update changes to

$$w \leftarrow (1 - \varepsilon\lambda)w - \varepsilon\nabla_w J(w;X,y)$$

- the addition of the weight decay term has modified the learning rule to shrink the weight vector on each step

- we make a quadratic approximation to the objective function in the neighborhood of the value w^*, the optimal weights where unregularized training cost is minimal

$$\hat{J}(w) = J(w^*) + \frac{1}{2}(w - w*)^T H(w - w^*)$$

- where H is the Hessian matrix of J with respect to w evaluated at w^*
- the minimum of the regularized version of $\tilde{J}$ is at

$$\tilde{w} = (H + \lambda I)^{-1} H w^*$$

- If we decompose $H = Q\Lambda Q^T$ into a diagonal matrix Λ and an orthonormal basis of eigenvectors Q we get

$$\tilde{w} = Q(\Lambda + \lambda I)^{-1} \Lambda Q^T w^*$$

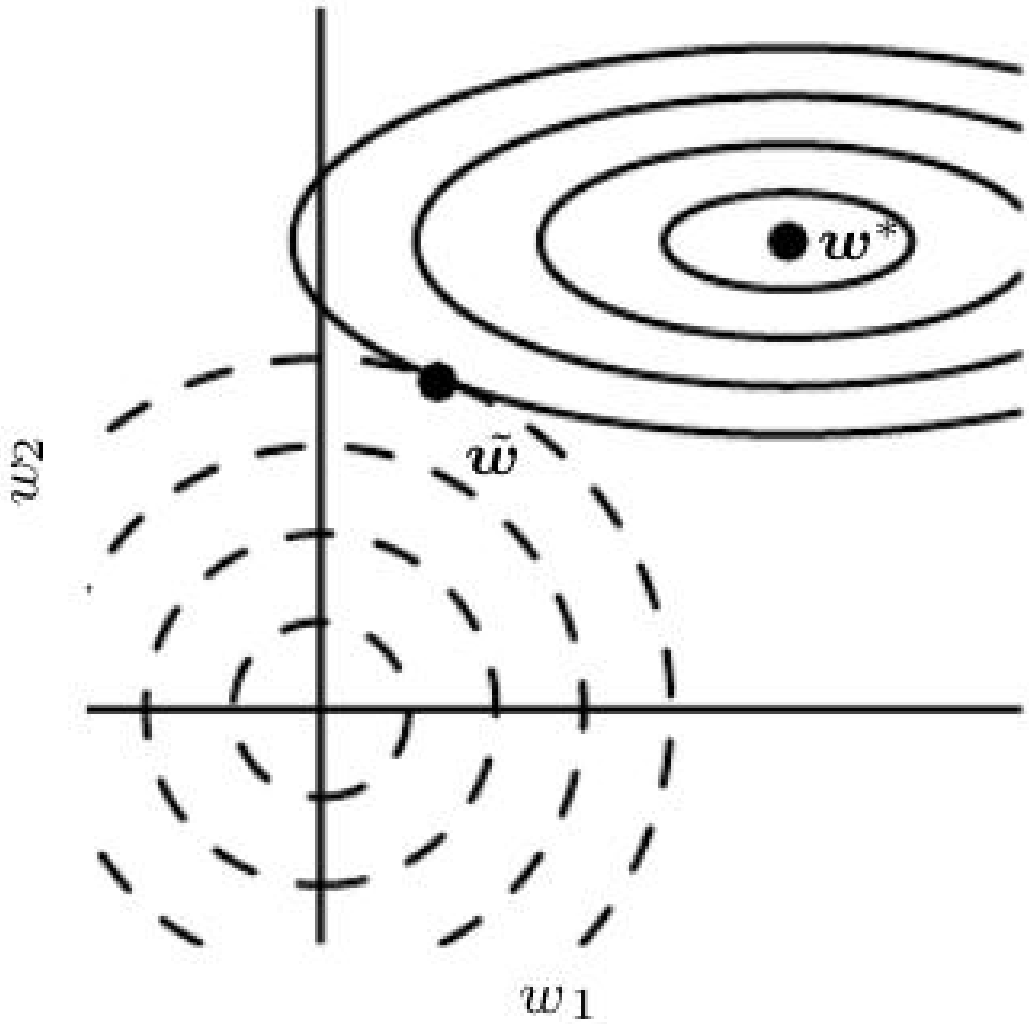


Figure from Goodfellow 2016

- In comparison to L^2 regularization, L^1 regularization results in a solution that is more sparse.
- Sparsity in this context refers to the fact that some weights have an optimal value of zero.

Let us have a look at the learning procedure at `playground.tensorflow.org`

Dynamic Approximate Programming

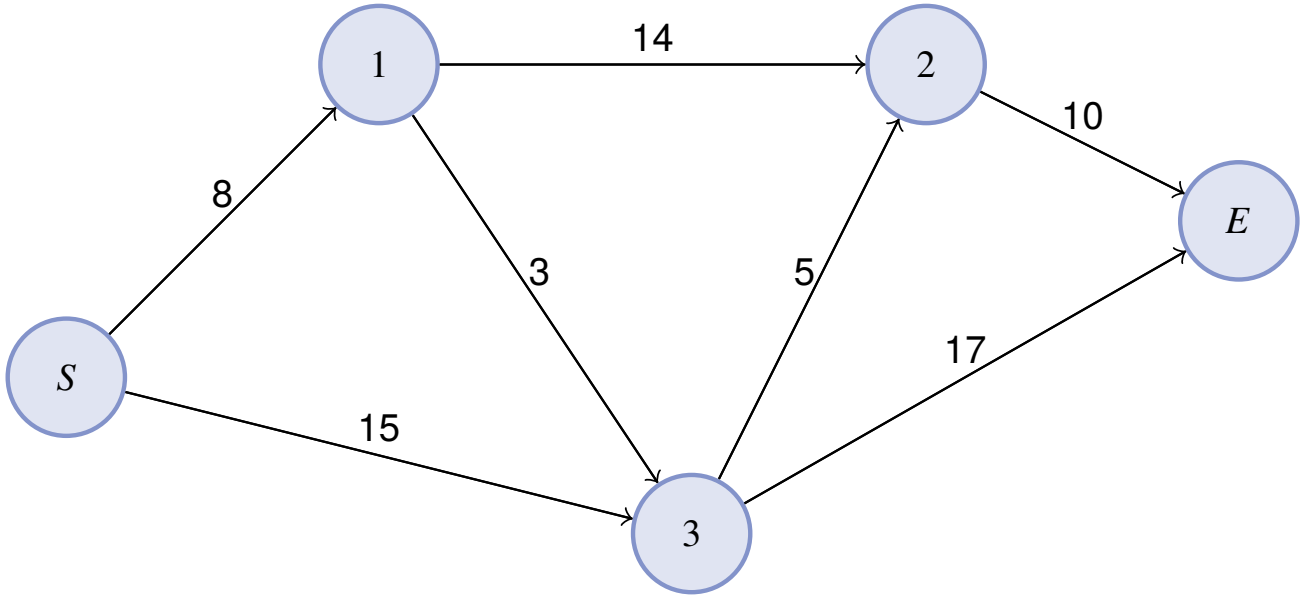
- From now on, we study the field of dynamic approximate programming (ADP) following Powell (2011)[19].
- As we already learned, there are many dialects in this field and we treat them here. This includes **reinforcment learning**, and a classic reference is Sutton & Barto[20]. For further references consider Powell (2011).
- The terminology "reinforcementßtems from behavioural sciences. A positive reinforcer is something that increases the probability of a preceding response (in contrast to a negative reinforcer, like an electronic shock), see also Watkins (1989).
- Examples are: moving a robot, investing in stocks, playing chess or go.
- The system contains four main elements: a **policy**, a **reward funciton**, a **value function** and (optional) a **model** of the environment.

[19] Warren B Powell (2011). **Approximate Dynamic Programming: Solving the curses of dimensionality**. Bd. 703. John Wiley & Sons.

[20] R. S. Sutton und A. G. Barto (1998). **Reinforcement Learning : An Introduction**. MIT Press.

An Example

Let us start with a simple example.



It is our goal to find the shortest path from Start to End.

- By $\mathscr{I}$ we denote the set of intersections $(S,1,\ldots,E)$,
- if we are at intersection i we can go to $j \in \mathscr{I}_i^+$ at cost c_{ij},
- we start at S and end in E. Denote

$$v_i := \text{cost from } i \text{ to } E$$

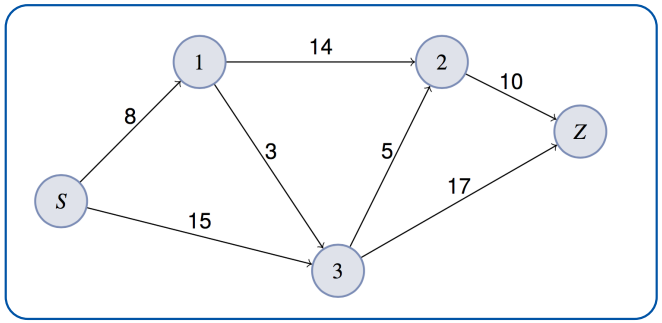
and we could iterate

$$v_i \leftarrow \min\left\{v_i, \min_{j\in\mathscr{I}_i}(c_{ij}+v_j)\right\}, \quad v_i \in \mathscr{I}$$

and stop if the iteration does not change.

Iteration	S	1	2	3	E
1	∞	∞	∞	∞	0
2	∞	∞	10	15	0
3	30	18	10	15	0
4	26	18	10	15	0

- What is an efficient algorithm for solving this problem ?



- This is a **shortest-path problem**. Let us introduce some notation for this. At time t, we start from a state S_t and can choose **action** a_t which leads to the transition to state S_{t+1} given by the **transition function** S, s.t.
$$S_{t+1} = S(S_t, a_t)$$
- Aditionally there is a **reward**, denoted by $C_t(S_t, a_t)$ and we define the value of being in state S_t by
$$V_t(S_t) = \max_{a_t} \{C_t(S_t, a_t) + V_{t+1}(S_{t+1})\}, \qquad S_t \in \mathscr{S}_t,$$
cS_t denoting the possible states at time t.
- Let us visit some further examples.

Gambling

- Consider a gambler who plays T rounds, on an i.i.d. $(W_t)_{t=1,\dots,T}$ game with probability $p = \mathbb{P}(W_t = 1) > 1 - p$ of winning. We want to maximize $\mathbb{E}[\log(S_T)]$. It can be shown that it is optimal to proceed backwards in time using conditional expectations (this is dynamic programming)!
- Here, a_t is the amount he bets at t and we require $a_t \leq S_{t-1}$. Then,

$$S_t = S_{t-1} + a_t W_t - a_t(1 - W_t).$$

- The value at time t, given his stock is in state S_t is

$$V_t(S_t) = \max_{0 \leq a_{t+1} \leq S_t} \mathbb{E}\left[V_{t+1}(S_{t+1}) | S_t\right].$$

- Now we proceed backwards. Clearly,

$$\begin{aligned} V_T(s) &= \log s \\ V_{T-1}(s) &= \max_{0\le a\le s} \mathbb{E}\big[V_T(s+aW_T - a(1-W_T))|S_{T-1}=s\big] \\ &= \max_{0\le a\le s} \Big(p\log(s+a) + (1-p)\log(s-a)\Big). \end{aligned}$$

- The maximum is attained for $a^* = (2p-1)s$ and $V_{T-1}(s) = \log(s) + K$, with costant $K = p\log(2p) + (1-p)\log(2(1-p))$. Backward in time we obtain

$$V_t(s) = \log S_t + K_t,$$

with an explicit constant K_t. Our **optimal policy** is

$$a_t = (2p-1)S_{t-1}.$$

The bandit problem

- When the distribution of the game is not known, one has to acquire information, and the classical example is the bandit problem. Consider a gambler who can choose betwee K machines.
- The probability of winning might be different and are **unknown** to us.
- A trade-off arises between playing only the optimal machine or trying other machines with (estimated) lower probability for minimizing the variance which is one-to-one to learning better their true probability.
- For a nite treatment, consider for example Richard Weber (1992). „On the Gittins Index for Multiarmed Bandits“. In: **Ann. Appl. Probab.** 2.4, S. 1024–1033.

Markov decision problems

- We give a short introduction into the field[21]. Assume that the state space $\mathscr{S}$ if **finite**.
- We have a set $\mathscr{A}_t(s)$ of possible actions at time t when the system is in state s. An action at t is a measurable mapping a_t such that $a_t(s) \in \mathscr{A}_t(s)$ for all $s \in \mathscr{S}$.
- A **policy** is a collection of actions $\pi = (a_0, \dots, a_{T-1})$. We assume that the set of policies is non-empty.
- The dynamics of the model is specified via the (conditional) transition matrix

$$(p_t(s_{t+1}|s_t, a_t))_{s_{t+1}, s_t \in \mathscr{S}}$$

  specifying $\mathbb{P}(S_{t+1} = s_{t+1}|S_t = s_t, a_t) = p_t(s_{t+1}|s_t, a_t)$.
- Hence, the dynamics and with it the probability for evaluation depends on π. We denote

$$\mathbb{P}^\pi_{t,s}(\cdot) := \mathbb{P}^\pi(\cdot|S_t = s)$$

  and by $\mathbb{E}^\pi_{t,s}$ the associated expectation.

[21] See N. Bäuerle und U. Rieder (2011). **Markov decision processes with applications to finance**. for details and further information.

- Our aim is to **maximize** the contribution given by the functions $C_t(s,a)$ where $C_T(s,a) = C_T(s)$ does not depend on a. We additionally assume that the contribution is sufficiently integrable.
- Our goal is to aim at

$$\sup_{\pi} \mathbb{E}^{\pi} \left[\sum_{0=1}^{T} C_t(S_t, a_t) \right].$$

  For example, we could consider $C_t(s,a) = \gamma^t C(s,a)$ with possible discounting factor $\gamma > 0$.

The Bellman Equation

- The key to dynamic programming is that in our set-up, allowing the policy to depend on the full history does not improve the maximal expected reward, see Theorem 2.2.3. in Bäuerle&Rieder (2011).
- We define the **value function** by

$$V_t(s) = \sup_{\pi} \mathbb{E}^{\pi}_{t,s}\left[\sum_{s=t}^{T} C_t(S_t, a_t)\right].$$

Remark

In general V_t need not be measurable which causes a number of delicate problems, see D. P. Bertsekas und S. Shreve (2004). **Stochastic optimal control: the discrete-time case**. *for a detailed treatment. The reason can be traced back to the fact that a projection of a Borel set need not be Borel (which leads to the fruitful notion of analytic sets, however).*

- Define

$$C_t^*(s) := \sup_{a_t \in \mathcal{A}_t} \left(C_t(s, a_t) + \mathbb{E}\left[V_{t+1}(S_{t+1}) | S_t = s, a_t \right] \right) \tag{7}$$

  Recall, that S_{t+1} also depends on $a_t = a_t(s)$ (which we suppress in the notation).

- The optimal policy can be computed backward by **reward iteration**. Let a_t^* be a maximizing policy, that is a_t^* achieves C_t^* in Equation (7).
- One can now show that the **Bellman equation** holds, i.e.

$$V_t(s) = C_t^*(s) \quad t = 0, \dots, T.$$

- Under an additional (mild) structural assumption, one may verify that there always exist optimal policies π^* which can be obtained by maximizing the value function in each period (Theorem 2.3.8. in Bäuerle Rieder).

Algorithm

Step 0 Initialize by the terminal condition $V_T(S_T)$ and set $t = T-1$

Step 1 Compute

$$V_t(s) = \sup_{a_t \in \mathscr{A}_t} \left(C_t(s,a_t) + \mathbb{E}\left[V_{t+1}(S_{t+1}) | S_t = s, a_t \right] \right)$$

for all $s \in \mathscr{S}$

Step 2 Decrement t and repeat Step 1 until $t = 0$

Infinite-time-horizon

- For this case several algorithms exist, to name **value iteration** and **policy iteration** which will not be discussed here, see Powell Section 3.3. ff.
- For more mathematical details (and there are many!) we refer to Powell, Bäuerle&Rieder and the excellent source Bertsekas&Shreve.

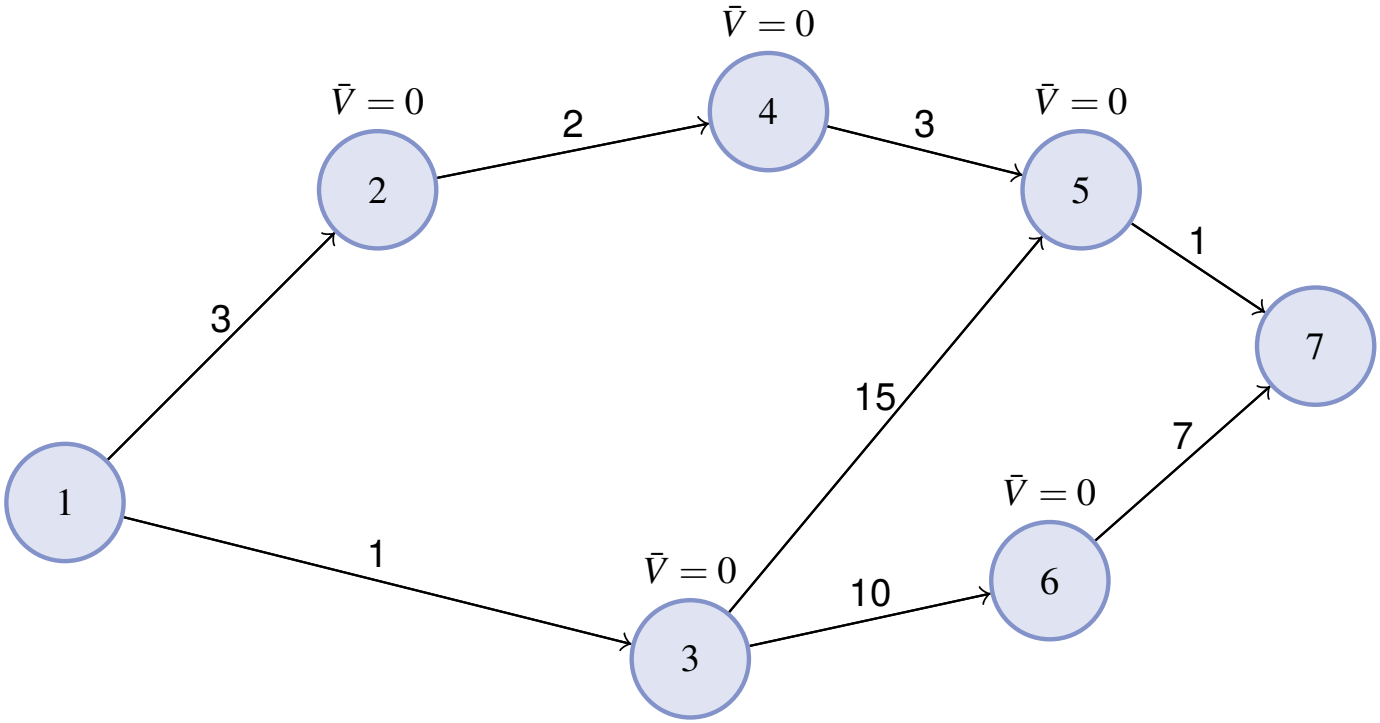
Approximate dynamic programming (ADP)

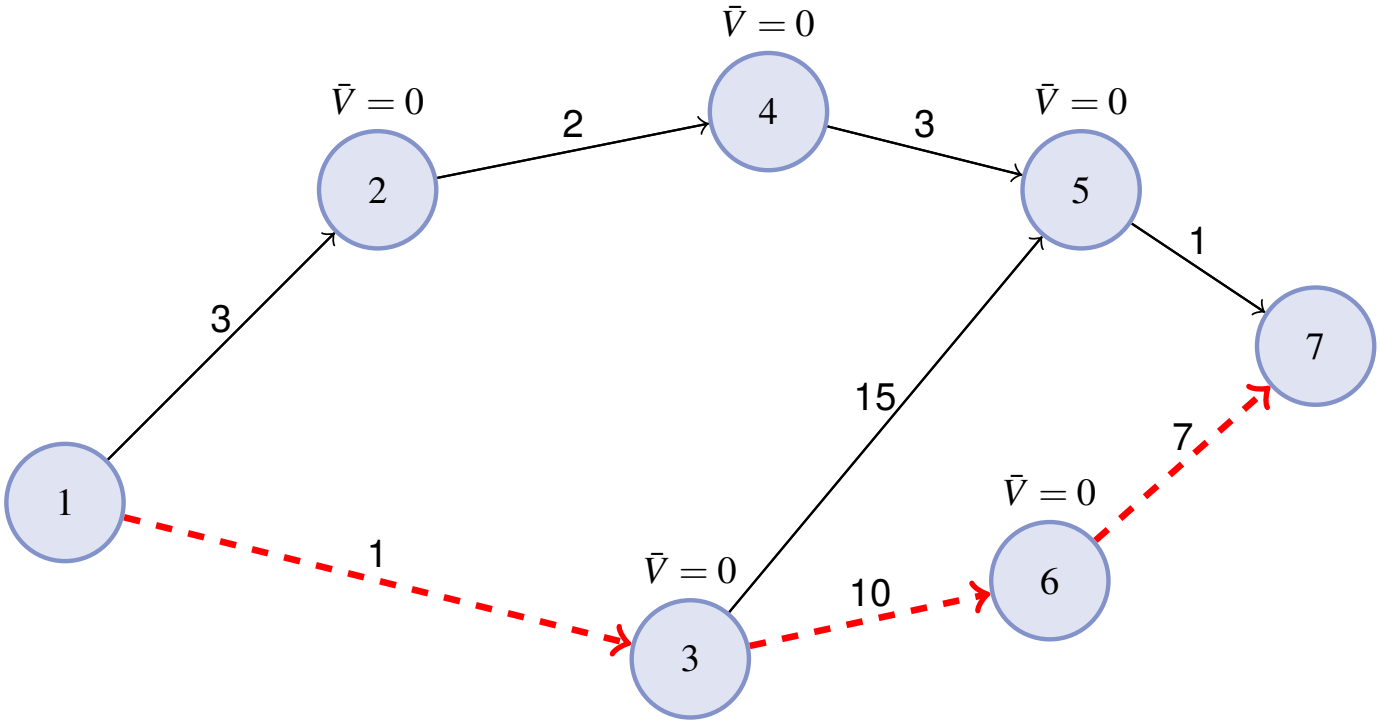
- While we introduce a nice theory beforehand, the core equation

$$\sup_{\pi} \mathbb{E}^{\pi}\left[\sum_{0=1}^{T} C_t(S_t, a_t)\right]$$

  my be intractable even for very small problems.
- ADP now offers a powerful set of strategies to solve these problems approximately.
- The idea stems from the 1950's while a lot of the core work was done in the 80's and 90's.
- We have the problem of curse of dimensionality in **state space**, **outcome space** and **action space**.

We illustrate this by an example: we approximate the value function by the function $\bar{V}$ which we update iteratively.



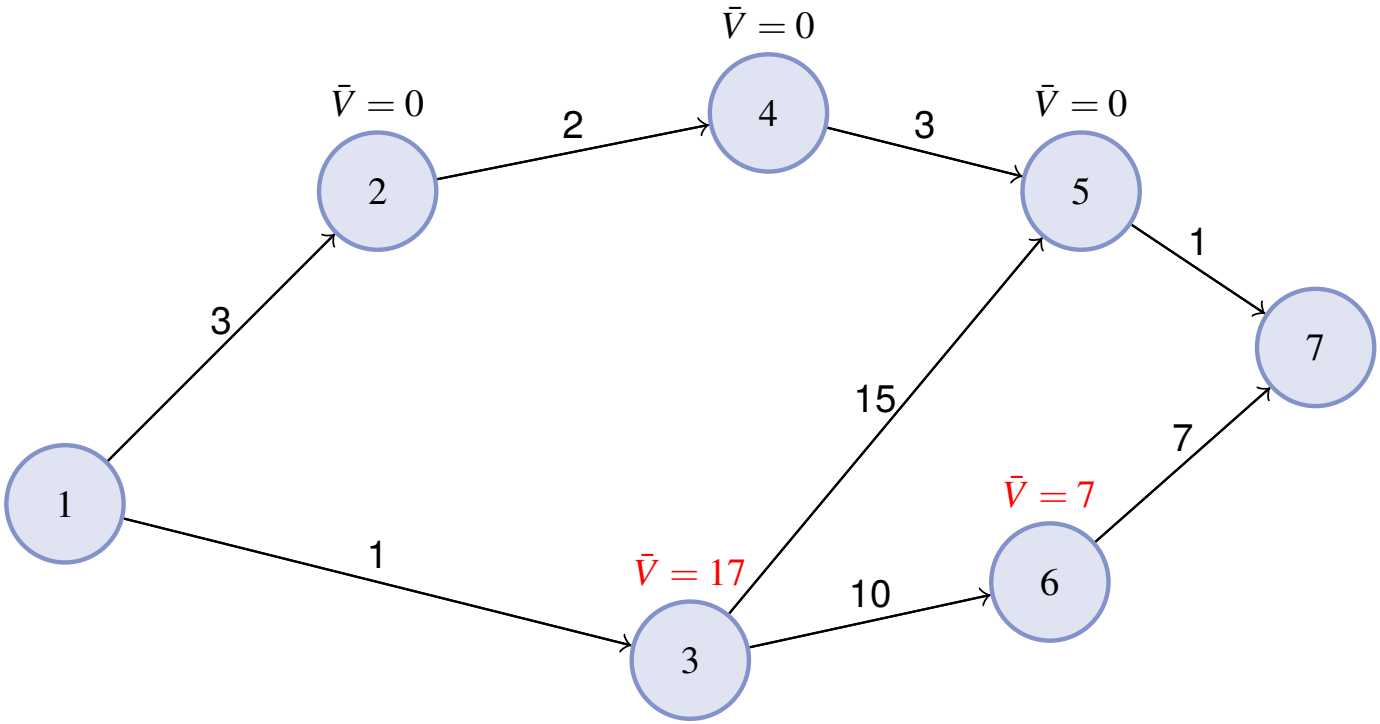


The approximation of the value function is **optimistic**: $\bar{V}=0$ at all states. We start (forward !) in node 1 and choose the node where

$$c_{ij}+\bar{V}(j)$$

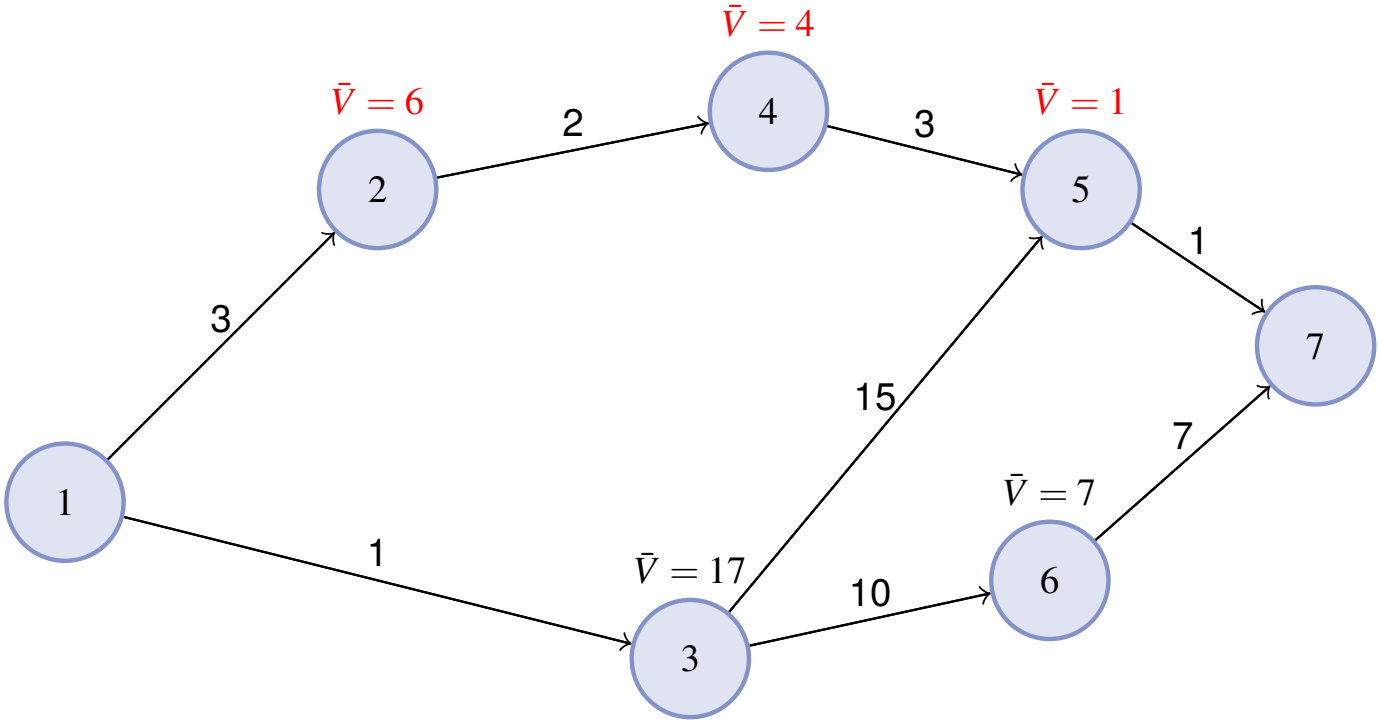
is minimal. This means we choose $1-3-6-7$ and update (!) accordingly:

$$\bar{V}(3)=17, \quad \bar{V}(6)=7.$$



Now (because we initiated optimistically) we take another round and go $1-2-4-5-7$, again updating the weights.

Now (because we initiated optimistically) we take another round and go $1-2-4-5-7$, again updating the weights.



We have found the optimal path !

The above example (still a deterministic one) shows a number of interesting features:

- We proceed forward - which is suboptimal, but repeat until we have found an optimal (or close-to-optimal) solution.
- The value function is approximated.
- The choice of the initial $\bar{V}$ can make us explorative or less explorative - it will become important further on to have this in mind.
- Typical examples are the learning of a robot (for example to stop a ball).

The basic idea

- There are many variants of ADP - here we look at the basic idea: we proceed forward and approximate $\bar{V}$ iteratively.
- We start with an initial approximation

$$\bar{V}_t^0(s), \qquad \text{for all } t = 0, \dots, T-1,\ s \in \mathscr{S}.$$

- Then we proceed iteratively.

Basic ADP algorithm

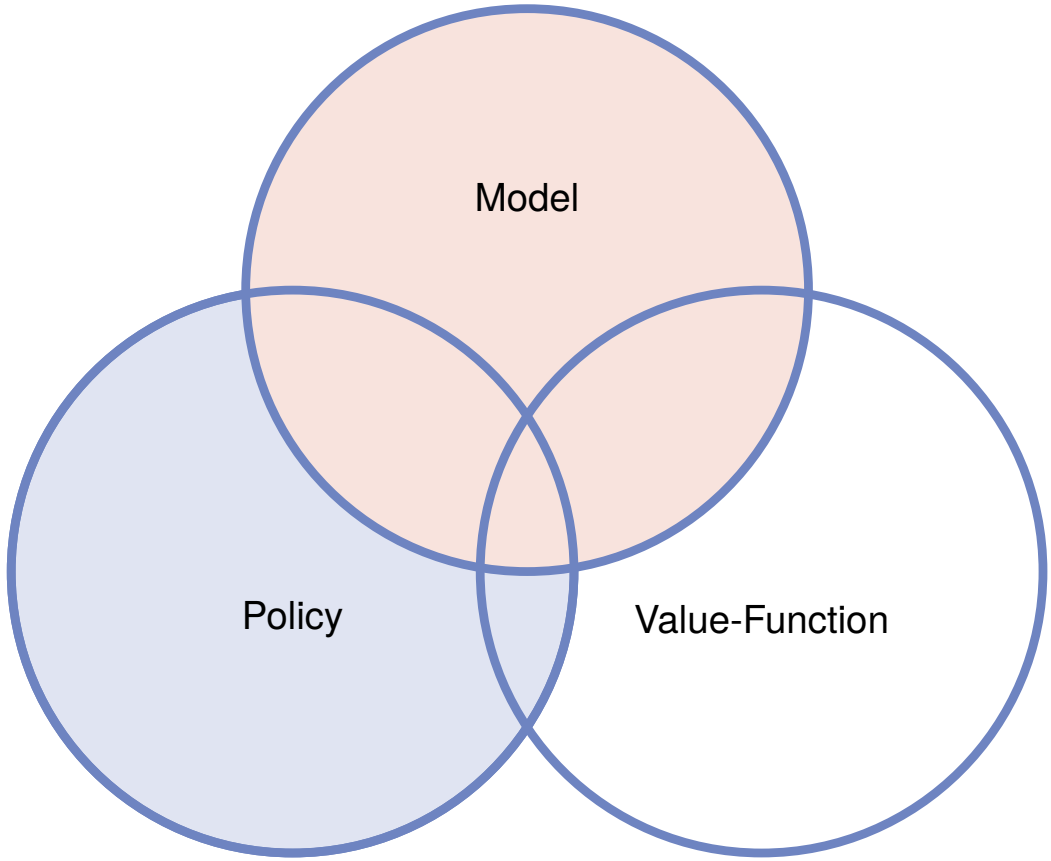
Starting from $\bar{V}^{n-1}$ we proceed as follows:

1. simulate a path $S(\omega) =: (s_0, s_1, \dots, s_T)$.
2. at $t = 0$ we compute
$$\hat{v}_0^n = \hat{v}_0^n(\omega) = \max_{a \in \mathscr{A}_0(s_0)} \left\{ C_0(s_0, a) + \mathbb{E}[\bar{V}_1^{n-1}(S_1) | S_0 = s_0] \right\}.$$
3. Thereafter, we solve
$$\hat{v}_t^n = \max_{a \in \mathscr{A}_t(s_t)} \left\{ C_t(s_t, a) + \mathbb{E}[\bar{V}_{t+1}^{n-1}(S_{t+1}) | S_t = s_t] \right\}$$
and continue iteratively until $t = T$.
4. Finally, we update $\bar{V}$ by letting
$$\bar{V}_t^n(s) = \begin{cases} \hat{v}_t^n, & \text{if } s = s_t \\ \bar{V}_t^{n-1}(s) & \text{otherwise.} \end{cases}$$

- Note that we still need to be able to compute the expectation (from the transition probabilities). This might be difficult (and for example for a robo running around in the world, infeasible and unwanted)
- We only update $\bar{V}$ for those states we visit. We therefore need to make sure that we are explorative enough to visit sufficiently many states
- We might get caught in a circle and a convergence proof is lacking.

Overview

We have three ingredients: model, policy, value-function. Consequently, we have associated groups: model-free / model-based, value-based and policy-based.



- Due to the supremum, the Bellman equation is non-linear and a variety of methods for solving it exist:
- Value iteration
- Policy iteration
- Q-Learning
- SARSA

We start with Q-Learning, value and policy iteration typically apply to ∞-time horizon problems, but will be discussed shortly as well.

Q-Learning

A first model-free approach is the following:

- The Q-learning ADP was proposed in Watkins[22] (an interesting read).
- The idea is again to approximate the value function. This time we look at the function $Q(s,a)$ which gives the value of action a when being in state s, i.e. we are looking for

$$Q : S \times a$$

- This gives an immediate hand on the optimal policy, $a^*(s) = \arg\max_a Q(s,a)$.
- Again, we proceed iteratively. The assumption we make is that once we chooce action a we observe the contribution $\hat{C}(S_t, a)$ and the next state S_{t+1}.
- We call an alogrithm **greedy**, if it bases its decision on the value function.
- Assume we are only interested in $V_0(\cdot)$.

[22] Christopher John Cornish Hellaby Watkins (1989). „Learning from delayed rewards“. Diss. King's College, Cambridge.

Q-Learning

- Start with an initial $\bar{Q}^0$.
- Suppose we are in step n and at position S^n. We choose action a^n greedy, i.e.

$$a^n := \arg\max_{a \in \mathcal{A}(S^n)} \bar{Q}^{n-1}(S^n, a).$$

- We observe $\hat{C}(S^n, a^n)$ and S^{n+1}.
- Compute

$$\hat{q}^n = \hat{C}(S^n, a^n) + \gamma \bar{Q}^{n-1}(S^{n+1}, a^n)$$

and update with **stepsize** or **learning rate** α_n:

$$\begin{aligned}\bar{Q}^n(S^n, a^n) &= (1-\alpha_n)\bar{Q}^{n-1}(S^n, a^n) + \alpha_{n-1}\hat{q}^n \\ &= \bar{Q}^{n-1}(S^n, a^n) + \alpha_n \left(\hat{C}(S^n, a^n) + \gamma \bar{Q}^{n-1}(S^{n+1}, a^n) - \bar{Q}^{n-1}(S^n, a^n) \right).\end{aligned}$$

Note that no expectation needs to be taken nor any model comes into play.

- A simple implementation just stores the values of Q in a table, which might be less efficient if the spaces get bigger.
- One possibility to solve this issue is to use an artificial network to learn this function (by the universal approximation theorem this is always possible), leading to ”deep reinforcement learning” schemes, as proposed by DeepMind for playing Atari Games.
- Other variants concern speeding up the rates of convergence, as in its current form Q-Learning can be quite slow.

Implementations

- A variety of implementations are available:
- Car steering
  `http://blog.nycdatascience.com/student-works/capstone/reinforcement-learning-car/`
- a nice blog by Andrej Karpathy about the Atari game pong
  `http://karpathy.github.io/2016/05/31/rl/`
- The R package ReinforcementLearning from N Pröllochs (Freiburg!)[23]

[23] `https://github.com/nproellochs/ReinforcementLearning`

SARSA

- SARSA is an algorithm to **estimate** the value of a fixed policy, which will be important, for example, for policy iteration.
- The name stems from the following acronym: suppose we are in state s, take action a, observe afterwards the reward r, go to state s' and take action a'.
- More precisely, we estimate the value of the policy π by iterating (infinitely often for convergence) as follows: the policy comes with the rule $A^\pi(s)$ of choosing an action. So starting from state S^n, we choose $a^n = A^\pi(S^n)$.
- Given our transition law, we simulate S^{n+1}. Naturally, then we choose $a^{n+1} = A^\pi(S^{n+1})$ and approximate the value of being in state S^n and taking action a^n by the one-step prediction

$$\hat{q}^n(S^n, a^n) = C(S^n, a^n) + \gamma \bar{Q}^{n-1}(S^{n+1}, a^{n+1}).$$

- Then, we use q^n to update $\bar{Q}^{n-1}$, just as in Q-learning.

Infinite-Horizon Problems

Starting from the Bellman Equation

$$V_t(s) = \sup_{a_t \in \mathscr{A}_t} \left(C_t(s,a_t) + \mathbb{E}\left[V_{t+1}(S_{t+1}) | S_t = s, a_t \right] \right)$$

we would intuitively arrive at a infinity-time horizon formulation

$$V(s) = \sup_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t C_t(S_t, A_t^{\pi}(S_t)) \right].$$

Define

$$P^{\pi,t} = \prod_{s=0}^{t-1} P^{\pi}$$

where P^{π} is the one-step transition matrix given policy π. Then

$$\begin{aligned} V_0^{\pi}(s) &= \sum_{u=0}^{\infty} \gamma^{u-t} P^{\pi,u-t} C_t(s, A^{\pi}(s)) \\ &= C_0^{\pi}(s) + \gamma P^{\pi} \sum_{u=1}^{\infty} \gamma^{u-t} P^{\pi,u-t} C_t(s, A^{\pi}(s)) \\ &= C_0^{\pi}(s) + \gamma P^{\pi} V_0^{\pi}(s). \end{aligned}$$

In simple cases this equation can be solved and we arrive at

$$V^\pi = (1 - \gamma P^\pi)^{-1} C_0^\pi. \tag{8}$$

Value iteration

Very similar to backward dynamic programming we can iterate the value function to find an optimal solution.

- We start with $v^{=}0$,
- for each $s \in \mathscr{S}$ we set

$$V^n(s) = \max_{a \in \mathscr{A}} \left(C(s,a) + \gamma \mathbb{E}_{s,a}[V^{n-1}(S)] \right)$$

- and stop if $\| V^n - V^{n-1} \|$ is sufficiently small.

See Powell, Section 3.10.3 for a proof of the convergence.

Policy iteration

While the value equation starts from the Bellman equation, policy iteration starts from Equation (8),

$$V^\pi = (1-\gamma P^\pi)^{-1} C_0^\pi.$$

- Starting with a policy π^0.
- Set $\pi' = \pi^{n-1}$. We compute $c^{n-1} = C_0^{\pi'}(s, A^{\pi'})$ and solve

  $$(1-\gamma P^{\pi'}) V^{\pi'} = c^{n-1}$$

  fir $V = V'$. The policy π^n is the solution of

  $$\arg\max_{a \in \mathscr{A}} \left(C(a) + \gamma P^\pi V' \right).$$

- Iterate until convergence.

Exploration-Expectation

In the search for an optimal policy, we may not always meet all possible states. One way out of this is to use a **randomized** strategy, which is called ε-greedy.

- This strategy chooses with probability $1-\varepsilon$ the optimal strategy and
- with probability ε a random strategy which allows us to explore.

Proof of the Bellman equation

Approximating the value function for fixed π

- As already illustrated it is important to approximate the value function efficiently.
- We will always denote by $\hat{V}$ the approximation (estimation) of our value function.
- If we have a state S^n and the associated estimate $\hat{V}^n$ at hand, we can update **or** estimate the $\hat{V}^{n+1}$, depending on what method we choose (again see Powell for numerous such approaches).
- We illustrate this once more: for N times we simulate as follows
  1. Simulate S_0^n, for $t = 0, \dots, T$ choose $a_t^\pi = A^\pi(S_t^n)$, and simulate S_{t+1}^n depending on a_t^n
  2. Given this, we compute $\hat{V}^n = \sum_{t=0}^T \gamma^t C(S_t^n, a_t^n)$
  3. Finally we use $(S^n, \hat{V}^n)_{n=1,\dots,N}$ to fit $\bar{V}^\pi$

Temporal Differences

A very well-known approach in this regard are **temporal differences**. We choose $\gamma = 1$ first. Clearly

$$\begin{aligned}\hat{V}_t^n &= \sum_{u=t}^{T} C(S_u^n, a_u^n) \\ &= \sum_{u=t}^{T} \left(C(S_u^n, a_u^n) - (\bar{V}_u^{n-1}(S_u) - \bar{V}_{u+1}^{n-1}(S_{u+1})) \right) + \bar{V}_t^{n-1}(S_t) - \bar{V}_{T+1}^{n-1}(S_{T+1})\end{aligned}$$

We use the freedom to set $V_{T+1} = 0$ and obtain the nice representation

$$\hat{V}_t^n = \bar{V}_t^{n-1}(S_t) + \sum_{u=t}^{T} \delta_u^\pi \tag{9}$$

with temporal differences

$$\delta_t^\pi = C(S_t^n, a_t^n) + \bar{V}_{t+1}^{n-1}(S_{t+1}) - \bar{V}_t^{n-1}(S_t).$$

Using a stochastic gradient algorithm leads to $\bar{V}_t^n(S_t^n) = \bar{V}_t^{n-1}(S_t^n) - \alpha_n(\bar{V}_t^{n-1}(S_t^n) - \hat{V}_t^n)$ and we arrive (including discounting now and a time-discount λ) at

$$\bar{V}_t^n(S_t) = \bar{V}_t^{n-1}(S_t) + \alpha_{n-1} \sum_{u=t}^{T} (\gamma\lambda)^{u-t} \delta_u^\pi.$$

The magic of Reinforcement Learning

Its incredible performance in games[24]:

- RL play checkers **perfect**
- Backgammon, Scrabble, Poker **superhuman**.
- They play Chess and Go on the level of a Grandmaster

[24] See David Silvers lectures

They produce interesting behaviour and results.

- https://www.youtube.com/watch?v=CIF2SBVY-J0
- Implementation in R:
  http://www.rblog.uni-freiburg.de/2017/04/08/
  reinforcementlearning-a-package-for-replicating-human-behavior-in-r/

Bayesian Optimization (BO)

- Typically we are interested in a problem

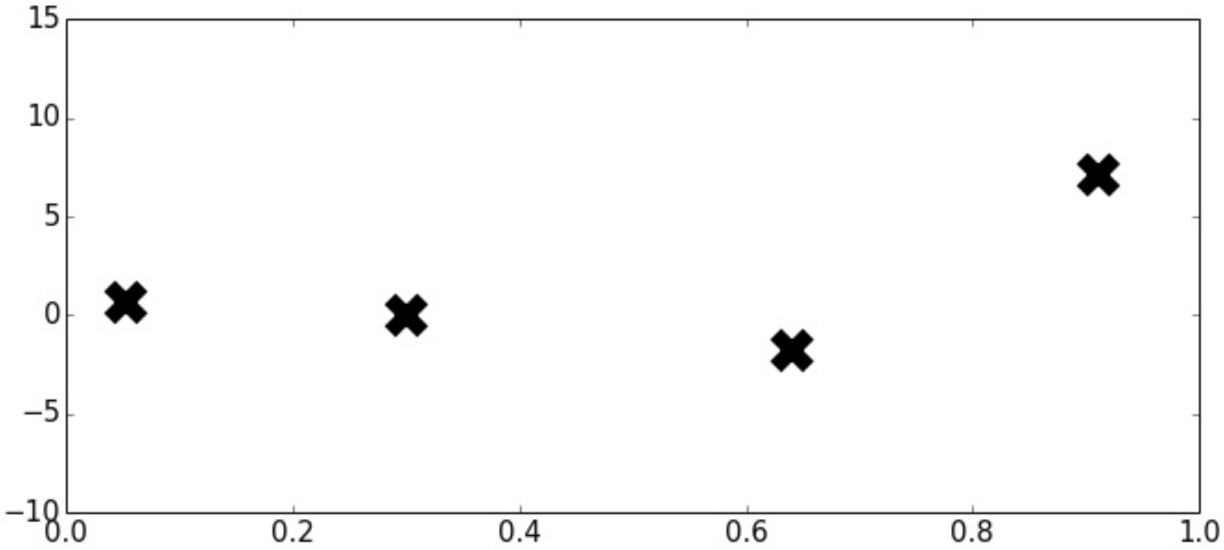
$$x^* = \arg\min_{x \in \mathcal{X}} f(x)$$

  with some "well behaved" function $f : \mathcal{X} \to \mathbb{R}^d$.
- However, in many cases f is not explicitly known and it also might be multimodal.
- Also the evaluations of f might contain errors or might be very expensive.
- A nowadays famous application is (hyper-) parameter tuning in Machine Learning. Such parameters are: the number of layers / units per layers, penalties, learning rates, etc.
- A classical example is the optimal design of experiments, or the case when statistics is needed but the likelihood is intractable.

- Currently feasiable are: grid search. This will need many function evaluations, which is not good if evaluations are expensive.
- **Random search** is a well-known alternative. The usage of pseudo-random numbers even improves performance.

The problem

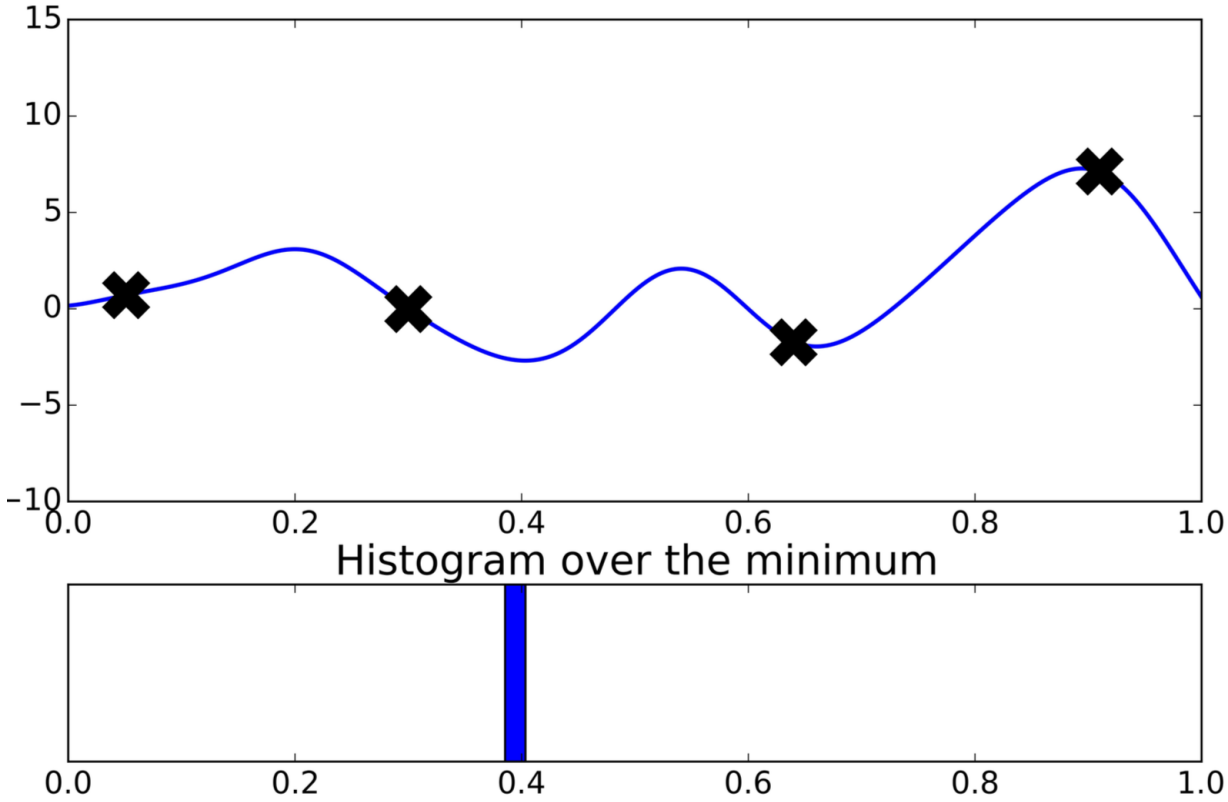
Let us illustrate the problem with a few pictures[25]



Where to choose the next point x where we evaluate $f(x)$??

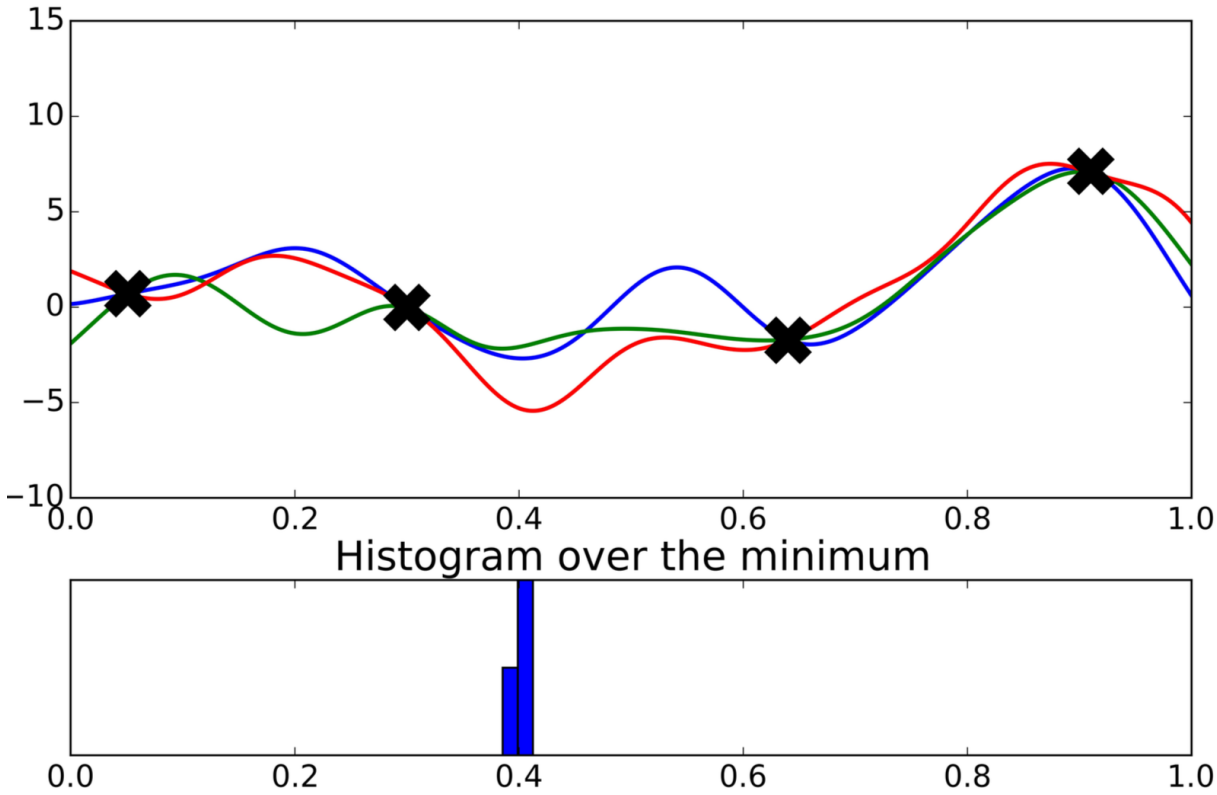
[25] Source: Javier González, Introduction to Bayesian Optimization. Masterclass, 2017 at Lancaster University.

Let us consider some possible curves. Here is one:

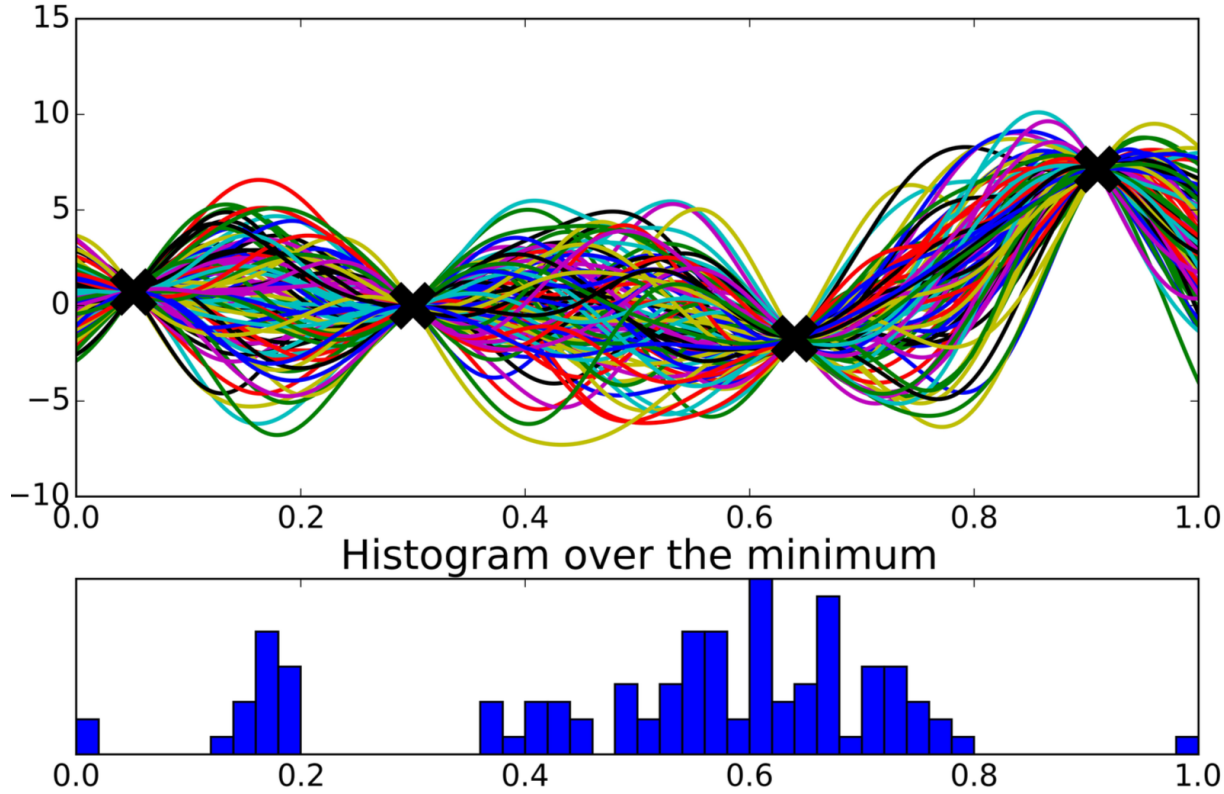


Clearly, we would choose to evaluate at the minimun and are finished. But this is not the only possible curve !

Three curves

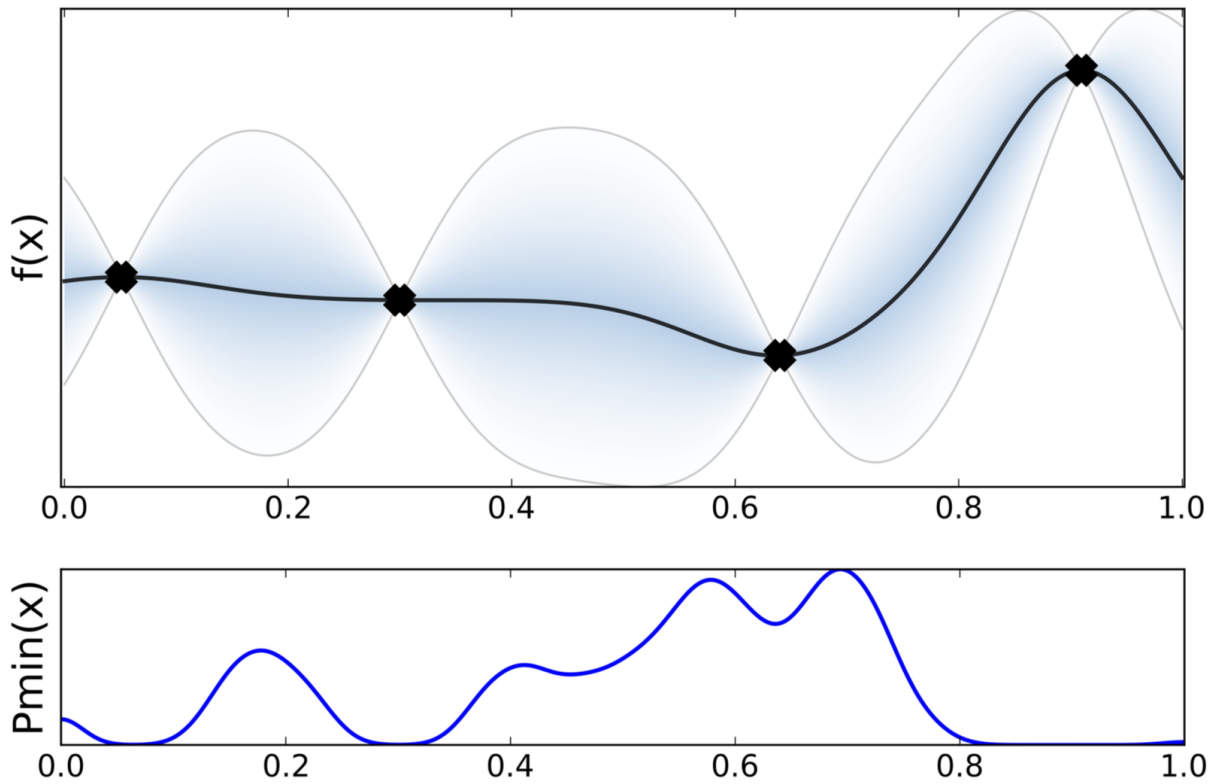


Many curves



If we think of a continuum of course, we arrive at the Bayesian representation of the problem.

We consider a density over the possible curves, which is called **prior**.



Where should we optimally place our next evaluation x^n??

- The approach is clear: we have a prior distribution p.
- Given some data $\mathscr{D}$ we update through Bayes' rule

$$p(x|\mathscr{D}) = \frac{p(\mathscr{D}|x)p(x)}{\mathbb{P}(\mathscr{D})}.$$

- Clearly, this is only possible if $\mathbb{P}(\mathscr{D}) \neq= 0$. If this is the case, we will use a conditionaly density given by

$$f(x|y) = \frac{f(x,y)}{f(y)}$$

where $f(x,y)$ is the joint density of x and y and $f(y)$ is the marginal density.

Historical overview

- Bayesian optimization dates back at least to works by Kushner[26] in 1964 and Mockus[27] in 1978.
- Since about 10 years there is a considerable interest of these methods in the machine learning community.

[26] Harold J Kushner (1964). „A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise“. In: **Journal of Basic Engineering** 86.1, S. 97–106.

[27] J Močkus (1975). „On Bayesian methods for seeking the extremum“. In: **Optimization Techniques IFIP Technical Conference**. Springer, S. 400–404.

Mathematical formulation

- In most cases the prior is chosen to be Gaussian - this is the case we will also focus here. There are other variants (Student processes) and interesting research questions in this direction
- A **Gaussian process** is a family $(X(x))_{x\in\mathcal{X}}$ of random variables, where for any (finite) $x_1,\dots,x_n$ the joint distribution of
$$X(x_1),\dots,X(x_n)$$
is Gaussian.
- The Gaussian process can be characterized by its **mean function**
$$m(x) := \mathbb{E}[X(x)]$$
and its **covariance function**
$$c(x,y) := \mathrm{Cov}(X(x),X(y)).$$
- We are able to observe (at a certain cost) $X(x)$ for a fixed sample $x_1,\dots,x_n$

- Typically we specify some kind of regression for our setup, like
$$X(x) = \beta x + \varepsilon_x$$
where the $\varepsilon(x_i)$, $i = 1, \dots, n$ are i.i.d.
- However, if x_1 is close to x_2 we would expect close outcomes rather than independent outcomes.
- This motivatives covariance functions of the form
$$c(x,y) \propto e^{-K(x,y)}$$
with a kernel function K. Often, $K(x,y) = \| x - y \|^{\alpha}$

Example

Gaussian process regression For example suppose that our observation is unbiased, i.e. we observe $Y(x)$ such that

$$\mathbb{E}[Y(x)] = f(x).$$

A model for this is the Gaussian regression

$$Y(x) = f(x) + \varepsilon(x).$$

The **posterior** distribution is given by

$$X(z)|X(x) = f \sim \mathcal{N}(\mu, \sigma^2)$$

where $\mu = \mu(f,x,z)$ and $\sigma = \sigma(f,x,z)$ are given by

$$\mu = m(z) + K(z,x)\frac{f - m(x)}{K(x,x) + \sigma^2 I_n}$$

$$\sigma^2 = K(z,z) - K(z,x)\frac{K(x,z)}{K(x,x) + \sigma^2 I_n}.$$

At the core is the following result. Consider the case where (X,Y) is a two-dimensional normal random variable with mean (a,A) and covariance matrix

$$\begin{pmatrix} b^2 & \rho bB \\ \rho bB & B^2 \end{pmatrix}.$$

Lemma

The conditional distribution of X given Y is Gaussian and

$$\mathbb{E}[X|Y] = a + \rho \frac{b}{B}(Y - A)$$
$$\mathbb{E}[(X - \mathbb{E}[X|Y])^2|Y] = b^2(1-\rho^2).$$

Beweis.

First consider standard normal X and Y with correlation ρ. The conditional density of X given $Y = y$ is

$$f(x|y) = \frac{f(x,y)}{f(y)} = \frac{2\pi\sqrt{1-\rho^2}}{\sqrt{2\pi}} \frac{\exp\left(-\frac{1}{2(1-\rho^2)}(x^2 - 2\rho xy + y^2)\right)}{\exp\left(-\frac{y^2}{2}\right)}$$

$$= \frac{1}{\sqrt{2\pi(1-\rho^2)}} \exp\left(-\frac{(x-\rho y)^2}{2(1-\rho^2)}\right).$$

Note that $\mathbb{E}[X|Y] = \rho Y$ and that $X - \rho Y$ is independent of Y as $\mathrm{Cov}(X - \rho Y, Y) = \rho - \rho = 0$. □

...

For the general case observe that $Z_1 := b^{-1}(X-a)$ and $Z_2 := B^{-1}(Y-A)$ are standard normal and $\mathrm{Cov}(Z_1, Z_2) = \rho$. Hence, X conditional on Y is again normally distributed and

$$\mathbb{E}[X|Y] = \mathbb{E}[a + bZ_1|Y] = a + b\rho Z_2 = a + \frac{\rho b}{B}(Y-A).$$

and we conclude by computing the conditional variance,

$$\mathbb{E}[(X - \mathbb{E}[X|Y])^2] = \mathbb{E}[(bZ_1 - \rho b Z_2)^2|Y] = b^2\mathbb{E}[(Z_1 - \rho Z_2)^2] = b^2(1-\rho^2). \qquad \square$$

Acquisition

- The next step is to **acquire** new data through an acquisition cirterium. Recall we have the observation $X(x)$ where we are now interested in choosing x optimally.
- The predictive variance is
$$\gamma(x) = \frac{f(x^*) - \mu(x)}{\sigma(x)}.$$
- Kushner suggest to study the probability of improvement
$$\alpha_{PI}(x) = \Phi(\gamma(x)).$$
- Mockus suggest the **expected improvement** and a further alternative (Srinivas e.a. 2010) is the lower confidence bound
$$\alpha_{LCB}(x) = \mu(x) - \kappa\sigma(x).$$