# Machine Learning

**Lecture notes**


Author

Peter Zaspel

May 18, 2021

# Contents

# Part I.

# Primer in mathematics

# 0. Univariate probability

Tools from probability are key for the modeling, derivation and analysis of machine learning methods. We expect from the regular reader that she or he has been exposed to a standard introduction to univariate probability. This practically means that all basic notions that are associated to one-dimensional discrete and continuous random variables are expected to be well known.

Nevertheless, the last contact to the respective mathematics might have been some time ago or not all notions have been studied that are understood as "standard", here. Therefore, in this chapter, we will have a crash-course style repetition of all necessary definitions and facts (without proofs) that will become important. This will also help us to fix notation. In addition, we will discuss a few examples. Nevertheless, it should be kept in mind, that this short overview does not intend to be a true introduction into the field.

The following overview is mostly based on [3, 4, 1].

## 0.1 Elementary notions and results

We start at the very beginning and build up terminology for chance experiments:

**Definition 0.1** We consider a chance experiment. The set of all outcomes that may appear in a realization of this experiment is called **sample space**, denoted by $\Omega$. We call $\omega \in \Omega$ **sample points** and the associated sets $\{\omega\}$ **elementary events** for this experiment. A given subset $A \subseteq \Omega$ is called **event**.

We will use the following examples throughout this chapter.

**Example 0.1** (Coin toss) We model coin toss(es). Our sample points are $H$ for "head" and $T$ for "tail". Then the sample space $\Omega_1$ for a single coin toss is

$$\Omega_1 = \{H, T\}\,.$$

Similarly, we can model the sample space for tossing two coins by

$$\Omega_2 = \{HH, HT, TH, TT\}\,.$$

One potential event in the two coin chance experiment could be AT LEAST ONE HEAD IN TWO COIN TOSSES. If we call this even $A$, it would, for the given $\Omega_2$, look like

$$A = \{HH, HT, TH\}.$$

△

**Example 0.2** (Throw of dice) In case of throwing a dice, a natural sample space $\Omega$ is given by

$$\Omega = \{1, 2, 3, 4, 5, 6\}.$$

The associated elementary events are

$$A_1 = \{1\}, \quad A_2 = \{2\}, \quad \dots$$

and the event

$$B = \{1, 3, 5\}$$

models ODD NUMBER. △

Let us now introduce some basic operations on events.

**Definition 0.2** Let $\Omega$ be a sample space and $A, B \subseteq \Omega$ events.

- $A$ and $B$ are **equivalent**, if $A = B$.
- The **union** of $A$ and $B$ is $A \cup B$.
- The **intersection** of $A$ and $B$ is $A \cap B$.
- $A$ and $B$ are **disjoint** or **mutually exclusive** if $A \cap B = \emptyset$.
- The complement of $A$ is $\bar{A} := \Omega \setminus A$.

**Example 0.3** (Throw of dice) We continue the dice chance experiment example:

- Let $A = \{1\}$, $B = \{6\}$ be two events, then $A \cup B$ corresponds to the event EITHER 1 OR 6 IS THE OUTCOME.
- The event ODD NUMBER is given by $A = \{1, 3, 5\}$ and the event NUMBER SMALLER 4 is given by $B = \{1, 2, 3\}$. Then $A \cap B = \{1, 3\}$ models ODD NUMBER THAT IS SMALLER THAN 4[1)].

△

Now we come to the most important part, the introduction of the concept of *probability*:

**Definition 0.3** Let $\Omega$ be a sample space and $P : \mathcal{P}(\Omega) \to \mathbb{R}$ a real-valued function on events on $\Omega$. $P$ is called **probability on** $\Omega$, if the following axioms hold:

[1)] To be more precise, this would be the event ODD NUMBER AND NUMBER SMALLER 4.

1. $P(A) \geq 0$ for all events $A$.
2. $P(\Omega) = 1$.
3. If $A_1, A_2, \ldots$ is a pairwise disjoint sequence of events, then

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)\,.$$

The real number $P(A)$ is the **probability of the event** $A$.

**Remark** (Power set) Recall that $\mathcal{P}(\Omega)$ is the *power set* of $\Omega$, thus the set of all subsets of $\Omega$. △

**Remark** (Mathematical terminology and rigor) The experienced reader will have noticed that our definition of a probability on $\Omega$ as a function $P : \mathcal{P}(\Omega) \to \mathbb{R}$ is a bit too restrictive, as it implies that the probability is defined on all subsets of $\Omega$. Here, classical definitions would first introduce the set of all possible events on $\Omega$ that can be generated using the operations from Definition 0.2. Then, the probability would only be a function on these events. Being even more formal, we would hence need to introduce a *$\sigma$-algebra* (i.e. that set of events).

One big challenge in machine learning education is the appropriate choice of mathematical terminology and rigor that is used to describe all objects of interest. In principle, we would have to talk about *measure spaces*, *$\sigma$-algebras*, *measurable functions*, etc. to achieve full mathematical exactness. At the same time, not all readers will be familiar with many of these notions. Our way to discuss these topics will therefore sometimes have to be a compromise between full mathematical details / exactness and readability. There is the effort to, as much as possible, comment on mathematical short-cuts in the upcoming chapters. Nevertheless, this might not always be the case. Readers that feel to profit more from a more rigorous presentation of statements might be referred to the very good reference [9]. △

**Example 0.4** (Throw of dice) For a fair dice, the following statements will hold:

- The elementary events $A_1 = \{1\}, \ldots$ have probabilities

$$P(A_i) = \frac{1}{6}\,.$$

- The probability of event $A = \{1, 3, 5\}$ is

$$P(A) = \frac{1}{2}\,.$$

△

We summarize a few rules of of computing with probabilities that can be derived from the previous definitions.

**Lemma 0.1** Let $A, B \subseteq \Omega$ be events. It holds

- $P(\bar{A}) = 1 - P(A)$.
- $P(B) \leq P(A)$ if $B \subseteq A$.
- $P(A) \leq 1$
- $P(A \cup B) = P(A) + P(B) - P(A \cap B)$
- $P(A \cup B) = P(A) + P(B)$, if $A$ and $B$ are disjoint.

Conditional probabilities will become very important in machine learning. We start with their definition.

**Definition 0.4** Let $\Omega$ be a sample space, $B \subseteq \Omega$ be an event with $P(B) > 0$ and $A \subseteq \Omega$ be an arbitrary event. We define the **conditional probability of $A$ given $B$** as

$$P(A|B) := \frac{P(A \cap B)}{P(B)} = \frac{P(A, B)}{P(B)}$$

**Remark** Note that it holds $P(A, B) := P(A \cap B)$, hence we used the above definition to additionally introduce some notation that is used in some mathematics literature. △

**Example 0.5** Let us assume that the probability of getting older than 70 is 0.87. Similarly, we assume that the probability of getting at least and older than 75 is 0.8.

Question: We select randomly a person of 70 years. What is the probability of surviving at least until 75?

We call the event PERSON LIVES AT LEAST 75 YEARS $A$ and the event PERSON LIVES AT LEAST 70 YEARS $B$. We know

$$P(A) = 0.8, \quad P(B) = 0.87\,.$$

Since $A$ is a subset of $B$, we know $A \cap B = A$. The question asks for the conditional probability of $A$ given $B$, which we compute as

$$P(A|B) = \frac{P(A, B)}{P(B)} = \frac{P(A)}{P(B)} = \frac{0.8}{0.87} \approx 0.92\,.$$

△

And again we can state a few results on conditional probabilities based on the given definitions.

**Lemma 0.2** Let $\Omega$ be a sample space, $B \subseteq \Omega$ be an event with $P(B) > 0$ and $A \subseteq B$ be an arbitrary event. It holds:

- $P(\Omega|B) = 1$.
- If $A_1, A_2, \ldots$ are pairwise disjoint, then

$$P\left(\bigcup_{i=1}^{\infty} A_i \middle| B\right) = \sum_{i=1}^{\infty} P(A_i|B)\,.$$

**Lemma 0.3** Let $A, B \subseteq \Omega$ be events and we require $0 < P(B) < 1$, then it holds

$$P(A) = P(A|B)P(B) + P(A|\bar{B})P(\bar{B})$$

**Lemma 0.4** Let $B_1, B_2, \ldots, B_n$ be a partition of $\Omega$, such that $P(B_i) > 0$, for all $i = 1, \ldots, n$, then it holds for any $A \subseteq \Omega$ that

$$P(A) = \sum_{i=1}^{n} P(A|B_i)P(B_i)$$

The following result, *Bayes' theorem*, is of particular importance in machine learning:

**Theorem 0.1** (Bayes) Let $B_1, B_2, \ldots, B_n$ be a partition of $\Omega$, such that $P(B_i) > 0$, for all $i = 1, \ldots, n$, then it holds for any $A \subseteq \Omega$ with $P(A) > 0$ that

$$P(B_i|A) = \frac{P(A|B_i)P(B_i)}{\sum_{j=1}^{n} P(A|B_j)P(B_j)}\,.$$

**Remark** (Terminology) The different probabilities in Bayes' theorem have specific names. The probabilities $P(B_i|A)$ are called **a posterior** probabilities, while the probabilities $P(B_i)$ are called **priori** / **unconditional** probabilities. Moreover we call the $P(A|B_i)$ **conditional** probabilities. △

We can briefly derive another form of Bayes' theorem:

**Corollary 0.1** Let $B \subseteq \Omega$ be an event, such that $0 < P(B) < 1$, then it holds for any $A \subseteq \Omega$ with $P(A) > 0$ that

$$P(B|A) = \frac{P(A|B)P(B)}{P(A|B)P(B) + P\left(A|\overline{B}\right) P\left(\overline{B}\right)}\,.$$

Another elementary notion is the *(in)dependence* of two events:

**Definition 0.5** Let $A, B \subseteq \Omega$ be events. They are called **independent** if

$$P(A \cap B) = P(A)P(B)\,.$$

Otherwise, they are called **dependent**.

## 0.2 Discrete random variables

In machine learning, we model data via *random variables* (RVs). Therefore, it is crucial to understand these.

**Definition 0.6** Let $\Omega$ be a sample space. We call a function

$$X : \Omega \to \mathbb{R}$$

**random variable**, if for any interval $I \subset \mathbb{R}$, the set $\{\omega \in \Omega | X(\omega) \in I\}$ is an event on $\Omega$. By $P(X \in I)$, we denote the probability of $X$ to take values on $I$.

**Remark** This is a fairly general definition for random variables, which indeed covers both, *discrete* and *continuous* RVs. Discrete RVs will be introduced very soon. In their case, we have as image of $X$ a finite or a countable infinite set. In that case, as seen soon, we will not write $P(X \in I)$ but $P(X = x)$ with $x$ a value from the image. △

**Remark** We give no example here, but introduce a first example as soon as we actually arrived at *discrete* RVs. △

The *(cumulative) distribution function* assigns a probability to an RV.

**Definition 0.7** Let $X : \Omega \to \mathbb{R}$ be a random variable. The function $F : \mathbb{R} \to \mathbb{R}$ with

$$F(t) = P(X \leq t) = P(\omega \in \Omega | X(\omega) \leq t)$$

is called **(cumulative) distribution function** (CDF) of $X$.

For the CDF we know some basic properties.

**Lemma 0.5** Let $\Omega$ be a sample space, $X$ a random variable on $\Omega$ and $F$ the CDF of $X$. It holds:

- $F(t_1) \leq F(t_2)$ for $t_1 < t_2$.
- $\lim_{t\to\infty} F(t) = 1$, $\lim_{t\to-\infty} F(t) = 0$
- For $\{t_n\}_{n\geq 1}$, $t_n \in \mathbb{R}$, a decreasing sequence with limit $t$, we have $\lim_{n\to\infty} F(t_n) = F(t)$.

Rather important for the calculation of probabilities for the outcomes of RVs is this lemma.

**Lemma 0.6**

Let $F$ be a CDF for the random variable $X$ and let $a, b \in \mathbb{R}$, $a < b$ be given. It holds

$$P(a < X \leq b) = F(b) - F(a)\,.$$

While all previous statements and definitions have been independent of the actual flavor of random variable, we now come to the crucial differentiation.

**Definition 0.8** Let $\Omega$ be a sample space and $X$ a random variable on $\Omega$. We call $X$ a **discrete random variable** if $X$ can take only a finite or at most infinite but countable number of values. In this case, the CDF $F$ of $X$ is called **discrete distribution**.

Another case are continuous random variables, which we will introduce in the next section.

**Definition 0.9** Let $X : \Omega \to \mathbb{R}$ be a discrete random variable. We call the function $p : \mathbb{R} \to \mathbb{R}$ with

$$p(x) := P(X = x)$$

**probability (mass) function** (PMF) of $X$. It describes the probability of the event $X = x$, i.e. that $X$ takes the value $x$.

We finally come to our first example of a (discrete) random variable.

**Example 0.6** (Coin toss) We again consider coin tosses. Specifically, our chance experiment considers three independent, subsequent coin tosses. The sample space is thus

$$\Omega = \{HHH, HHT, HTH, \ldots\}\,.$$

We introduce a random variable $X : \Omega \to \mathbb{R}$ that gives the number of tails in three tosses of a coin. $X$ can be defined by enumerating all cases:

$$X(HHH) = 0, \quad X(HHT) = 1, \quad X(HTH) = 1, \quad \ldots$$

Moreover, we can immediately give the associated PMF $p$:

$$p(0) = P(X = 0) = \frac{1}{8}, \quad P(X = 1) = P(\{HHT, HTH, THH\}) = \frac{3}{8}, \quad \ldots$$

△

**Remark** (Range of $X$) Let $X : \Omega \to \mathbb{R}$ be a discrete RV. While we still state that $X$ maps to values in $\mathbb{R}$, we will often also give the range $R_X$ of $X$, which we define as the image of the input $\Omega$ under the function $X$, i.e. $R_X := X(\Omega)$ for clarification. △

**Example 0.7** (Throw of dice) The shown number after the throw of a dice can also be modeled by a discrete random variable:

$$X : \Omega \to \mathbb{R}$$

Obviously, the range of $X$ is $R_X = \{1, 2, 3, 4, 5, 6\}$. The RV $X$ has the PMF $p : \mathbb{R} \to \mathbb{R}$ with

$$p(x) = \begin{cases} \frac{1}{6} & x \in R_X\,, \\ 0 & \text{else}\,. \end{cases}$$

We can also give its CDF evaluated at the most relevant points

- $F(1) = P(X \leq 1) = \frac{1}{6}$
- $F(2) = P(X \leq 2) = \frac{1}{3}$
- $F(3) = P(X \leq 3) = \frac{1}{2}$
- $F(4) = P(X \leq 4) = \frac{2}{3}$
- $F(5) = P(X \leq 5) = \frac{5}{6}$
- $F(6) = P(X \leq 6) = 1.$

△

A few basic properties can be shown for PMFs:

**Lemma 0.7** Let $X$ be a discrete random variable and $p$ its PMF. Let the range of $X$ be $R_X := \{x_1, x_2, \ldots\}$. It holds:

- $p(x) = 0$, for all $x \notin R_X$.

- $p(x_i) \geq 0$, for all $i = 1, 2, \ldots$.
- $\sum_{i=1}^{|R_X|} p(x_i) = 1$.

We come to a next fundamental notion:

**Definition 0.10** Let $X$ be a discrete random variable with PMF $p$ and range $R_X$. The **expected value / expectation / mean** of $X$ is denoted by $E(X)$ and is given by

$$\mathrm{E}\,(X) = \sum_{x \in R_X} x\, p(x) = \sum_{x \in R_X} x\, P(X = x)\,,$$

under the assumption that this series converges absolutely. The expectation *does not exist*, if this assumption is not fulfilled.

**Example 0.8** (Throw of dice) Let $X : \Omega \to \mathbb{R}$ again be the number on a dice after one throw. We can compute the mean of this RV as

$$\mathrm{E}\,(X) = \sum_{x \in R_X} x\, p(x) = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} = 3.5\,.$$

△

We can also apply functions to random variables and compute the mean for the outcome of these functions.

**Lemma 0.8** Let $X$ be a discrete RV with PMF $p$ and range $R_X$. For a given function

$$g : \mathbb{R} \to \mathbb{R}$$

the expectation of the new random variable $g(X)$ is given by

$$\mathrm{E}\,(g(X)) = \sum_{x \in R_X} g(x)p(x),\,.$$

In many calculations, we will use the linearity of the expectation.

**Lemma 0.9** (Linearity of expectation) Let $X_1$, $X_2$ be discrete RVs over $\Omega$ and $c \in \mathbb{R}$. It holds:

$$\mathrm{E}\,(cX_1) = c\,\mathrm{E}\,(X_1)\ ,$$

$$\mathrm{E}\left(X_1 + X_2\right) = \mathrm{E}\left(X_1\right) + \mathrm{E}\left(X_2\right) .$$

Beside of the mean, two other important statistical properties for a random variable are the *variance* and the *standard deviation*.

**Definition 0.11** Let $X$ be a discrete RV with existing mean $\mu = \mathrm{E}\left(X\right)$. We call the quantity

$$\mathrm{Var}\left(X\right) = \mathrm{E}\left((X-\mu)^2\right) = \mathrm{E}\left((X - E(X))^2\right)$$

**variance** of $X$. Furthermore the real value $\sigma = \sqrt{\mathrm{Var}\left(X\right)}$ is called **standard deviation** of $X$.

**Remark** The variance is often denoted by $\sigma^2$. △

The following lemma might simplify the evaluation of the variance.

**Lemma 0.10** For a discrete RV $X$, it holds

$$\mathrm{Var}\left(X\right) = \mathrm{E}\left(X^2\right) - \left(\mathrm{E}\left(X\right)\right)^2$$

**Example 0.9** (Throw of dice) We continue our dice example and compute the variance of the RV $X$ by

$$\mathrm{Var}\left(X\right) = \mathrm{E}\left(X^2\right) - \left(\mathrm{E}\left(X\right)\right)^2 = \frac{105}{36} \approx 2.9$$

△

There is a simplified way do evaluate the variance of a linear function in a given random variable:

**Lemma 0.11** Let $X$ be a discrete RV and $a, b \in \mathbb{R}$. For the variable

$$Y = aX + b$$

the variance and standard deviation are given by

$$\mathrm{Var}\left(Y\right) = a^2 \,\mathrm{Var}\left(X\right), \quad \sigma_Y = |a|\sigma_X .$$

The following two inequalities are sometimes used to find estimates for probabilities of RVs. We collect them here to be able to reference them.

**Theorem 0.2** (Markov's inequality) Let $X : \mathbb{R} \to \mathbb{R}_{\geq 0}$ be a discrete RV, such that $\mathrm{E}(X)$ exists. For $t > 0$, we have

$$P(X \geq t) \leq \frac{\mathrm{E}(X)}{t}.$$

**Theorem 0.3** (Chebyshev's inequality) Let $X$ be a discrete RV with mean $\mu$ and variance $\sigma^2$. For $t > 0$, we have

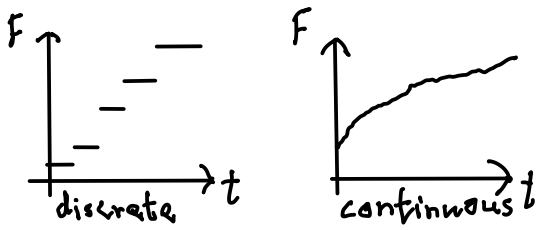
$$P(|X - \mu| \geq t) \leq \frac{\sigma^2}{t^2}.$$

## 0.3 Continuous random variables

In this section we will discuss important notions for *continuous* RVs.

**Definition 0.12** Let $\Omega$ be a sample space and $X$ a random variable on $\Omega$. We call its CDF $F(t) = P(X \leq t)$ a **continuous distribution**, if $F$ is continuous everywhere. A random variable with a continuous distribution is called **continuous random variable**. It has an uncountable range.

**Remark** (Discrete vs. continuous distribution) Let us briefly visualize the difference between a discrete (left) and a continuous (right) distribution:




△

In case of discrete random variables, we could associate probabilities to events via their probability (mass) functions. Continuous RVs have no probability mass functions (PMFs). Instead, in this case, the most similar concept compared to a PMF is a *probability density function*.

**Definition 0.13** Let $X$ be a continuous RV. Let us assume that there exists a function

$\rho : \mathbb{R} \to \mathbb{R}_{\geq 0}$ such that it holds

$$P(X \in A) = \int_A \rho(x)dx$$

for all subsets $A \subseteq \mathbb{R}$, which can be written as the union of a finite / infinite number of intervals. If $\rho$ and the above integral exist, we call $X$ **absolutely continuous** and $\rho$ the **(probability) density function** (PDF) or **density** of $X$.

**Example 0.10** (Car repair) We could model the time (in hours) required to repair a car in a garage by a continuous RV $X : \Omega \to [0, 4]$ with a density

$$\rho(x) = \frac{3}{32}(4x - x^2)\,.$$

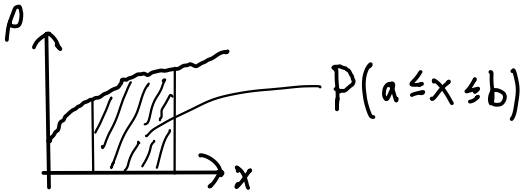
△

The fourth point of the next lemma is important for calculating probabilities in the continuous setting.

**Lemma 0.12** Let $X$ be a continuous RV with CDF $F$ and we expect that there exists the associated PDF $\rho$, then it holds:

- $\int_{-\infty}^{\infty} \rho(x)dx = 1$.
- $F(t) = \int_{-\infty}^{t} \rho(x)dx$.
- $\rho(x) = \frac{\partial}{\partial x}F(x)$ for all $x$, at which $\rho$ is continuous.
- $P(a \leq X \leq b) = F(b) - F(a) = \int_a^b \rho(x)dx$.
  (Any $\leq$ can be replaced by $<$.)

**Remark** The interpretation of the fourth item in the beforehand lemma is that the probability can be evaluated as the area below the density curve on the respective interval:




△

In case we construct a new continuous RV $Y$ as $Y := g(X)$ and function $g$ fulfills some properties, we can immediately derive the density of $Y$ from the density of $X$:

**Lemma 0.13** Let $X$ be continuous RV with density $\rho_X$, $a, b \in \mathbb{R}$, and $\rho_X$ such that $\rho_X(x) > 0$ for $a < x < b$. We assume to have an invertible, differentiable function $g : \mathbb{R} \to \mathbb{R}$ with no sign change on the open interval $(a, b)$. Then, we can give the density function $\rho_Y$ of the random variable

$$Y = g(X)$$

for all $y \in (g(a), g(b))$ by the formula

$$\rho_Y(y) = \left| \frac{\partial g^{-1}}{\partial y}(y) \right| \rho_X \left( g^{-1}(y) \right)$$

For continuous RVs we can, similarly to the case of discrete RVs, introduce an expected value:

**Definition 0.14** Let $X$ be a continuous RV with density $\rho$. We define the **expected value / expectation / mean** of $X$ by

$$\mathrm{E}(X) = \int_{-\infty}^{\infty} x \, \rho(x) dx$$

under the assumption that the integral exists and converges absolutely, hence

$$\int_{-\infty}^{\infty} |x| \, \rho(x) dx < \infty \, .$$

If the assumption is not fulfilled, the expectation of $X$ does not exist.

**Example 0.11** We can continue our car repair example and compute the mean time for repairing a car by

$$\mathrm{E}(X) = \int_0^4 x(4x - x^2) dx = \left[ 4 \cdot \frac{1}{3} x^3 - \frac{1}{4} x^4 \right]_0^4 = \ldots$$

△

We also have a simple rule to compute means of continuous RVs to which a function is applied.

**Lemma 0.14** Let $X$ be a continuous RV with density $\rho$. Let further $g : \mathbb{R} \to \mathbb{R}$ be given. Under the assumption of existence, the expectation of the new random variable

$Y = g(X)$ is given by

$$\mathrm{E}\left(g(X)\right) = \int_{-\infty}^{\infty} g(x)\rho(x)dx$$

Moreover, we observe also for the continuous case that the mean is linear.

**Lemma 0.15** Let $X_1$, $X_2$ be a continuous RVs over $\Omega$ and $c \in \mathbb{R}$. It holds:

$$\mathrm{E}\left(cX_1\right) = c\,\mathrm{E}\left(X_1\right) ,$$

$$\mathrm{E}\left(X_1 + X_2\right) = \mathrm{E}\left(X_1\right) + \mathrm{E}\left(X_2\right) .$$

Certainly, in full analogy to the discrete case, we introduce variance and standard deviation,

**Definition 0.15** Let $X$ be a continuous RV with existing mean $\mu = \mathrm{E}\left(X\right)$. We call the quantity

$$\sigma^2 = \mathrm{Var}\left(X\right) = \mathrm{E}\left((X-\mu)^2\right) = \mathrm{E}\left((X - E(X))^2\right)$$

**variance** of $X$. Furthermore the real value $\sigma = \sqrt{\mathrm{Var}\left(X\right)}$ is called **standard deviation** of $X$.

find a simplified rule for their evaluation,

**Lemma 0.16** For a continuous RV $X$, it holds

$$\mathrm{Var}\left(X\right) = \mathrm{E}\left(X^2\right) - \left(\mathrm{E}\left(X\right)\right)^2$$

and give a statement on the variance for a linear transformation of an underlying continuous RV.

**Lemma 0.17** Let $X$ be a continuous RV and $a, b \in \mathbb{R}$. For the variable

$$Y = aX + b$$

the variance and standard deviation are given by

$$\mathrm{Var}\left(Y\right) = a^2\,\mathrm{Var}\left(X\right), \quad \sigma_Y = |a|\sigma_X .$$

Finally, Markov's and Chebyshev's inequality also hold for the continuous case:

**Theorem 0.4** (Markov's inequality) Let $X : \mathbb{R} \to \mathbb{R}_{\geq 0}$ be a continuous RV, such that $\mathrm{E}(X)$ exists. For $t > 0$, we have

$$P(X \geq t) \leq \frac{\mathrm{E}(X)}{t}.$$

**Theorem 0.5** (Chebyshev's inequality) Let $X$ be a continuous RV with mean $\mu$ and variance $\sigma^2$. For $t > 0$, we have

$$P(|X - \mu| \geq t) \leq \frac{\sigma^2}{t^2}.$$

## 0.4 Important continuous distributions

We conclude this chapter by briefly discussing a few common continuous distributions. Somewhat the simplest example is the *uniform distribution*.

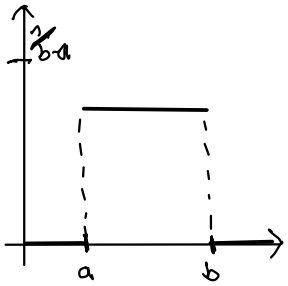
**Definition 0.16** Let $X$ be a continuous RV with density

$$\rho(x) = \begin{cases} \frac{1}{b-a}, & a \leq x \leq b, \\ 0, & \text{else.} \end{cases}$$

In this case, we say that $X$ follows the **uniform distribution** over the interval $[a, b]$, denoting it by

$$X \sim \mathcal{U}[a, b].$$

**Remark** To get an intuition for the density of the uniform distribution, we give its visualization:




△

It is then easy do derive the mean and the variance for a random variable that has the uniform distribution.

**Lemma 0.18** The expectation and variance of a RV $X \sim \mathcal{U}[a, b]$ are

$$\mathrm{E}(X) = \frac{a+b}{2}, \quad \mathrm{Var}(X) = \frac{(b-a)^2}{12}.$$

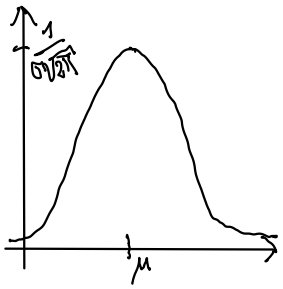
An even more important case is the *Gaussian distribution*:

**Definition 0.17** Let $X$ be a continuous RV. It is said to follow the **normal / Gaussian distribution** with parameters $\mu \in \mathbb{R}$ and $\sigma^2 > 0$, if its density function $\rho$ is given by

$$\rho(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}.$$

We denote this by $X \sim \mathcal{N}(\mu, \sigma^2)$.

**Remark** We visualize the density of the Gaussian distribution:




△

Mean and variance of the Gaussian distribution are given by

**Lemma 0.19** The expectation, variance and standard deviation of a RV $X \sim \mathcal{N}(\mu, \sigma^2)$ are

$$\mathrm{E}(X) = \mu, \quad \mathrm{Var}(X) = \sigma^2, \quad \sqrt{\mathrm{Var}(X)} = \sigma\,.$$

A further common distribution is the *exponential distribution*.

**Definition 0.18** Let $X$ be a continuous RV. It is said to follow the **exponential distribution** with parameter $\lambda > 0$, if its density function $\rho$ is given by

$$\rho(x) = \begin{cases} \lambda e^{-\lambda x}, & x \geq 0, \\ 0, & \text{else.} \end{cases}$$

We denote this by $X \sim \mathcal{E}(\lambda)$.

**Example 0.12** Examples for the use of the exponential distribution are a model for the time until a radioactive particle decays or a model for the time until a next phone call comes in. △

We know the mean and variance for the exponential distribution by

**Lemma 0.20** The expectation and variance of a RV $X \sim \mathcal{E}(\lambda)$ are

$$\mathrm{E}(X) = \frac{1}{\lambda}, \quad \mathrm{Var}(X) = \frac{1}{\lambda^2}\,.$$

We also briefly characterize the *Gamma distribution*.

**Definition 0.19** Let $X$ be a continuous RV. It is said to follow the **Gamma distribution** with parameters $\alpha, \beta > 0$, if its density function $\rho$ is given by

$$\rho(x) = \begin{cases} \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}, & x \geq 0, \\ 0, & \text{else.} \end{cases}$$

We denote this by $X \sim \Gamma(\alpha, \beta)$.

**Example 0.13** The Gamma distribution has been used to model aggregated insurance claims or accumulated rainfall in a basin. △

Its mean and variance are given by

**Lemma 0.21** The expectation and variance of a RV $X \sim \Gamma(\alpha, \beta)$ are

$$\mathrm{E}(X) = \frac{\alpha}{\beta}, \quad \mathrm{Var}(X) = \frac{\alpha}{\beta^2}.$$

To conclude, we state the *Beta distribution* with its mean and variance.

**Definition 0.20** Let $X$ be a continuous RV. It is said to follow the **Beta distribution** with parameters $\alpha, \beta > 0$, if its density function $\rho$ is given by

$$\rho(x) = \begin{cases} \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}, & 0 < x < 1, \\ 0, & \text{else.} \end{cases}.$$

We denote this by $X \sim \mathrm{Beta}(\alpha, \beta)$. The **Beta function** $B(\alpha, \beta)$ is given by

$$B(\alpha, \beta) = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1} dx.$$

**Lemma 0.22** The expectation and variance of a RV $X \sim \mathrm{Beta}(\alpha, \beta)$ are

$$\mathrm{E}(X) = \frac{\alpha}{\alpha+\beta}, \quad \mathrm{Var}(X) = \frac{\alpha\beta}{(\alpha+\beta+1)(\alpha+\beta)^2}.$$

# 1. Multivariate probability

In machine learning, we often model our input and output data as samples drawn from some random variables. Let us consider the example of image classification. Here, we have images as inputs and some labels as outputs. If we were to model this input as random variable, we would need a vector of random variables, where each entry in the vector corresponds to one of three colors in one of e.g. $1024 \times 1024$ pixels. Effectively, we would have to deal with a vector of random variables with several millions entries. Moreover, as we will see later, we will model the output (vector) as a random variable that is dependent on the input, leading to a complex interaction between input and output random variables / vectors.[1)]

It is evident, that the terminology and tools that we have discussed in the previous chapter do not allow us yet to describe and analyse potentially interacting vectors of random variables. Since the target reader might not have seen the necessary theory on multivariate probability, we will use this chapter to go through the necessary basics in this field. While we discuss all necessary notions with examples, we will not be able to go into the details of the necessary proofs. Readers interested in the full theory of multivariate probability are referred to the respective literature or courses on advanced probability.

## 1.1 Discrete random variables

We start with the case of two discrete random variables and will extend our statements towards vectors of random variables towards the end of this section.

If we consider two discrete random variables, a first task is to introduce how a (joint) PMF for theses RV will look like:

**Definition 1.1** Let $X, Y$ be two discrete RVs on $\Omega$. We call the function $p : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ **joint probability mass function**, short **joint PMF**, of $X, Y$, if it holds

$$\begin{aligned} p(x,y) &:= P(X = x, Y = y) \\ &= P(\{\omega \in \Omega | X(\omega) = x\} \cap \{\omega \in \Omega | Y(\omega) = y\}) \end{aligned}$$

[1)] We will talk much more about this in the next chapter.

Hence the joint PMF simply gives the probability for the case that the events $X = x$ and $Y = y$ hold at the same time. We immediately come to an example.

**Example 1.1** (Three coin tosses) We continue our coin toss examples. This time we have a chance experiment with three independent coin tosses, leading to the sample space $\Omega = \{HHH, HHT, HTT, \ldots\}$. We introduce two random variables

$$X : \Omega \to \mathbb{R}, \quad Y : \Omega \to \mathbb{R},$$

where the first random variable gives the number of "heads" in the first two tosses and the second RV gives the number of "heads" in the last two tosses.[2)]

We would like to compute their joint PMF:

$$\begin{aligned}
p(0,0) &= P(X=0, Y=0) = P(\{\omega \in \Omega | X(\omega) = 0\} \cap \{\omega \in \Omega | Y(\omega) = 0\}) \\
&= P(\{TTH, TTT\} \cap \{HTT, TTT\}) = P(\{TTT\}) = \frac{1}{8} \\
p(0,1) &= P(X=0, Y=1) = P(\{TTH, TTT\} \cap \{HHT, HTH, THT, TTH\}) \\
&= P(\{TTH\}) = \frac{1}{8} \\
&\ldots \\
p(1,1) &= P(X=1, Y=1) \\
&= P(\{HTH, THH, HTT, THT\} \cap \{HHT, HTH, THT, TTH\}) \\
&= P(\{HTH, THT\}) = \frac{1}{4} \\
&\ldots
\end{aligned}$$

△

The simple calculation rules that we know from (non-joint) PMFs, easily extend to the new definition:

**Lemma 1.1** Let $X, Y$ be two discrete RVs on $\Omega$ and $p$ their joint PMF. Let the ranges of $X$, $Y$ be $R_X = \{x_1, x_2, \ldots\}$ and $R_Y = \{y_1, y_2, \ldots\}$. It holds:

1. $p(x, y) = 0$, for all $(x, y) \notin R_X \times R_Y$.
2. $p(x_i, y_j) \geq 0$, for all $(x_i, y_j) \in R_X \times R_Y$.
3. $\sum_i \sum_j p(x_i, y_j) = 1$.

The notion of a joint PMF also extends naturally to a joint CDF:

[2)] Obviously, both RVs are not independent of each other.

**Definition 1.2** Let $X$,$Y$ be two discrete RVs on a common sample space $\Omega$. The **(joint) CDF** of $(X, Y)$ is a function $F : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ such that

$$F(s,t) = P(X \leq s, Y \leq t)\,,$$

for all $s, t \in \mathbb{R}$.

The next definition is somewhat new. Therefore, we would like to motivate it first. Let us assume that we are given two discrete RVs with their joint PMF. We might ask ourselves, what would be the PMF of each individual RV. Indeed, we can recover these PMFs from the joint PMF as the *marginal PMF*:

**Definition 1.3** Let $X$,$Y$ be two discrete RVs on a common sample space $\Omega$ with joint PMF $p$. The **marginal PMF of** $X$ is given by

$$p_X(x) = \sum_{y \in R_Y} p(x,y)\,,$$

and similarly the **marginal PMF of** $Y$ is given by

$$p_Y(y) = \sum_{x \in R_X} p(x,y)\,.$$

We give an example

**Example 1.2** (Marginal PMFs) Let a chance experiment of two rolls of a dice be considered. We introduce the random variables $X$ and $Y$, that give the two results from the dice, such that $X \geq Y$. Hence, $X$ always gives the larger result and $Y$ gives the equal/smaller result. With a little bit of work, we can compute the joint PMF of both RVs:

| $X \setminus Y$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | $\frac{1}{36}$ | 0 | 0 | 0 | 0 | 0 |
| 2 | $\frac{1}{18}$ | $\frac{1}{36}$ | 0 | 0 | 0 | 0 |
| 3 | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{36}$ | 0 | 0 | 0 |
| 4 | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{36}$ | 0 | 0 |
| 5 | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{36}$ | 0 |
| 6 | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{18}$ | $\frac{1}{36}$ |

We can recover the individual PMFs for $X$ and $Y$ by evaluating the marginal PMFs $p_X(x) = \sum_{y \in R_Y} p(x,y)$ and $p_Y(y) = \sum_{x \in R_X} p(x,y)$. Limiting us to the PMF for $X$, we

obtain:

$$\begin{aligned}
P_X(1) &= \frac{1}{36} \\
P_X(2) &= \frac{1}{18} + \frac{1}{36} = \frac{1}{12} \\
P_X(3) &= \frac{1}{18} + \frac{1}{18} + \frac{1}{36} = \frac{5}{36} \\
P_X(4) &= \frac{1}{18} + \frac{1}{18} + \frac{1}{18} + \frac{1}{36} = \frac{7}{36} \\
P_X(5) &= \frac{1}{18} + \frac{1}{18} + \frac{1}{18} + \frac{1}{18} + \frac{1}{36} = \frac{1}{4} \\
P_X(6) &= \frac{1}{18} + \frac{1}{18} + \frac{1}{18} + \frac{1}{18} + \frac{1}{18} + \frac{1}{36} = \frac{11}{36}
\end{aligned}$$

△

Next, we would like to consider in which sense the *expectation* might be influenced by having more than one RV. There is no "joint expectation". However, it can happen that we want to compute the expectation / mean of a function in e.g. two random variables. It is easy to show that for such a case, it holds

**Theorem 1.1** Let $(X, Y)$ be discrete RVs with joint PMF $p$, and let further $g : R_X \times R_Y \to \mathbb{R}$ be given. We can compute the expectation value / expectation / mean of $g(X, Y)$, if it exists, by

$$\mathrm{E}\left(g(X,Y)\right) = \sum_x \sum_y g(x,y)p(x,y)\,.$$

Existence is guaranteed, if $\sum_x \sum_y |g(x,y)|p(x,y) < \infty$.

**Example 1.3** We continue the previous example, where $X$ was the larger and $Y$ was the smaller value in two independent rolls of a dice. Using the joint PMF, we can compute the mean of the sum of both values by setting $g(X, Y) := X + Y$ in the just given theorem and obtain

$$\mathrm{E}\left(X+Y\right) = \sum_{x \in R_X} \sum_{y \in R_Y} (x+y)p(x,y)\,.$$

We leave it to the reader to explicitly evaluate the term. △

In the previous chapter, we have seen the concept of a *conditional probability*. We can extend this concept to distributions/PMFs and expectations between two random variables:

**Definition 1.4** Let $(X, Y)$ be discrete RVs with joint PMF $p$. We define the **conditional distribution of $X$ given $Y = y$** by

$$p(x|y) = P(X = x|Y = y) = \frac{p(x, y)}{p_Y(y)}\,.$$

Furthermore, we define the **conditional expectation of $X$ given $Y = y$** by

$$\mathrm{E}\,(X|y) = \mathrm{E}\,(X|Y = y) = \sum_x x\, p(x|y) = \frac{\sum_x x\, p(x, y)}{p_Y(y)} = \frac{\sum_x x\, p(x, y)}{\sum_x p(x, y)}\,.$$

Let us study what we can do with a conditional expectation.

**Example 1.4** We continue the previous example of two independent throws of a dice and the RVs $X$, $Y$, of the outcomes with $X \geq Y$.

We want to know what is the expected larger outcome, if the smaller outcome was 2. This is nothing else than the conditional expectation of $X$ given $Y = 2$, i.e. $\mathrm{E}\,(X|Y = 2)$. To compute this quantity, we first use the definition and obtain

$$\mathrm{E}\,(X|Y = 2) = \sum_{x \in R_x} x\, p(x|2)\,.$$

Hence, we have to compute the conditional PMF $p(x|2)$ for all $x$:

$$\begin{aligned}
P(1|2) &= P(X = 1|Y = 2) = \frac{p(1, 2)}{p_Y(2)} = \frac{0}{\frac{1}{4}} = 0 \\
P(2|2) &= P(X = 2|Y = 2) = \frac{p(2, 2)}{p_Y(2)} = \frac{\frac{1}{36}}{\frac{1}{4}} = \frac{1}{9} \\
P(3|2) &= P(X = 3|Y = 2) = \frac{p(3, 2)}{p_Y(2)} = \frac{\frac{1}{18}}{\frac{1}{4}} = \frac{2}{9} \\
P(4|2) &= P(X = 4|Y = 2) = \frac{p(4, 2)}{p_Y(2)} = \frac{\frac{1}{18}}{\frac{1}{4}} = \frac{2}{9} \\
P(5|2) &= P(X = 5|Y = 2) = \frac{p(5, 2)}{p_Y(2)} = \frac{\frac{1}{18}}{\frac{1}{4}} = \frac{2}{9} \\
P(6|2) &= P(X = 6|Y = 2) = \frac{p(6, 2)}{p_Y(2)} = \frac{\frac{1}{18}}{\frac{1}{4}} = \frac{2}{9}
\end{aligned}$$

Finally, we evaluate the conditional expectation as

$$\mathrm{E}\,(X|Y = 2) = 1 \cdot 0 + 2 \cdot \frac{1}{9} + 3 \cdot \frac{2}{9} + 4 \cdot \frac{2}{9} + 5 \cdot \frac{2}{9} + 6 \cdot \frac{2}{9} = \frac{38}{9} \approx 4.2\,.$$

△

Combining Theorem 1.1 and Definition 1.4, we obtain the following simple rules for conditional expectations of functions of RVs.

**Lemma 1.2** Let $X, Y$ be discrete RVs on a common sample space and $g : R_Y \to \mathbb{R}$ some function. It holds

1. $\mathrm{E}\,(g(Y)|Y = y) = g(y)$ for all $y \in R_Y$.
2. $\mathrm{E}\,(Xg(Y)|Y = y) = g(y)\,\mathrm{E}\,(X|Y = y)$ for all $y \in R_Y$.

The definition of independence for two events naturally extends to random variables:

**Definition 1.5** Let $X, Y$ be discrete RVs on a common sample space. $X$ and $Y$ are **independent**, if

$$P(X \leq x, Y \leq y) = P(X \leq x)P(Y \leq y), \quad \text{for all } x, y \in \mathbb{R}\,.$$

We give an example:

**Example 1.5** Looking again at the chance experiment of two (independent) coin tosses, we define the two random variables $X : \Omega \to \{0, 1\}$ and $Y : \Omega \to \{0, 1\}$, with

$$X(\omega) = \begin{cases} 0 & \text{if } \omega \in \{HH, HT\} \\ 1 & \text{if } \omega \in \{TH, TT\} \end{cases}, \quad Y(\omega) = \begin{cases} 0 & \text{if } \omega \in \{HH, TH\} \\ 1 & \text{if } \omega \in \{HT, TT\} \end{cases}.$$

We observe that $X$ gives a 0 if the first toss is a head and a 1 if the first toss is a tail. The RV $Y$ does the same for the second coin toss.

Intuitively, since the coin tosses are independent and the RVs only refer to the individual tosses, we would claim that $X$ and $Y$ are independent. Let us check this.

First, the joint PMF of both RVs is

$$p(0, 0) = p(0, 1) = p(1, 0) = p(1, 1) = \frac{1}{4}\,.$$

Then we evaluate the joint CDF as

$$\begin{aligned} P(X \leq 0, Y \leq 0) &= \frac{1}{4} \\ P(X \leq 0, Y \leq 1) &= \frac{1}{2} \\ P(X \leq 1, Y \leq 0) &= \frac{1}{2} \\ P(X \leq 1, Y \leq 1) &= \frac{1}{1} \end{aligned}$$

Identically, we evaluate the products of the individual CDFs as

$$\begin{aligned} P(X \leq 0)P(Y \leq 0) &= \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} \\ P(X \leq 0)P(Y \leq 1) &= \frac{1}{2} \cdot 1 = \frac{1}{2} \\ P(X \leq 1)P(Y \leq 0) &= 1 \cdot \frac{1}{2} = \frac{1}{2} \\ P(X \leq 1)P(Y \leq 1) &= 1 \cdot 1 = 1 \end{aligned}$$

As expected, the condition for the independence of the RVs is fulfilled. △

The following lemma gives us some other ways to express the independence of two RVs:

**Lemma 1.3** Let $X, Y$ be discrete RVs on a common sample space. The following four conditions are equivalent:

1. $p(x|y) = p_X(x)$, for all $x, y \in \mathbb{R}$ such that $p_Y(y) > 0$.
2. $p(y|x) = p_Y(y)$, for all $x, y \in \mathbb{R}$ such that $p_X(x) > 0$.
3. $p(x, y) = p_X(x)p_Y(y)$, for all $x, y \in \mathbb{R}$.
4. $P(X \leq x, Y \leq y) = P(X \leq x)P(Y \leq y)$, for all $x, y \in \mathbb{R}$.

While intuitively clear, it is sill interesting to observe that independence of RVs also implies no effect of the one RV to the other RV in the conditional mean:

**Lemma 1.4** Let $X, Y$ be discrete, independent RVs on a common sample space. Moreover, we are given an arbitrary function $g : R_X \to \mathbb{R}$ and an arbitrary $y \in Y$. Under the assumption of existence, it holds:

$$\mathrm{E}\left(g(X)|Y = y\right) = \mathrm{E}\left(g(X)\right) .$$

The notion of a conditional expectation can be easily carried over to a conditional variance by

**Definition 1.6** Let $(X, Y)$ be discrete RVs with joint PMF $p$. We set $\mu_X(y) = E(X|Y = y)$. Then the **conditional variance of $X$ given $Y = y$** is defined to be

$$\mathrm{Var}\left(X|Y = y\right) = \mathrm{E}\left((X - \mu_X(y))^2|Y = y\right) = \sum_x (x - \mu_X(y))^2 p(x|y) .$$

A statistical quantity that we did not see before is the *covariance*, given by

**Definition 1.7** Let $X, Y$ be discrete RVs on a common sample space, such that $\mathrm{E}(XY)$, $\mathrm{E}(X)$, $\mathrm{E}(Y)$ exist. We define the **covariance** of $X$ and $Y$ by

$$\mathrm{Cov}(X,Y) = \mathrm{E}(XY) - \mathrm{E}(X)\,\mathrm{E}(Y) = \mathrm{E}\left((X - \mathrm{E}(X))(Y - \mathrm{E}(Y))\right) .$$

In some sense it is the extension of the idea of a variance to (in this case) two random variables. We give an example:

**Example 1.6** We continue the last two coin toss example with RVs $X$ and $Y$. We compute their covariance as

$$\begin{aligned}\mathrm{Cov}(X,Y) &= \mathrm{E}\left((X-\frac{1}{2})\left(Y-\frac{1}{2}\right)\right) = \sum_{x=0}^{1}\sum_{y=0}^{1}\left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right)p(x,y)\\ &= \frac{1}{4}\sum_{x=0}^{1}\sum_{y=0}^{1}\left(x-\frac{1}{2}\right)\left(y-\frac{1}{2}\right) = \frac{1}{4}\left(\frac{1}{4}-\frac{1}{4}-\frac{1}{4}+\frac{1}{4}\right) = 0\end{aligned}$$

△

How to interpret the covariance?

**Remark** (Interpretation of the covariance) The covariance is a quantity that is giving an indication whether two RVs tend to increase or decrease together. To give an example, roughly speaking, if for two RVs $X$ and $Y$ a larger $X$ causes a larger $Y$, it might result in a positive covariance. △

Nevertheless, the covariance is just a rather vague indicator since it is not normalized. This is improved by the *correlation*.

**Definition 1.8** Let $X, Y$ be discrete RVs on a common sample space, such that $\mathrm{Var}(X)$ and $\mathrm{Var}(Y)$ are finite and non-zero. We define the **correlation** of $X$ and $Y$ by

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)}\sqrt{\mathrm{Var}(Y)}} .$$

The correlation is the covariance scaled to values in $[-1, 1]$.

In the previous example for the covariance, we have seen a covariance value of 0. The following theorem summarizes some properties of the covariance and the correlation.

**Theorem 1.2** Let $X, Y$ be discrete RVs on a common sample space. Under the assumption of existence of all involved quantities it holds:

- $\mathrm{Cov}(X, c) = 0$ for all constants $c \in \mathbb{R}$.
- $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$.
- If $X, Y$ are also independent, it holds $\mathrm{Cov}(X, Y) = \mathrm{Corr}(X, Y) = 0$.
- $-1 \leq \mathrm{Corr}(X, Y) \leq 1$.
- $\mathrm{Corr}(X, Y) = 1$ iff it holds $P(Y = a + bX) = 1$ for $a, b \in \mathbb{R}$, $b > 0$.
- $\mathrm{Corr}(X, Y) = -1$ iff it holds $P(Y = a + bX) = 1$ for $a, b \in \mathbb{R}$, $b < 0$.

From this theorem, we now understand that the independence of $X$ and $Y$ in the example implied the covariance of zero.

With this, we have introduced all necessary probability notions for the case of two discrete random variables. However, as mentioned initially, we will even have to talk about *vectors* of random variables, i.e. interactions of $n$ random variables. Most definitions and results easily carry over to this case. Therefore, we skip most of them and only briefly give the most important definitions as examples.
We start by a *joint PMF* for $n$ RVs:

**Definition 1.9** Let $X_1, \ldots, X_n$ be discrete RVs on a common sample space $\Omega$. We call the function $p : \mathbb{R}^n \to \mathbb{R}$ **joint PMF**, of $(X_1, \ldots, X_n)$, if it holds

$$p(x_1, \ldots, x_n) = P(X_1 = x_1, \ldots, X_n = x_n)$$

We could also have called this a PMF for a *random vector* $\boldsymbol{X} = (X_1, \ldots, X_n)$, where

$$p(\boldsymbol{x}) = p(x_1, \ldots, x_n) = P(X_1 = x_1, \ldots, X_n = x_n) = P(\boldsymbol{X} = \boldsymbol{x})\,.$$

This is just a notational convention.
Similarly, we state

**Lemma 1.5** Let $X_1, \ldots, X_n$ be discrete RVs on $\Omega$, with each $X_i$ taking values in some countable set $R_{X_i}$. Let further $p$ be their joint PMF. It holds:

1. $p(x_1, \ldots, x_n) = 0$,
   for all $(x_1, \ldots, x_n) \notin R_{X_1} \times \cdots \times R_{X_n}$.
2. $p(x_1, \ldots, x_n) \geq 0$,
   for all $(x_1, \ldots, x_n) \in R_{X_1} \times \cdots \times R_{X_n}$
3. $\sum_{x_1 \in R_{X_1}, \ldots, x_n \in R_{X_n}} p(x_1, \ldots, x_n) = 1$.

And the *joint CDF* is given by

**Definition 1.10** Let $X_1, \dots, X_n$ be discrete RVs on a common sample space $\Omega$. The **(joint) CDF** of $(X_1, \dots, X_n)$ is a function $F : \mathbb{R}^n \to \mathbb{R}$ such that

$$F(x_1, \dots, x_n) = P(X_1 \leq x_1, \dots, X_n \leq x_n)\,,$$

for all $x_1, \dots, x_n \in \mathbb{R}$.

As discussed before, the remaining definitions and results easily extend to the case of $n$ RVs or an $n$-dimensional random vector. As an example, we mention that the covariance of two vectors of random vectors/variables $\boldsymbol{X} = (X_1, \dots, X_n)$, $\boldsymbol{Y} = (Y_1, \dots, Y_m)$ becomes a *covariance matrix*

$$C = \begin{pmatrix} \operatorname{Cov}(X_1, Y_1) & \dots & \operatorname{Cov}(X_1, Y_m) \\ \vdots & \ddots & \vdots \\ \operatorname{Cov}(X_n, Y_1) & \dots & \operatorname{Cov}(X_n, Y_m) \end{pmatrix}.$$

## 1.2 Continuous random variables

The discussion of discrete multivariate random variables is an easy start into the field, as the new concepts are introduced without using truly new mathematical tools. This is a little bit different for continuous multivariate random variables. Clearly, the rough intuition of replacing the finite or countable sums by integrals still holds. However, we will suddenly have to deal with multivariate integrals, that might not have been seen by all readers, before. This would at least require some advanced calculus background.

There is no easy way to overcome this problem, as we do not intend to give a full advanced calculus introduction, here. In some sense, we will have to work with a lot of intuition. Having said that, we come to our first definition:

**Definition 1.11** Let $X = (X_1, \dots, X_n)$ be an $n$-dimensional random vector of continuous RVs $X_i : \Omega \to \mathbb{R}$. We call the function $\rho : \mathbb{R}^n \to \mathbb{R}$ **joint density**, short **density**, of $X$, if it holds

$$\begin{aligned} &P(a_1 \leq X_1 \leq b_1, \dots, a_n \leq X_n \leq b_n) \\ &\quad = \int_{a_n}^{b_n} \cdots \int_{a_1}^{b_1} \rho(x_1, \dots, x_n) dx_1 \cdots dx_n \end{aligned}$$

How to interprete this integral?

**Remark** (Multivariate integrals) The above integral is a classical Riemann integral in higher dimensions. Let us have a look of the case of $n = 2$. Then, the integral would look like

$$\int_{a_2}^{b^2} \int_{a_1}^{b^1} \rho(x_1, x_2)\, dx_1\, dx_2\,.$$

Let us replace the density by some (non-density) easy function:

$$\int_{a_2}^{b^2} \int_{a_1}^{b^1} (x_1 + x_2)\, dx_1\, dx_2\,.$$

How to compute this? This is rather simple: From the inside to the outside. In our mind, we should put some brackets into the formula and start to evaluate it from the inside to the outside:

$$\begin{aligned}
\int_{a_2}^{b^2} \int_{a_1}^{b^1} (x_1 + x_2)\, dx_1\, dx_2 &= \int_{a_2}^{b^2} \left( \int_{a_1}^{b^1} (x_1 + x_2)\, dx_1 \right) dx_2 \\
&= \int_{a_2}^{b^2} \left( \left[ \frac{1}{2} x_1^2 + x_2\, x_1 \right]_{a_1}^{b_1} \right) dx_2 \\
&= \int_{a_2}^{b^2} \left( \frac{1}{2} b_1^2 + b_1 x_2 - \frac{1}{2} a_1^2 - a_1 x_2 \right) dx_2
\end{aligned}$$

So far, we have evaluated the inner integral. Notice, that we treated $x_2$ as a constant in the evaluation of the inner integral. The second step would be to evaluate the outer integral. However, this is just a univariate integral, thus known to the reader. Hence, we skip this.

Coming back to the $n$-dimensional integral

$$\int_{a_n}^{b_n} \cdots \int_{a_1}^{b_1} \rho(x_1, \ldots, x_n) dx_1 \cdots dx_n\,,$$

it would be the exact same story to evaluate that one. We would start with the "most inside" integral over $x_1$. Then, iteratively we would go for $x_2$, $x_3$, etc. △

We give an example for a multivariate density.

**Example 1.7** We consider a three-dimensional random variable / random vector $\boldsymbol{X} = (X_1, X_2, X_3)$, which is built from three independent, identically distributed random variables $X_i$ that follow the the normal distribution $\mathcal{N}(\mu, \sigma^2)$. As we will see soon, the density of the resulting vector is a product of the densities of the individual vectors (since they are independent). Therefore, we get

$$\begin{aligned}
\rho(x_1, x_2, x_3) &= \frac{1}{\sigma\sqrt{2\pi}} e^{-(x_1-\mu)^2/(2\sigma^2)} \frac{1}{\sigma\sqrt{2\pi}} e^{-(x_2-\mu)^2/(2\sigma^2)} \frac{1}{\sigma\sqrt{2\pi}} e^{-(x_3-\mu)^2/(2\sigma^2)} \\
&= \left( \frac{1}{\sigma\sqrt{2\pi}} \right)^3 e^{-\sum_{i=1}^3 (x_i-\mu)^2/(2\sigma^2)}
\end{aligned}$$

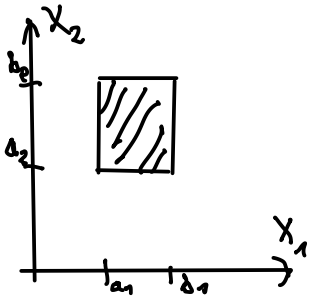
△

In some sense, the definition of a joint density and the resulting probability for multivariate continuous RVs limits us a little bit:

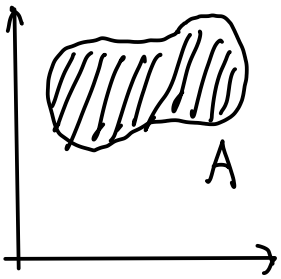
**Remark** (Universality of our definitions) To keep things readable for a broad audience, we simplify the definitions for multivariate RVs a little bit. Right-now, the probabilities induced by the densities are only considered over products of intervals, e.g. for the two-dimensional case we can write down the probability

$$P(a_1 \leq X_1 \leq b_1, a_2 \leq X_2 \leq b_2) = \int_{a_2}^{b^2} \int_{a_1}^{b^1} \rho(x_1, x_2)\, dx_1\, dx_2\,,$$

where we limit the boundaries to intervals. Visually, in this example, we would compute the integral over a rectangle:




However, in practice, there might be reasons to consider more complex events that are, keeping the two-dimensional example, induced by subsets $A \subseteq \mathbb{R}^2$. Hence, we would have to compute the integral over a more complex domain $A$:




This however, could not be easily described by an "iterative" product integral as we have seen it in the previous remark. Instead, the integral could be written down like this

$$P(X_1, X_2) = \int_A \rho(x_1, x_2) d(x_1, x_2)\,.$$

In context of probability (theory), we would usually switch to the *Lebesgue integral* notion and would have to work with *measures*, etc.

We skip all this, here, but the interested reader should keep in mind that we might not cover all meaningful situations by our definitions. △

We come to a first lemma that naturally extends the properties of densities from the univariate to the multivariate case.

**Lemma 1.6** Let $X = (X_1, \dots, X_n)$ be an $n$-dimensional random vector of continuous RVs $X_i$ with joint density $\rho$. It holds:

1. $\rho(x_1, \dots, x_n) \geq 0$, for all $(x_1, \dots, x_n) \in \mathbb{R}^n$.
2. $\int_{\mathbb{R}^n} \rho(x_1, \dots, x_n) dx_1 \cdots dx_n = 1$.

We have been lazy with the notation here, and replaced the $n$-fold integral symbol by a single integral symbol over $R^n$.

Similarly, we immediately get the *joint CDF*.

**Definition 1.12** Let $X = (X_1, \dots, X_n)$ be an $n$-dimensional random vector of continuous RVs on a common sample space $\Omega$. The **(joint) CDF** of $X$ is a function $F : \mathbb{R}^n \to \mathbb{R}$ such that

$$F(x_1, \dots, x_n) = \int_{-\infty}^{x_n} \cdots \int_{-\infty}^{x_1} \rho(t_1, \dots, t_n) dt_1 \cdots dt_n \,,$$

for all $t_1, \dots, t_n \in \mathbb{R}$.

Similar to the marginal PMF, we can introduce a marginal joint density by

**Definition 1.13** Let $X = (X_1, \dots, X_n)$ be an $n$-dimensional random vector of continuous RVs with joint density $\rho$. We are further given $1 \leq p < n$. The **marginal joint density** of $(X_1, \dots, X_p)$ is defined by

$$\rho_{1,2,\dots,p}(x_1, \dots, x_p) = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \rho(x_1, \dots, x_n) dx_{p+1} \cdots dx_n \,.$$

On first sight, this definition might be a bit overwhelming. Let us therefore, for the sake of clarification, go to the $n = 2$ case. In this case, we have marginal densities

$$\rho_1(x_1) = \int_{-\infty}^{\infty} \rho(x_1, x_2) dx_2 \quad \text{and} \quad \rho_2(x_2) = \int_{-\infty}^{\infty} \rho(x_1, x_2) dx_1 \,.$$

We could have also called the first marginal density $\rho_{X_1}$ and the second one $\rho_{X_2}$. This definition is very similar to the marginal PMF, but in the continuous case, hence we "*integrate* out" the random variable that we are not interested in.

The fully general case presented in Definition 1.13 simply allows for arbitrary marinalizations of subsets of the set of random variables.

We consider an example.

**Example 1.8** Let $X_1$ and $X_2$ be two independent, identically distributed random variables that follow each the uniform distribution $\mathcal{U}[a,b]$. On the domain $[a,b]\times[a,b]$ their joint density is given by

$$\rho(x,y) = \frac{1}{b-a}\frac{1}{b-a}\,.$$

We are interested to compute the marginal density $\rho_1$ on $[a,b]$. It is given by[3)]

$$\rho_1(x_1) = \int_a^b \rho(x_1,x_2)dx_2 = \left[\frac{1}{(b-a)^2}x_2\right]_a^b = \frac{b-a}{(b-a)^2} = \frac{1}{b-a}\,.$$

△

In previous examples, we already considered the case of independent RVs that form one random vector. We already stated, that their joint density would be given as the product of the univariate densities. Now, we formalize this in

**Lemma 1.7** Let $X=(X_1,\ldots,X_n)$ be an $n$-dimensional random vector of continuous RVs with joint density $\rho$. $X_1,\ldots,X_n$ are independent iff it holds

$$\rho(x_1,\ldots,x_n) = \prod_{i=1}^n \rho_i(x_i)\,,$$

with $\rho_i(x_i)$ the marginal density of $X_i$.

As seen for the multivariate discrete case, we can compute the mean of a function in a vector of random variables via

**Theorem 1.3** Let $X=(X_1,\ldots,X_n)$ be an $n$-dimensional random vector of continuous RVs with joint density $\rho$. We are further given a function $g:\mathbb{R}^n\to\mathbb{R}$. We can compute the **expectation value / expectation / mean** of $g(X,Y)$, if it exists, by

$$\mathrm{E}\left(g(X_1,\ldots,X_n)\right) = \int_{\mathbb{R}^n} g(x_1,\ldots,x_n)\rho(x_1,\ldots,x_n)dx_1\cdots dx_n\,.$$

Existence is guaranteed, if $\int_{\mathbb{R}^n}|g(x_1,\ldots,x_n)|p(x_1,\ldots,x_n)dx_1\cdots dx_n<\infty$.

A typical example would be the case of $n=2$ and $g(X_1,X_2)=X_1+X_2$, i.e. the sum of two RVs.

Next, we introduce conditional densities and expectations similar to the discrete case.

[3)] Since we know that the joint density is zero outside of $[a,b]\times[a,b]$, we can skip over some special cases and simply limit the integral to the domain $[a,b]$.

**Definition 1.14** Let $(X, Y)$ be continuous RVs with joint density $\rho$. We define the **conditional density of $X$ given $Y = y$** by

$$\rho(x|y) = \rho(x|Y = y) = \frac{\rho(x, y)}{\rho_Y(y)},$$

for all $y$ such that $\rho_Y(y) > 0$. Furthermore, we define the **conditional expectation of $X$ given $Y = y$** by

$$\mathrm{E}\,(X|y) = \mathrm{E}\,(X|Y = y) = \int_{-\infty}^{\infty} x\,\rho(x|y)dx = \frac{\int_{-\infty}^{\infty} x\,\rho(x, y)dx}{\int_{-\infty}^{\infty} \rho(x, y)dx},$$

for all $y$ such that $\rho_Y(y) > 0$.

At the beginning of the previous chapter, we also formulated Bayes' theorem. We can adapt this to multivariate densities as follows:

**Theorem 1.4** (Bayes) Let $(X, Y)$ be continuous RVs with joint density $\rho$. For all $x, y \in \mathbb{R}$, such that $\rho_X(x), \rho_Y(y) > 0$, it holds

$$\rho(y|x) = \frac{\rho(x|y)\rho_Y(y)}{\rho_X(x)}.$$

Similar to the discrete case, we introduce the *conditional variance*.

**Definition 1.15** Let $(X, Y)$ be continuous RVs with joint density $\rho$. We set $\mu_X(y) = E(X|y)$. Then the **conditional variance of $X$ given $Y = y$** is defined to be

$$\mathrm{Var}\,(X|y) = \mathrm{Var}\,(X|Y = y) = \mathrm{E}\left((X - \mu_X(y))^2|Y = y\right) = \frac{\int_{-\infty}^{\infty}(x - \mu_X(y))^2\rho(x, y)dx}{\int_{-\infty}^{\infty} \rho(x, y)dx},$$

for all $y \in \mathbb{R}$ such that $\rho_Y(y) > 0$.

We see here, again, that all results of the discrete case naturally extend to the continuous case. Therefore, we will skip over most of them and just remind the reader that the *covariance* is given by

**Definition 1.16** Let $X, Y$ be continuous RVs on a common sample space, such that $\mathrm{E}(XY)$, $\mathrm{E}(X)$, $\mathrm{E}(Y)$ exist. We define the **covariance** of $X$ and $Y$ by

$$\mathrm{Cov}(X,Y) = \mathrm{E}(XY) - \mathrm{E}(X)\,\mathrm{E}(Y) = \mathrm{E}((X - \mathrm{E}(X))(Y - \mathrm{E}(Y))) \ .$$

and that the *correlation* is given by

**Definition 1.17** Let $X, Y$ be continuous RVs on a common sample space, such that $\mathrm{Var}(X)$ and $\mathrm{Var}(Y)$ are finite. We define the **correlation** of $X$ and $Y$ by

$$\mathrm{Corr}(X,Y) = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{Var}(X)}\sqrt{\mathrm{Var}(Y)}} \ .$$

Finally, the usual rules for both quantities hold:

**Theorem 1.5** Let $X, Y$ be continuous RVs on a common sample space. Under the assumption of existence of all involved quantities it holds:

- $\mathrm{Cov}(X, c) = 0$ for all constants $c \in \mathbb{R}$.
- $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$.
- If $X, Y$ are also independent, it holds $\mathrm{Cov}(X,Y) = \mathrm{Corr}(X,Y) = 0$.
- $-1 \leq \mathrm{Corr}(X,Y) \leq 1$.
- $\mathrm{Corr}(X,Y) = 1$ iff it holds $P(Y = a + bX) = 1$ for $a, b \in \mathbb{R}$, $b > 0$.
- $\mathrm{Corr}(X,Y) = -1$ iff it holds $P(Y = a + bX) = 1$ for $a, b \in \mathbb{R}$, $b < 0$.

## 1.3 Important continuous multivariate distributions

We quickly want to have a look at two important multivariate distributions, as we will come across these distributions in our discussion about machine learning.
The *multivariate uniform distribution* is given by

**Definition 1.18** Let $X = (X_1, \ldots, X_n)$ be a vector of continuous (independent) RVs, where each individual random variable $X_i$ has density

$$\rho^{(i)}(x_i) = \begin{cases} \frac{1}{b_i - a_i}, & a_i \leq x_i \leq b_i, \\ 0, & \text{else}, \end{cases}$$

and $X$ has the density

$$\rho(x) = \prod_{i=1}^{n} \rho^{(i)} .$$

In this case, we say that $X$ has the **multivariate uniform distribution** over the domain $[a_1, b_1] \times \cdots \times [a_n, b_n]$, denoting it by

$$X \sim \mathcal{U}(\boldsymbol{a}, \boldsymbol{b}) .$$

It simply extends the idea of the uniform distribution in the univariate case to higher dimensions. The expectation of such a random variable is given by

**Lemma 1.8** The expectation of an RV $X \sim \mathcal{U}(\boldsymbol{a}, \boldsymbol{b})$ is

$$\mathrm{E}\,(X) = \prod_{i=1}^{n} \frac{a_i + b_i}{2} .$$

Moreover, we have a look at the *multivariate normal distribution*:

**Definition 1.19** Let $X = (X_1, \ldots, X_n)$ be a vector of continuous RVs. It is said to follow the **multivariate normal / Gaussian distribution** with parameters containing the vector $\boldsymbol{\mu} \in \mathbb{R}^n$ and the positive definite matrix $\pm \in \mathbb{R}^{n \times n}$, if its density function $\rho$ is given by

$$\rho(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^n \det \Sigma}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \pm^{-1} (\boldsymbol{x}-\boldsymbol{\mu})} .$$

We denote this by $X \sim \mathcal{N}(\boldsymbol{\mu}, \pm)$.

**Remark** Indeed, the random variables $X_1, \ldots, X_n$ are not independent by construction. Therefore, we also give instead of the variance $\sigma^2$ a (covariance) matrix $\pm$. If we want these random variables to be independent, we chose $\pm$ as a diagonal matrix. △

Finally, we give the expectation and the covariance matrix of a random variable that follows the multivariate normal distribution:

**Lemma 1.9** The expectation and covariance matrix of a RV $X \sim \mathcal{N}(\boldsymbol{\mu}, \pm)$ are

$$\mathrm{E}\,(\boldsymbol{X}) = \boldsymbol{\mu}, \quad \mathrm{Cov}\,(\boldsymbol{X}) = \pm .$$

This concludes our rather quick introduction into multivariate probability. We will see pretty much all concepts introduced here in the upcoming discussion of *machine learning*.

# Part II.

# Traditional learning

# 2. Introduction

Over the recent years, *machine learning* has received a tremendous attention by research, industry and media, such that the reader will most likely have come across some general ideas and notions from the field of machine learning. Depending on the source of information, a more practical, a more theoretical or a more intuition-based view of machine learning might have been developed by the reader.

In this chapter, we will start with a short motivation, in which we try to catch the reader form where she/he is. We will then become rather practical and will give an overview of the typical machine learning pipeline, i.e. going from the first piece of data to the fully optimized model that is ready to be evaluated. This discussion will be the starting point for this work. Pretty much everything that we present afterwards can be somewhat associated to that pipeline. The rest of the chapter is concerned with the first part of the pipeline, in which we model, pre-process and/or represent data. Moreover, we will give a decent number of examples.

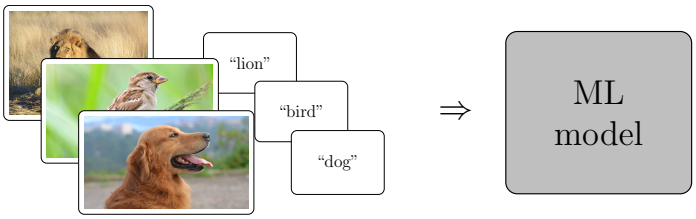
## 2.1 Motivation

The average reader, being asked for examples on "What is machine learning?", might come up with a view as broad as self-driving cars, face detection or speech recognition. The good news is, all of this has something to do with machine learning. The bad news is that these are overly complicated examples, which will not help us to come to the core of "What is machine learning?".

Therefore, we will start by picking a very simple example, that is most likely known to everyone: *Image classification*. In image classification, we are given an image and ask a machine learning model (ignoring that we don't know what this is), to assign that picture to a "class". But wait! How can that model know how to classify an image? To answer this question, we need to go a step back and need to start from another perspective.

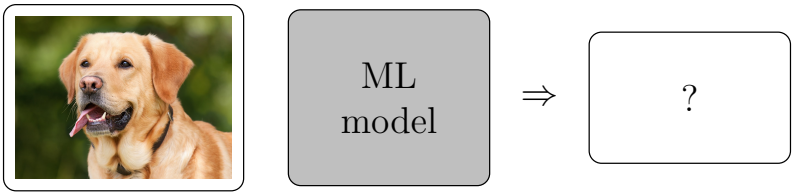
Our new starting point is a pile of images, where we have assigned to each image a label or class:


In our case, we only have three images of animals and have assigned to them the type of animal that is found on the picture. Then we take this input data, and certainly many more pictures with many more labels, and use them as an input to some algorithm. That algorithm uses the pictures to *train* a machine learning model:




For now we do not care so much what *training* means but just accept that, at the end of this *training phase*, we have some operational algorithm that has associated data, containing information extracted from our originally given data.

Then finally, we come to what we started with: We now have that algorithm and data, which we might want to call *machine learning model*. And now, we only give an input picture, without associated label, to that model and hope for a meaningful answer:




In this example, our intuition would tell us that the model should give us the label "dog" as answer. We might want to call that last step *prediction/inference*.

Most likely, most of the readers will have seen this or a similar application. This task is called *classification*, where we used images and labels as inputs and outputs.

More generally, we distinguish in machine learning between *supervised machine learning* and *unsupervised machine learning*.[1)] In supervised machine learning, we start from many *training samples / examples* that associate to an *input* some *output*. Then, the task is to use these samples of data to infer a model that allows to predict for an unseen input an output that somewhat will follow the characteristics of the given training samples.

Obviously, classification is supervised machine learning. The other important task in supervised machine learning is *regression*. While classification associates to an input a categorical output, i.e. a class, a label, etc., regression associates to the input one or

[1)] Indeed, there are more flavours of machine learning, such as *semi-supervised machine learning* and others. However, we try to keep things simple, here, and will also only restrict ourselves to the discussion of these two flavors, over this whole work.

many continuous numerical values. Predicting the temperature at some location based on a machine learning model could be an example of a regression task. Summarizing supervised machine learning, we aim at finding the hidden relationship between the inputs and outputs and use this information to generate meaningful outputs for unseen inputs.

In unsupervised machine learning, we only deal with inputs. Thus in some sense, we only provide data, but give no hints on how to interpret that data. In this case, we just try to find some relationship in the not annotated data. One example of an unsupervised machine learning task is *clustering*. Here, we provide samples of inputs and ask for a "meaningful" grouping of the different samples.

Let us again pick some pictures of animals:

 ,  ,  ,  .

A meaningful grouping of these pictures could be e.g.

i.e. we might hope that the algorithm itself would be able to put the pictures in meaningful groups.[2)] Another unsupervised learning task that we will discuss is the *principle component analysis*. To cut the discussion short, this task aims at finding "main components" that "explain" the inputs.

The objective of this work is to answer the following and many more fundamental questions in machine learning:

- What types of machine learning problems / tasks exist?
- How can we model these problems in a mathematical way?
- What different models/methods exist, to solve the problems?
- Why do these methods work?
- Which algorithms are used to implement the methods?

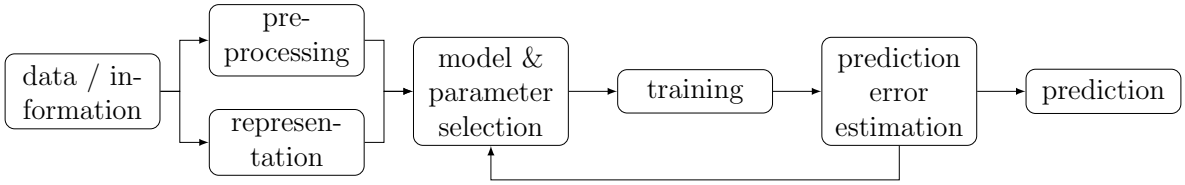
[2)] In reality, it is impossible to get such a clustering with only four pictures and still rather unlikely with many pictures. Still, we take this example to clarify the task.

- What are the differences between the methods?

We already had a very quick look on a high-level answer to the first question. All other questions will be covered to some extend in the upcoming sections and chapters.

## 2.2 Machine learning pipeline

Our main focus in this work will be on *supervised* machine learning. Therefore, it is worth to have a look at the steps of solving a supervised machine learning task. We will call the full sequence of these steps the *machine learning pipeline*:




Let us discuss all parts of this pipeline.

**Data / information** The input to our pipeline is data / information. In supervised machine learning, we will need *training data* namely, a set of input data with associated output data. Input data can be pretty much everything, including images, measurement times, geometric objects, coordinates, text, speech, molecules, etc. Output data can be as simple as class labels or vectors of numbers but can go up to, again, pictures, text, speech, etc. In Section 2.3, we will discuss, how to model input and output data in mathematical terms.

**Pre-processing** Some of the data that is used for training a machine learning model might come from measurements, surveys or experiments. In these cases, it is very likely that some entries will be errorneous, missing or have other sorts of issues. Moreover, data might not come e.g. in the most optimal value range that fits well for the training process. Therefore, we will have to deal with appropriate pre-processing of the data. Pre-processing of the data is discussed in Section 2.4.

**Representation** Another issue of our input/output data could be the kind of data that we deal with. Looking at our examples, we also had geometric objects, text or molecules, which do not follow the usual pattern that we would expect for input and output data.[3)] Therefore, we will have to transform the data into a *representation* that makes is accessible to machine learning tools. Finding good representations is a research topic by itself and will be briefly studied in Section 2.5.

[3)] As we will see, we mostly aim for having vectors of numbers as input and vectors of numbers / categories as output.

**Model & parameter selection** Given the input / output data and the research question, we easily identify the machine learning task, i.e. regression or classification. However, there are are dozens of machine learning models, *losses* and *hyper-parameters* that we can go for. (A hyper-parameter is an additional input to a machine learning model that changes some of the properties of the model. A loss characterizes the way to measure error by the model.) Selecting the "right" model with the right parameters in the first place is often not possible. So we will have to iterate the process, as seen under "Prediction error estimation". Chapters 4, 10, 11 and 12 will deal with models for regression, while chapters 8, 11 and 12 will deal with models for classification.

**Training** As soon as we have chosen the machine learning model, we will use (a subset of) the given input/output data to *train* the machine learning model. Essentially, training means that we solve a mathematical optimization/minimization problem that – for the given *training data* – provides as outcome a set of numerical parameters / coefficients such that the machine learning model describes best the training data. In Chapter 3 we will give a mathematical definition of the optimization/minimization problem that we need to solve. With each model that we introduce, we will also discuss how to solve the associated minimization problem. However, we will also dedicate the full Chapter 5 to a general set of mathematical tools on how to solve such minimization problems.

**Prediction error estimation** After having trained the machine learning model, we – in principle – could immediately use it to make predictions. However, at this stage, we need to validate the model by analysing its *prediction error* on unseen data. Depending on the outcome, we might either want to change to a completely different model or we might want to fine-tune the model. Fine-tuning is an iterative process, in which we change some of the hyper-parameters of the model and look for an improvement in the prediction error. There are even ways to automatize this fine-tuning. In Chapter 6, we will discuss the practical considerations of (prediction) error measurements for machine learning models. Chapter 7, with the bias vs. variance discussion, will provide further theoretical foundations that help us to better understand the observed errors of different models.

**Prediction** Finally, we apply the chosen and fine-tuned machine learning model to unseen data. The model might then be provisioned in a more complex application like a self-driving car, a robot, a recommender system, etc.

At this point, it is worth mentioning that *unsupervised* learning still has many important applications. However, since it is not the predominant content of this work, we will dedicate only Chapter 9 to this task. Additionally, we use Chapter **??** to discuss the influence of the dimensionality of the input/output data on machine learning tasks.

## 2.3 Modeling input and output data

To develop a mathematically rigorous approach to discuss machine learning, we need to start by a proper modeling of our data. Intuitively, we learned before that we would start in supervised learning from example *inputs* and *outputs* that help us to find a model. In unsupervised learning, we only have the example *inputs*. Let us define what is an *input* and let us further give a better notion for "examples".

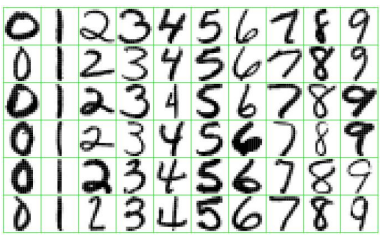
**Definition 2.1** (Input variables and their measurements)

- **Inputs / predictors / features** are vectors of $D$ *variables*, usually called $X$, composed from univariate variables $X_1, \dots, X_D$:
$$X = (X_1, \dots, X_D)^\top$$
- Inputs can be measured or preset. We denote the $i$th **measurement** or **observation** of the input $X$ as $x_i$ and collect $N$ measurements $\{x_1, \dots, x_N\}$.
- We write $X \in \mathbb{R}^D$ to indicate that $X$ can take values in $\mathbb{R}^D$, hence observations are of form $x_i \in \mathbb{R}^D$

**Remark** (Terminology) As we can see, *inputs*, can also be called *predictors* or *features*. This heavily depends on the research community, hence whether the same topic is discussed in *machine learning*, *statistical learning* or *pattern recognition*. Similar statements hold for *measurements* or *observations*. We will use all of these notions at some time, however observe that e.g. *inputs* seems to be the most modern terminology. △

To fill this definition with life, we immediately come to three examples.

**Example 2.1** (MNIST – Handwritten digit recognition) One of the early, very successful examples of a machine learning task was the recognition of handwritten digits for ZIP postal codes. There exists a widely used benchmark data set, called MNIST that we will use here to given an example for an input.
In MNIST, the *measured input*, is a vector $X \in \mathbb{R}^{784}$ of $28 \cdot 28$ variables
$$X = (X_1, \dots, X_{784})$$
describing the greyscale value of each pixel of a $28 \times 28$ resolution image of a single, centered hand-written digit. A few examples of such images are given below:



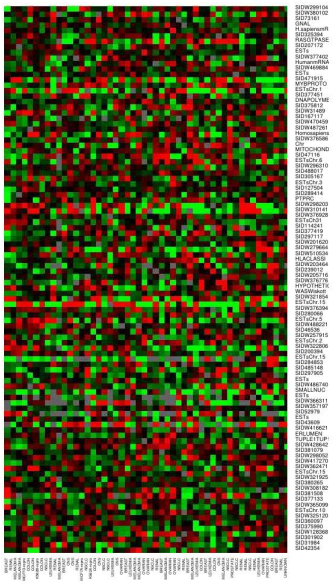
Each of these images corresponds to one *measurement* $x_i$ of the input $X$, i.e. we have $x_i \in \mathbb{R}^{768}$. The MNIST data set has a total of 60,000 such measurements. The data can be used in a classification task. △

**Example 2.2** (Energy efficiency prediction) Another good educational data set is energy efficiency data set from [10]. I collects as predictors a vector $X$ of 8 variables

$$X = (X_1, \ldots, X_8)\,,$$

where each variable corresponds to a different properties describing the shape and further properties of a house. Examples for these these properties are overall height, the roof area and the glazing area. A total of 768 measurements are contained in the data set, i.e. we have the observations $\{x_i\}_{i=1}^{768}$, $x_i \in \mathbb{R}^8$. The data set can be used in a regression task to predict the required heating and / or cooling load. △

**Example 2.3** (DNA Expression Microarrays) We also study data that has been collected from a DNA Expression Microarray. Let us first have a look at the data taken from [6]:



The picture characterizes the output of a DNA analysis device. Each row corresponds to a specific gene, with a total of 6830 genes. Each column corresponds to samples taken from tumor tissue from different locations in bodies of patients. The red color indicates that a given gene is over-expressed, i.e. very active, while the green color indicates that a given gene is under-expressed, i.e. inactive. Trying to model this data, we use the input $X \in \mathbb{R}^{6830}$, i.e.

$$X = (X_1, \ldots, X_{6830})$$

to model the level of expression of the 6830 genes in a sample. The *observations* $\{x_i\}_{i=1}^{N}$, $x_i \in \mathbb{R}^D$ are the different measured expression levels for different samples. Unsupervised machine learning could help us to answer a question like "Do we have genes with high expression for specific cancer cells?". △

For the mathematically interested reader, the above definition might not yet be very satisfactory. Essentially, we did not yet clearly state what *is* a variable $X_i$. We will discuss this soon, however, first come to the definition of *output variables* and *training data*.

**Definition 2.2** (Output variables and training data) Let an input variable $X$ be given.

- **Outputs / responses** are vectors of $K$ *variables* and can be either **quantitative** or **qualitative / discrete**.
- Quantitative outputs are called $Y$, with $Y = (Y_1, \dots, Y_K)^\top$.
- Qualitative outputs are called $G$, with $G = (G_1, \dots, G_K)^\top$ and a variable $G_j$ can only take a finite number of values.
- A set of (output) measurements is of form $\{(x_i, y_i)\}_{i=1}^N$ or $\{(x_i, g_i)\}_{i=1}^N$ (since $Y$ or $G$ depend on $X$) and is called **training data**.

As it becomes apparent from the definition, measurements of outputs / responses are only properly given, if they are immediately associated to a measurement of an input. Quantitative outputs will lead to *regression* tasks and *qualitative* outputs will lead to classification tasks.

Let us come back to our first two examples:

**Example 2.4** (MNIST) Continuing our example on handwritten digit recognition, we add to the input $X \in \mathbb{R}^{768}$ the *qualitative* or *discrete* output $G$, which is one-dimensional. It takes the values $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The measurements contained in the MNIST data set form together the *training data* $\{(x_i, g_i)\}_{i=1}^{60000}$ associating each image of a handwritten digit to the represented digit. △

**Example 2.5** (Energy efficiency) In the energy efficiency prediction data set, we had the input $X \in \mathbb{R}^8$. To this input, we associate either the one-dimensional *quantitative* output $Y \in \mathbb{R}$ of the required heating load. Hence, the data set contains the training data $\{(x_i, y_i)\}_{i=1}^{768}$. △

In supervised machine learning, we aim at finding *predictions* for outputs (constrained to inputs) for a given set of training data. The next chapter will clarify this statement. Nevertheless, for now, we at least formalize the notion of a *prediction*

**Definition 2.3** (Prediction) Let an input variable $X$ with an associated output variable $Y$ or $G$ be given. **Predictions** of $Y$ or $G$ are variables $\hat{Y}$ or $\hat{G}$ such that $\hat{Y} \approx Y$or $\hat{G} \approx G$.

In the digit recognition example, we try to find the prediction $\hat{G}$ for the digit, and in the housing price example, we try to find the prediction $\hat{Y}$ for the price.

Now, we come back to the discussion of the mathematical interpretation of a *variable* $X_i$. We intentionally kept a very general formulation, as we tend to have different interpretations of a variable. In classical statistical learning and many recent applications, we have a *probabilistic interpretation* of inputs and outputs. In some other cases, we tend to have a *deterministic interpretation* of inputs and outputs. That is, we sometimes go for the one view and sometimes go for the other view. The following two remarks outline the two interpretations and thereby complete the earlier definitions.

**Remark** (Probabilistic interpretation of inputs/outputs) The predominant interpretation of input and output *variables* is a *probabilistic* one. Following this interpretation, we start from a sample space $\Omega$ and model the **input** $X$ as a $D$-dimensional continuous random variable / vector

$$X : \Omega \to \mathbb{R}^D$$

with density $\rho(x)$. In this case, the **measurements** $\{x_1, \dots x_N\}$ are realizations of $N$ independent, identically distributed (i.i.d.) random variables with density $\rho(x)$.
Moreover, a **quantitative output** $Y$ is a $K$-dimensional continuous random variable

$$Y : \Omega \to \mathbb{R}^K$$

over the same sample space and $X$, $Y$ are connected by a joint density $\rho(x, y)$. In this case, the **training data** $\{(x_i, y_i)\}_{i=1}^N$ is a set of realizations of $N$ i.i.d. random variables with joint density $\rho(x, y)$.
Similarly, a **qualitative output** $G$ is a $K$-dimensional *discrete* random variable

$$G : \Omega \to \mathbb{R}^K$$

over the same sample space and $X$, $G$ is connected by a joint density $\rho(x, g)$. In this case, the **training data** $\{(x_i, g_i)\}_{i=1}^N$ is a set of realizations of $N$ i.i.d. random variables with joint density $\rho(x, g)$.
In either cases, we usually do not know the joint density $\rho(x, y)$ / $\rho(x, g)$. Hence, supervised machine learning implicitly aims at reconstructing that joint density from the given training data. In practice, we build a **predictor** $\hat{Y}/\hat{G}$, i.e. a random variable, from that data. The predictor is supposed to approximate the output random variable $Y/G$ that is only known from the training data. △

Some reader will have observed that we have suddenly started to mix discrete and continuous random variables when dealing with qualitative outputs. We accept this for now, and come back to this after the next remark.

**Remark** (Deterministic interpretation of inputs/outputs) Depending on the context and method of choice, we also sometimes want to go for a purely deterministic interpretation

of input and output variables. In that case we tend to identify an input space

$$\mathcal{X} \subset \mathbb{R}^D \,,$$

where the *input* $X$ is indeed only a variable $X \in \mathcal{X}$. In this context, *measurements* are just finite, deterministic subsets $\{x_1, \dots, x_N\} \subset \mathcal{X}$ from that input space.
We identify a quantitative output space

$$\mathcal{Y} \subset \mathbb{R}^K \,,$$

where a **quantitative output** is only a variable $Y \in \mathcal{Y}$. Then the **training set** is a finite set of pairs $\{(x_1, y_1), \dots, (x_N, y_N)\} \subset \mathcal{X} \times \mathcal{Y}$.
In an analogous way, we identify a qualitative output space

$$\mathcal{G} \subset \mathbb{R}^K$$

that is *finite* and where a **qualitative output** is only a variable $G \in \mathcal{G}$. Again, the **training set** is a set of pairs $\{(x_1, g_1), \dots, (x_N, g_N)\} \subset \mathcal{X} \times \mathcal{G}$.
In the deterministic interpretation, we assume that there is a (hidden) relationship / map between the sets $\mathcal{X}$ and $\mathcal{Y}$. The relationship is given via the training set. Therefore the objective is to find a predictive function $f$ such that

$$y_i \approx f(x_i) \,,$$

for all $i \in \{1, \dots, N\}$. △

Trying to summarize both remarks, we treat in the probabilistic interpretation all inputs and outputs as random variables. Here, data is given by samples form i.i.d. random variables. In the deterministic interpretation, we just talk about fixed, non-probabilistic data.

At this point, we need to talk about a small, but important mathematical detail of the probabilistic interpretation of inputs/outputs for *classification*. I that case, we assume to have a joint density $\rho(x, g)$ for $X$ and $G$. However, $X$ is a continuous RV, while $G$ is a discrete RV. We did not consider such a *mixed* case before. We can solve this by defining a *mixed joint density* by

**Definition 2.4** Let $X = (X_1, \dots X_D)$ be a vector of continuous random variables and $G = (G_1, \dots, G_K)$ be a vector of discrete random variables. We define their **joint mixed**

**density** by

$$\begin{aligned}\rho(x,g) &= \rho(x|g)P(G=g) = \rho(x|g)p_G(g)\\ &= P(G=g|X=x)\rho_X(x) = p(g|x)\rho_X(x)\end{aligned}$$

This definition[4] helps us to express a joint mixed density in terms of products of (conditional)) densities in $X$ and (conditional) PMFs in $G$. Thereby, we end up with mathematical objects that we have discussed before.

To conclude the discussion of this issue, we add

**Example 2.6** (Mixed joint density) Let $X$ be a continuous random variable $X : \Omega \to \mathbb{R}^2$ that follows the uniform distribution on $[0,1]^2$. Moreover, let $G : \Omega \to \{1,2\}$ be a discrete random variable. We define $G$ via a function applied to $X$, i.e. with

$$G = f(X), \quad f(x) = f(x_1,x_2) = \begin{cases} 1 & \text{if } x_1 < 0.5 \\ 2 & \text{if } x_1 \geq 0.5 \end{cases}$$

So $G$ indicates whether the first component of $X$ is larger or smaller than 0.5.

We are now interested to calculate the joint mixed density $\rho(x,g)$ on $[0,1]^2 \times 1,2$. To do that, we simply apply the definition of the mixed joint density and get

$$\rho(x,g) = p(g|x)\rho_X(x) = P(G=g|X=x)\rho_X(x)\,.$$

As $X$ is uniformly distributed, we know that on $[0,1]^2$ it holds $\rho_X(x) = \left(\frac{1}{1-0}\right)^2 = 1$. And we had $G = f(X)$. Hence we continue with

$$\rho(x,g) = P(G=g|X=x) = P(f(X)=g|X=x) = \begin{cases} 1 & \text{if } f(x) = g \\ 0 & \text{if } f(x) \neq g \end{cases}\,.$$

The last equality is rather obvious, as the conditioning to $X = x$ makes the event $f(X) = g$ deterministic and only dependent on the choice of $x$ and $g$.

Let us finally check, whether $\rho(x,g)$ is properly normalized. In the discrete case, we would sum up the PMF over the range of the discrete RV. In the continuous case, we

[4] We *define* the joint mixed density here. If we would treat random variables with a more complex framework of distributions, $\sigma$-fields, measures, etc., we would not have to make that strict difference between discrete and continuous RVs and could state this *definition* as a *lemma* based on more general properties.

would "integrate out" both variables. In the mixed case, this becomes

$$\begin{aligned}
\int_{[0,1]^2} \sum_{g \in R_g} \rho(x,g) dx &= \int_{[0,1]^2} \sum_{g=1}^{2} \left\{ \begin{array}{ll} 1 & \text{if } f(x) = g \\ 0 & \text{if } f(x) \neq g \end{array} \right\} dx \\
&= \int_{[0,1]^2} \left\{ \begin{array}{ll} 1 & \text{if } f(x) = 1 \\ 0 & \text{if } f(x) \neq 1 \end{array} \right\} + \left\{ \begin{array}{ll} 1 & \text{if } f(x) = 2 \\ 0 & \text{if } f(x) \neq 2 \end{array} \right\} dx \\
&= \int_{[0,0.5]\times[0,1]} \left\{ \begin{array}{ll} 1 & \text{if } f(x) = 1 \\ 0 & \text{if } f(x) \neq 1 \end{array} \right\} + \left\{ \begin{array}{ll} 1 & \text{if } f(x) = 2 \\ 0 & \text{if } f(x) \neq 2 \end{array} \right\} dx \\
&+ \int_{[0.5,1]\times[0,1]} \left\{ \begin{array}{ll} 1 & \text{if } f(x) = 1 \\ 0 & \text{if } f(x) \neq 1 \end{array} \right\} + \left\{ \begin{array}{ll} 1 & \text{if } f(x) = 2 \\ 0 & \text{if } f(x) \neq 2 \end{array} \right\} dx \\
&= \int_{[0,0.5]\times[0,1]} (1+0)\, dx + \int_{[0.5,1]\times[0,1]} (0+1)\, dx \\
&= \frac{1}{2} + \frac{1}{2} = 1\,.
\end{aligned}$$

And indeed, the mixed joint density is properly normalized. △

**Remark** The reader will have noticed that working with mixed joint densities can be tedious and quite technical, if we look into the details. In some cases, we will have to do that. However, most of the more complex results discussed here, e.g. the bias-variance decomposition from Chapter **??**, will only be treated for the regression case, i.e. for standard joint densities. Looking into classical machine learning literature [2, 6], we also notice that the classification case is often neglected for deeper statements due to its technicallity. △

This concludes the discussion of a modeling of the input/output data.

## 2.4 Pre-processing

We briefly give two examples of potentially necessary pre-processing steps of data for machine learning. Certainly, there is way more to discuss on this matter. However, for that, we refer to the respective literature, e.g. [5].

**Example 2.7** (Cleaning) Under the notion of *cleaning*, we understand all sorts of operations on input data that makes sure that the data is in the format / shape that we expect it to have. If we e.g. work on data from surveys or measurements, there might be some missing values or values that do not fit into the expected range. Those wrong entries have to be removed or corrected. In case of some input signal (voice, etc.) we might also want to remove potentially existing noise. △

**Example 2.8** (Normalization) Often, we have input vectors, where some input variable has a large value range, e.g $[0, 10000]$, while some other input variable only has a

very small range $[0, 0.001]$. In other cases, the value ranges might have a large offset, e.g. $[10000, 11000]$. Some of the methods that we will study could be seriously affected by such deviations. Others might not be affected negatively, however, the e.g. the size of the input value might have an impact on how much that variable will be considered in the training. Therefore, we often have to shift and normalize the input data to a meaningful range. △

## 2.5 Representations

We are regularly confronted with input data, for which we do not have an obvious way to write it as a vector of real numbers. In those cases, we look for proper *representations* for these inputs. As mentioned before, the topic of finding good representations is a research by itself. Indeed, different representations can lead to strongly different results in terms of prediction error. Hence, finding a good representation for data will be an extremely important part of practical machine learning. Nevertheless, we will – in this work – mostly consider already existing good representations and will look in the theoretical and technical details of the machine learning training, evaluation, etc. process itself.

Still, we would like to pick two examples of very different input data, for which we give an exemplary idea of some proper representation.

**Example 2.9** (Text) One of the first very important applications for machine learning has been SPAM detection, i.e. classification. In that setting, we are given input strings, i.e. e-mails, and try to associate them with the label "SPAM" or "no SPAM". Very naively, we could describe each symbol in the string by its ASCII / Unicode code and would thus receive an input vector of values. However, this does not expose the structure of the data to the machine learning model.

Instead, deriving statistics and derived quantities of the input text is a much better way to represent text. In, e.g., [6] it is discussed a very simplistic approach representing potential SPAM text by a 57-dimensional input vector $X$ with variables

- $X_1, \ldots, X_{48}$: frequency of 48 key words,
- $X_{49}, \ldots, X_{54}$: frequency of 6 specific characters,
- $X_{55}$: average length of uninterrupted sequences of capital letters,
- $X_{56}$: longest length of uninterrupted sequences of capital letters, and
- $X_{57}$: total number of capital letters.

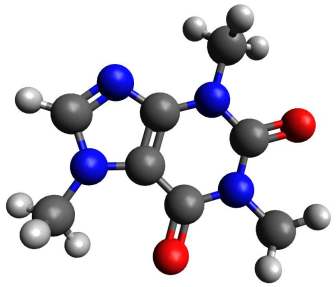
Intuitively, it might be clear, why this representation can help to characterize SPAM. However, if we want to go for more general input texts, we need less application-specific representations.

Nowadays Natural Language Processing frameworks automatically generate text representations for given text input data sets. These approaches combine various types of more sophisticated methodologies, see e.g. [7]. △

**Example 2.10** (Molecules) At the intersection of applications in chemistry, biology and physics, there is a strongly growing interest to use machine learning models to predict properties of *molecules*. This is of relevance for the design of new materials, for drug discovery or for modeling of complex biological systems. Let us, for the sake of simplicity, just assume that a molecule is given by a list of atoms with coordinates in three-dimensional space:

| | | | |
|---|---|---|---|
| N | 1.5808 | 0.7027 | -0.2279 |
| C | 1.7062 | -0.7374 | -0.2126 |
| N | 0.5340 | -1.5671 | -0.3503 |
| C | 0.3231 | 1.3600 | 0.0274 |
| C | -0.8123 | 0.4553 | 0.0817 |
| C | -0.6967 | -0.9322 | -0.0662 |
| N | -2.1886 | 0.6990 | 0.2783 |
| C | -2.8512 | -0.5205 | 0.2532 |
| N | -1.9537 | -1.5188 | 0.0426 |
| C | 0.6568 | -3.0274 | -0.1675 |
| O | 2.8136 | -1.2558 | -0.1693 |
| O | 0.2849 | 2.5744 | 0.1591 |
| C | -2.8096 | 2.0031 | 0.5032 |
| C | 2.8301 | 1.5004 | -0.1968 |
| H | -3.9271 | -0.6787 | 0.3762 |
| H | 1.4823 | -3.4046 | -0.7865 |
| H | -0.2708 | -3.5204 | -0.4868 |
| H | 0.8567 | -3.2990 | 0.8788 |
| H | -2.4123 | 2.7478 | -0.2017 |
| H | -2.6042 | 2.3621 | 1.5221 |
| H | -3.8973 | 1.9344 | 0.3695 |
| H | 3.5959 | 1.0333 | -0.8314 |
| H | 3.2249 | 1.5791 | 0.8255 |
| H | 2.6431 | 2.5130 | -0.5793 |

[5)]



This representation, even after converting the atom names to some numerical value, has turned out to be not helpful in order to describe a molecule as an input to a machine learning model.

A rather easy to explain and simple representation for molecules is proposed in [8] and represents molecules as *Coulomb matrices* $\mathcal{M}^{Coulomb} = (m_{ij})_{i,j=1}^{n}$ with

$$m_{ij} = \begin{cases} 0.5 Z_i^{2.4} & \text{if } i = j \\ \frac{Z_i Z_j}{R_{ij}} & \text{if } i \neq j \end{cases} .$$

The matrix is of size $n \times n$, where $n$ is the number of atoms in the given molecule. The property $Z_i$ is the *nuclear charge* (i.e. a known property) of the $i$th atom in the molecule. $R_{ij}$ is the distance between the atoms with the indices $i$ an $j$. To form a vector out of the matrix, the matrix it is simply linearized.

Why is this a better representation than pure coordinates? This representation is rotation- and translation-invariant, i.e. the actual position with respect to the coordinate system has no influence on the representation. In nowadays practice, much better representations are used, however they also follow that idea to be rotation- and translation-invariant. △

We have now seen the initial steps of the machine learning pipeline. Next, we enter the discussion of the (supervised) machine learning tasks.

[5)] The molecule shown here is consumed by many of us in the form of a hot beverage. :-)

# 3. Statistical Decision Theory

In this chapter, we would like to find proper mathematical defitions for the supervised machine learning tasks *regression* and *classification*. The *statistical decision theory* will provide the necessary notational tools and definitions. Specifially, it will also shed light on the questions of "What is a good prediction?", how to quantify "good" and "What is regression / classification?".

We start by the introduction of the core notions.

**Definition 3.1** (Regression) Let $X$ be an input and $Y$ be a quantitative output with joint density $\rho(X, Y)$. In **regression** we seek, given a training set $\{(x_i, y_i)\}_{i=1}^N$, for a function $f : \mathbb{R}^D \to \mathbb{R}^K$ that minimizes the error between $Y$ and its prediction $\hat{Y} = f(X)$.

**Definition 3.2** (Classification) Let $X$ be an input and $G$ be a qualalitative output with density $\rho(X, G)$. In **classification** we seek, given a training set $\{(x_i, g_i)\}_{i=1}^N$, for a function $f : \mathbb{R}^D \to R_G$ that minimizes the error between $G$ and its prediction $\hat{G} = f(X)$.

While these definitions for regression and classification might be more precise than previous attempts, some of us might still be dissatisfied, since we don't have defined the notion of a *prediction error*, yet. However, before we give that error definition, we first need a metric / distance concept for outputs.[1)]

**Definition 3.3** Let a quantitative output $Y \in \mathbb{R}^K$ or a qualitative output $G \in R_G$ be given. The **loss L** is a bivariate function on $\mathbb{R}^K$ or $R_G$ that gives a measure for the distance between elements in $\mathbb{R}^K$ / $R_G$.

**Example 3.1** (Losses) We give a few examples for losses. For $y, y' \in \mathbb{R}^K$, $g, g' \in \{1, \dots, r\}$, we know

[1)] Why do we only need a metric for outputs? We only need it on the output side, since our to be predicted data is the *output* data.

- *squared error / $\ell_2$ loss* ($K = 1$)

$$L_2(y, y') = (y - y')^2 \,,$$

- *squared error / $\ell_2$ loss* (for abitrary $K$)

$$L_2(y, y') = (y - y')^\top (y - y') \,,$$

- *$\ell_1$ loss* ($K = 1$)

$$L_1(y, y') = |y - y'| \,,$$

- *$0 - 1$ loss*

$$L_{0-1}(g, g') = \begin{cases} 0 & \text{if } g = g' \\ 1 & \text{if } g \neq g' \end{cases} \,.$$

The $0 - 1$ loss can be used for qulitative outputs, while the remaining losses can be used for quantitative outputs. △

## 3.1 Regression

Let us focus for now on the *regression* task and introduce a common measure for the error in this task.

**Definition 3.4** (Expected (squared) predition error in regression) Let $X$, $Y$ be input and quantitative output with joint density $\rho(x, y)$ and $f : \mathbb{R}^D \to \mathbb{R}^K$ a function to predict $Y$ from $X$. For the squared loss, the **expected (squared) prediction error** or **risk of** $f$, $\mathrm{EPE}(f)$, is defined as

$$\begin{aligned} \mathrm{EPE}(f) &= \mathrm{E}\left(L_2(Y, f(X))\right) \\ &= \int_{R^K} \int_{R^D} L_2(y, f(x)) \rho(x, y)\, dx dy \,. \end{aligned}$$

**Remark** (*Risk* terminology) Depending on the community, book, etc. the expected prediction error is also called *risk*, as stated in the definition. Often, this comes together with the use of the notation $R(f)$ instead of $EPE(f)$. Then, the objective is also not an *error* minimization, but a *risk* minimization. However, this is all primarily terminology and we stick here to the more "data-oriented" *error* wording. △

So what does the above definition mean? We introduce a *statistical* measure for the prediction error. Specifically, we look at the mean of the random variable $L_2(Y, f(X))$ that models the distance between our "true" output $Y$ and our model for the output that we get by applying the to be found function $f$ to our input. In some sense, we

imply that we will have found a good predictor $f(X)$ for $Y$, if they are in average close to each other.

The following example is a purely artificial one, as we will never know the dependency between input and output variables. Nevertheless, it will help us to better understand the notion of the expected prediction error. However, before we come to that example, we need a little bit more high-level probability theory, which we will try to accept as pure knowledge.[2)]

**Knowledge 3.1** Let $X, Y$ be continuous random variables and $Y$ is given as

$$Y = g(X)\,,$$

where $g$ is a function on the range of $X$ that needs to fulfill some properties. We can give the joint probability density for $X$ and $Y$ by

$$\rho(x, y) = \rho_X(x)\delta(y - g(x))\,.$$

Here, $\rho_X$ is the usual marginal density of $X$, while $\delta$ is the **Dirac delta function**[a]. We define the Dirac delta function only via its occurence as an integrand in a product with some function $f$. In this context it leads to

$$\int_{\mathbb{R}} f(x)\delta(x - x_0)dx = f(x_0)\,.$$

Others sometimes also define the Dirac delta function via

$$\delta(x - x_0) = \begin{cases} \infty & \text{if } x = x_0 \\ 0 & \text{else} \end{cases}$$

and

$$\int_{\mathbb{R}} \delta(x - x_0)dx = 1\,.$$

However, we stick to the the first view. Note that a deeper understanding of the above statements requires advanced knowlede on *measure theory* and *probability theory*. Specifically, we even no longer work with respect to the *Lebesgue measure*, which is quite common in probability theory.

[a] Actually, the *Dirac delta function* is not a function. However, it is still often called like that.

**Example 3.2** (Expected (squared) prediction error) Let $X : \Omega \to \mathbb{R}$ be an input variable and $Y : \Omega \to \mathbb{R}$ be an output variable. For the input variable we assume that it follows

[2)] Knowledge blocks will be used here and in the following to give the reader some mathematical knowledge that is necessary to understand some upcoming statement, while accepting that a more detailed explanation for the knowledge might require some much deeper mathematical tools, which will not be discussed here.

the uniform distribution as $X \sim \mathcal{U}[0,1]$. Moreover, we make the very strong assumption to know the "true" dependency between $X$ and $Y$. Specifically we define $Y$ via

$$Y := g(X), \quad \text{with} \quad g(x) = x^2 \,.$$

Now we look for a function $f$ that shall approximate that (usually unknown) relationship between $X$ and $Y$. We could come up with

$$f(x) = x \,,$$

thus claim that $f(X)$ will be a good approximation to $Y = g(X)$. Let us calculate the expected (squared) prediction error for $f$:

$$EPE(f) = \mathrm{E}\left(L_2(Y, f(X))\right) = \int_{\mathbb{R}} \int_{\mathbb{R}} (y - f(x))^2 \rho(x,y) dx dy$$

We briefly have to interrupt the calculation and need to find $\rho(x,y)$, which is however immediately given by the above knowledge block.

$$\rho(x,y) = \rho_X(x)\delta(y - x^2) \,.$$

We continue the evaluation of the expected prediction error by

$$\begin{aligned}
EPE(f) &= \int_{\mathbb{R}} \int_{\mathbb{R}} (y - f(x))^2 \rho(x,y) dx dy = \int_{\mathbb{R}} \int_{\mathbb{R}} (y - x)^2 \rho_X(x) \delta(y - x^2) dx dy \\
&= \int_{\mathbb{R}} \int_{\mathbb{R}} (y - x)^2 \rho_X(x) \delta(y - x^2) dy dx = \int_{\mathbb{R}} \rho_X(x)(x^2 - x)^2 dx \\
&= \int_0^1 (x^2 - x)^2 dx = \int_0^1 x^4 - 2x^3 + x^2 \, dx = \left[\frac{1}{5}x^5 - \frac{1}{2}x^4 + \frac{1}{3}x^3\right]_0^1 \\
&= \frac{1}{30} \approx 0.033
\end{aligned}$$

The first two equalities are pure use of the respective definitions. In the third equality, we change the integration order. The fourth equality applies the knowledge gathered on the Dirac delta function. In the fifth equality, we use that $X$ is uniformly distributed on $[0,1]$. The rest are basic calculations. △

The following fundamental theorem will give us the answer to the question "What *is* regression?" (assuming that we use the $L_2$ loss):

**Theorem 3.1** Let $X$, $Y$ be input and quantitative output with joint density $\rho(x,y)$. The function $f : \mathbb{R}^D \to \mathbb{R}^K$ that minimizes (under appropriate conditions) the expected (squared) prediction error $\mathrm{EPE}(f)$ is given by

$$f(x) = \underset{g(x)}{\arg\min}\, \mathrm{EPE}(g) = \mathrm{E}\left(Y | X = x\right)$$

$E(Y|X = x)$ is often called **regressor** / **regression function**.

Let us recall what we do in regression: We have quantitative inputs and outputs, have an (unknown) joint distribution between them, and seek for a function $f$ giving the variable $f(X)$ that shall approximate $Y$. Moreover, we do this, such that the expected (squared) prediction error becomes minimal. Under these circumstances, the above theorem tells us that the best possible predictor, the *regressor*, is the conditional mean of $Y$ given $X = x$.

However, we still don't talk about data, yet. Originally, we stated that we want to find the predictor from tranining data, but now we only got this very abstract formula. Indeed, the *regressor* will provide us a recipe to develop regression algorithms. Unfortunately, we still need a bit of patience to get there.

Before that, we still give an example for the evaluation of the regressor and need to give the proof for the theorem.

**Example 3.3** We continue our previous example. Recall that we had the uniformly distributed input $X$ and the output $Y$, for which we (atypically) knew its dependency on $X$ by $Y = g(X) = X^2$. According to Knowledge 3.1, this dependency is encoded in the conditional density

$$\rho(y|x) = \frac{\rho(y,x)}{\rho_X(x)} = \frac{\delta(y-x^2)\rho_X(x)}{\rho_X(x)} = \delta(y-x^2)\,.$$

Given this (perfect) knowledge, we evaluate the regressor to figure out, what would be the best predictor under the given circumstances:

$$\mathrm{E}\,(Y|X=x) = \int_{\mathbb{R}} y\rho(y|x)\,dy = \int_{\mathbb{R}} y\delta(y-x^2)dy = x^2$$

It seems not to be too surprising that indeed $x^2$, which is the exact dependency between $X$ and $Y$, is the best possible predictor for $Y$. This gives us at least the intuition that the regressor does its job. △

Next, we come to the proof of our regressor theorem. We enter the discussion of proofs by a small remark.

**Remark** (Proofs) Literature on machine learning treats mathematical statements differently. Some literature gives no argument or proof for the correctness of given mathematical statements. Others give a derivation, however jump over many steps which makes it hard to really understand the derivations.

In this work, we will give proofs for some important mathematical statements. The proofs will have will have a high degree of detail, i.e. we don't skip details. Thereby, they should allow the majority of the readers to indeed follow the individual steps.

Why can't we skipt the proofs? We discuss proofs / derivations since they allow us to gain further understanding for the objects at hand. In that sense, proofs should not be considered as abstract mathematical concepts but as examples of the use of all the objects that we learn here. △

We finally come to the proof of Theorem 3.1, which is assembled from [2, 6].

Proof

We restrict ourselves to $K = 1$, i.e. one-dimensional outputs. We first start from the expected (squared) prediction error $\mathrm{EPE}(g)$ for some function $g : \mathbb{R}^D \to \mathbb{R}$ and apply a couple of transformations to it. The core idea of these transformations is "conditioning on $X$", thus we express the mean as a conditional mean with respect to $X$. To do this, we apply the equality

$$\rho(x, y) = \rho(y, x) = \rho(y|x)\rho_X(x)$$

from the definition of the conditional density. Using this, we enter into the transformations for $\mathrm{EPE}(g)$:

$$\begin{aligned}
\mathrm{EPE}(g) &= \mathrm{E}\left(L_2(Y, g(X))\right) = \mathrm{E}\left((Y - g(X))^2\right) = \int_{\mathbb{R}} \int_{\mathbb{R}^D} (y - g(x))^2 \rho(x, y)\, dxdy \\
&= \int_{\mathbb{R}} \int_{\mathbb{R}^D} (y - g(x))^2 \rho(y|x)\rho_X(x)\, dxdy = \int_{\mathbb{R}^D} \int_{\mathbb{R}} (y - g(x))^2 \rho(y|x)dy\, \rho_X(x)dx \\
&= \int_{\mathbb{R}^D} \mathrm{E}_{Y|X}\left((Y - g(X))^2|X = x\right) \rho_X(x)dx = \mathrm{E}_X\left(\mathrm{E}_{Y|X}\left((Y - g(X))^2|X\right)\right) .
\end{aligned}$$

The first row of equalities just uses the corresponding definitions. The first equality in the second line uses the conditioning discussed above. Afterwards, we flip the two integrals, while assuming that the integrands below fullfill the necessary conditions. Then we use the definition of the conditional expectation to see that

$$\mathrm{E}_{Y|X}\left((Y - g(X))^2|X = x\right) = \int_{\mathbb{R}} (y - g(x))^2 \rho(y|x)dy\,.$$

Note that we could also have left away the subscript $Y|X$ at the (conditional) expectation operator. We only use it, to distinguish this conditional expectation from the (marginal) expectation with respect to $X$, afterwards. The last equality is again only the application of the definition of the expectation operator.

We continue to apply transformations to the result of the previous calculations:

$$\begin{aligned}
&\mathrm{E}_X\left(\mathrm{E}_{Y|X}\left((Y - g(X))^2|X\right)\right) = \mathrm{E}_X\left[\mathrm{E}_{Y|X}\left((Y + \mathrm{E}\left(Y|X\right) - \mathrm{E}\left(Y|X\right) - g(X))^2|X\right)\right] \\
= \;& \mathrm{E}_X\left[\mathrm{E}_{Y|X}\left((g(X) - \mathrm{E}\left(Y|X\right))^2 + 2(g(X) - \mathrm{E}\left(Y|X\right))(\mathrm{E}\left(Y|X\right) - Y) + (\mathrm{E}\left(Y|X\right) - Y)^2|X\right)\right] \\
= \;& \mathrm{E}_X\left[\mathrm{E}_{Y|X}\left((g(X) - \mathrm{E}\left(Y|X\right))^2|X\right)\right] - \mathrm{E}_X\left[\mathrm{E}_{Y|X}\left((\mathrm{E}\left(Y|X\right) - Y)^2|X\right)\right]
\end{aligned}$$

Here, the first equality adds a zero (i.e. $+\mathrm{E}\left(Y|X\right) - \mathrm{E}\left(Y|X\right)$). The next step is an elementary calculation. In the last equality, we use the linearity of the expectation and

$$\mathrm{E}_X\left[E_{Y|X}\left[2(g(X) - \mathrm{E}\left(Y|X\right))(\mathrm{E}\left(Y|X\right) - Y)\middle|X\right]\right] = 0\,.$$

We leave this last result as an exercise to the reader.[3)] Finally, we observe that the last term, $\mathrm{E}_X\left(\mathrm{E}_{Y|X}\left((\mathrm{E}\left(Y|X\right)-Y)^2|X\right)\right)$, is independent of $g$. We will need that immediately.

Until now, we applied conditioning and mostly elementary transformations to the expected prediction error. However, our main goal is to find its minimizer

$$f = \arg\min_g \mathrm{EPE}(g),$$

thus we carry out this minimization. Our previous calculations immediately give us

$$f = \arg\min_g \mathrm{E}_X\left[\mathrm{E}_{Y|X}\left((g(X)-\mathrm{E}\left(Y|X\right))^2|X\right)\right] - \mathrm{E}_X\left[\mathrm{E}_{Y|X}\left((\mathrm{E}\left(Y|X\right)-Y)^2|X\right)\right].$$

However, as the second term does not depend on $g$, we can simpliy leave it out in the minimization and get

$$\begin{aligned} f &= \arg\min_g \mathrm{E}_X\left[\mathrm{E}_{Y|X}\left((g(X)-\mathrm{E}\left(Y|X\right))^2|X\right)\right] \\ &= \arg\min_g \int_{\mathbb{R}^D}\int_{\mathbb{R}} (g(x)-\mathrm{E}\left(Y|X=x\right))^2 \rho(y|x)dy\rho_X(x)dx \\ &= \mathrm{E}\left(Y|X\right) \end{aligned}$$

How do we obtain the last equality? We have a look at the integrand $(g(x)-\mathrm{E}\left(Y|X=x\right))^2$, which is non-negative. Obviously, it will be minimized to zero by setting $g(x) = \mathrm{E}\left(Y|X=x\right)$. Then the multiple integral becomes zero and we obtain our (global) minimizer. □

As mentioned before, the regressor will be our starting point to derive regression methods for concrete training data. We will now give an example for such a derivation for the very simple *k-nearest neighborhood* (kNN) regression.

**Method 1** (k nearest neighborhood (kNN) regression) We can introduce *kNN regression* as an approximation to the regressor $\mathrm{E}\left(Y|X=x\right)$. More specifically, we approximate the mean by an averaging and relax the conditioning to a single point to a conditioning to a neighboring region. Thereby, we obtain the (kNN regression) predictor

$$\hat{f} = \mathrm{Ave}(y_i|x_i \in N_k(x)).$$

Here, $\{x_i, y_i\}_{i=1}^N$ is a given training set, $N_k(x)$ are the $k$ nearest neighbors of $x$ in $\{x_i\}_{i=1}^N$ and Ave applied to a set of values gives its average. △

How do we interpret this predictor formula? If we want to evaluate the predictor at a location $x$, we look for those training samples $x_i$ that are the $k$ nearest neighbors in the observations and find the associated outputs $y_i$. Then, we compute the average over these $y_i$, which gives us the prediction. This leads to

[3)] Hence we will show this as a homework assignment.

**Algorithm 1** (kNN regression)

**input:** $\{(x_i, y_i)\}_{i=1}^N$ training data, $x$ evaluation point, $k$ neighborhood size

**output:** kNN prediction $\hat{Y}(x)$

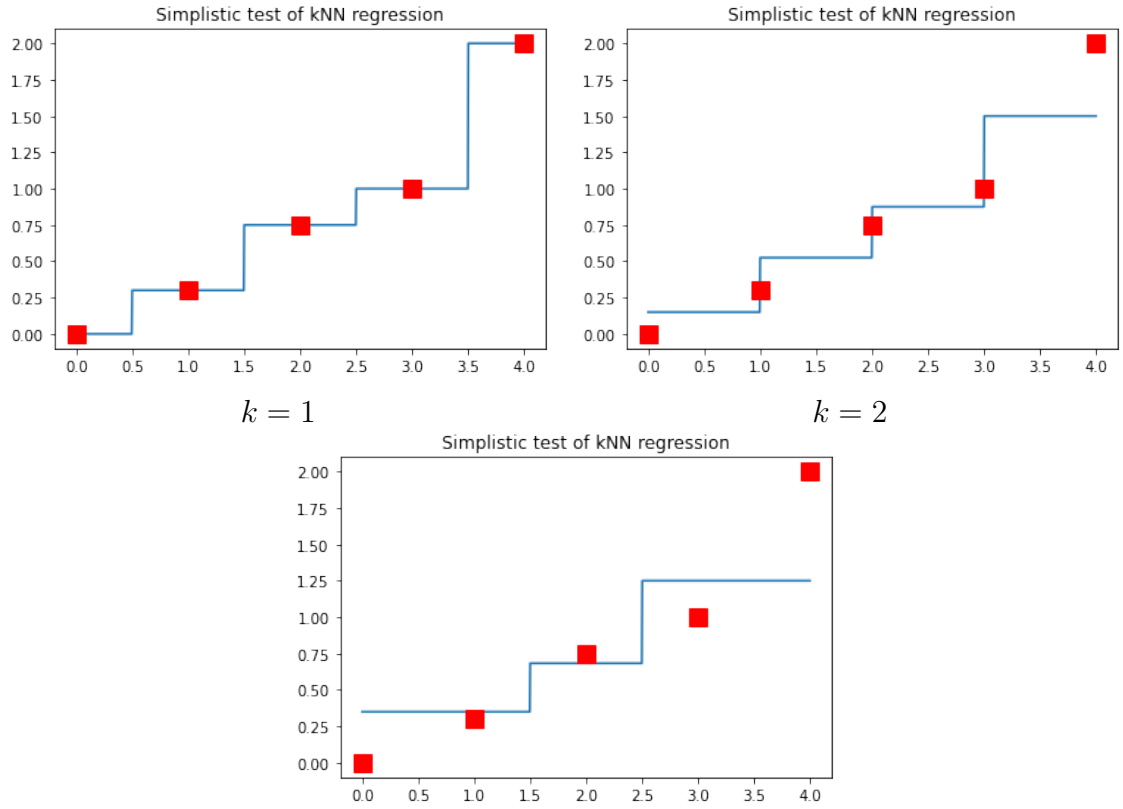
1. $\{i_1, \dots, i_k\} \leftarrow$ indices of $k$ nearest neighbors of $x$ in $\{x_i\}_{i=1}^N$
2. $\hat{Y}(x) \leftarrow \frac{1}{k} \sum_{j=1}^k y_{i_j}$
3. return $\hat{Y}(x)$

If we use a brute force kNN search, the above algorithm has a complexity of $O(D \cdot N)$. Using a kD-tree we obtain in the average case a complexity of $O(D \log N)$. We finally come up with an example.

**Example 3.4** (kNN regression) We consider one-dimensional input and output with the very simple training data

$$\{(0,0), (1,0.3), (2,0.75), (3,1), (4,2)\}.$$

Below, we give the predictor (blue) for the given training data (red) for $k = 1, 2, 3$.




$k = 1$ $k = 2$

$k = 3$

For smaller neighborhood, the predictor gets locally more accurate, i.e. it better fits the training data. However, globally, it has more fluctuations. On the other hand, if we go for a larger neighborhood size, we less accurately represent each sample, but get a "smoother" predictor. We will later study this as the *bias–variance trade-off*.

This regression task including the shown figures can be reproduced here: launch binder. △

We will discuss more properties of $kNN$ regression in Chapter 7.

## 3.2 Classification

In the previous section, we have discussed regression starting from an appropriate prediction error definition, going via a fundamental result on the structure of the regression task up to an exemplary first regression method.

This section will cover these same ideas for classification. Nevertheless, as we should have gained the big picture from the previous section, we will cut the discussion of classification a bit shorter.

We again start by introducing a meaningful prediction error concept.

**Definition 3.5** (Expected (0-1) predition error in classification) Let $X$, $G$ be input and a one-dimensional output with joint mixed density $\rho(x, g)$ and $f : \mathbb{R}^D \to R_G$ a function to predict $G$ from $X$. For the 0-1 loss, the **expected prediction error of** $f$, $\mathrm{EPE}(f)$, is defined as

$$\begin{aligned} \mathrm{EPE}(f) = \mathrm{E}\left(L_{0-1}(G, f(X))\right) &= \sum_{g \in R_G} \int_{\mathbb{R}^D} L_{0-1}(g, f(x)) \rho(x|g) dx \, p_G(g) \\ &= \int_{\mathbb{R}^D} \sum_{g \in R_G} \left(L_{0-1}(g, f(x)) p(g|x)\right) \, \rho_X(x) dx \, . \end{aligned}$$

As we can see, we use the $0-1$ loss from the initial loss example. The two terms on the right-hand side are the two potential flavors to express the expectation in this mixed joint density setting, see Definition 2.4.

The following fundamental theorem is the classification counterpart to the *regressor*.

**Theorem 3.2** Let $X$, $G \in \{1, \ldots, r\} =: R_G$ be input and qualitative output with joint mixed density $\rho(x, g)$. The function $f : \mathbb{R}^D \to R_G$ that minimizes (under appropriate conditions) the expected prediction error $\mathrm{EPE}(f)$ with respect to the $0-1$ loss is given

by

$$f(x) = \arg\min_{g \in R_G} [1 - p(g|x)] \,,$$

which can be simplified to

$$f(x) = \mathfrak{g} \quad \text{if} \quad p(\mathfrak{g}|x) = \max_{g \in R_G} p(g|x)$$

This minimizer is the **Bayes classifier**.

(We skip the proof, here.) Assuming that we consider the expected prediction error with the $0-1$ loss as the *right* way to measure error in classification, the *Bayes classifier* is the best possible predictor for classification tasks.

How to interpret the statement of the theorem? It gives us a rather clear recipe: For real data and a specific input $x$, we need to approximate the conditional probability / PMF of $G$ given $X = x$. Hence, we will get for each potential label $g \in R_G$ its (conditional) probability $p(g|x) = P(G = g|X = x)$. Then, we choose the label with the highest (conditional) probability as predictor.

We will use this recipe rather intensively for the construction of classifiers. In analogy to kNN regression, we here start with the kNN classification.

**Method 2** (k nearest neighborhood (kNN) classification) In this method, we start from the Bayes classifier

$$f(x) = \mathfrak{g} \quad \text{if} \quad p(\mathfrak{g}|x) = \max_{g \in R_G} p(g|x) \,.$$

Instead of using the conditional PMF $p(g|x)$ we consider training sample proportions and replace the point-wise conditioning by a conditioning over the $k$ nearest neighbors $N_k(x)$. Practically, this means that we find for an input $x$ the $k$ nearest neighbors in the input samples $\{x_i\}_{i=1}^N$. For each of these input samples, we look at the outputs. The output label that occurs most often in this neighborhood subset is then chosen as predictor. △

This method immediately results in the following algorithm.

**Algorithm 2** (kNN classification)

**input:** $\{(x_i, g_i)\}_{i=1}^N$ training data ($g_i \in \{1, \ldots, r\}$), $x$ evaluation point, $k$ neighborhood size

**output:** kNN prediction $\hat{G}(x)$

1. find $N_k(x)$ as the $k$ nearest neighbors of $x$ in $\{x_i\}_{i=1}^N$
2. $n_l \leftarrow \frac{1}{k} |\{x_i \in N_k(x) | g_i = l\}|$ for $l = 1, \ldots, r$
   *(relative number of samples with label $l$ in neigborhood)*

3. $\hat{G}(x) \leftarrow \arg\max_{l=1,\ldots,r} n_l$
4. return $\hat{G}(x)$

For simplicity the algorithm assumes that $r$ different labels with the values $\{1, \ldots, r\}$ are associated to the input measurements. Different e.g. string labels could be mapped to these values.

The running time of the kNN regression algorithm is, as for kNN regression, dominated by the kNN search.

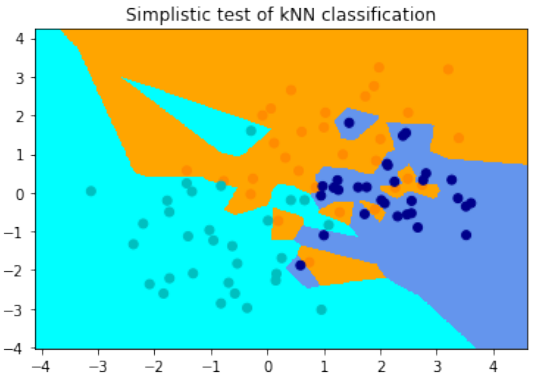
We conclude this section by an example for kNN classification.

**Example 3.5** (kNN classification) We consider $X, G$ with $X : \Omega \to \mathbb{R}^2$ and $G : \Omega \to \{1, 2, 3\}$. Our training data is artificially generated as follows. We sample from each of the normal distributions

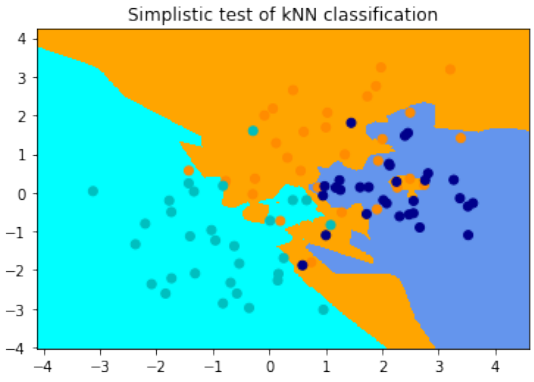
$$\mathcal{N}\left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\right), \quad \mathcal{N}\left(\begin{pmatrix} -1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\right), \quad \mathcal{N}\left(\begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\right)$$

thirty input samples. The input samples from the first distribution get the label 1 assigned, the input samples from the second distribution get the label 2 assigned and the input samples from the third distribution get the label 3 assigned.

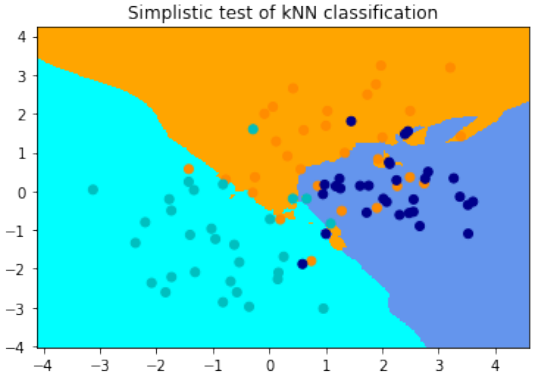
Then, we apply kNN classification to the training data and mark the areas in which the predictor will assign to the input the same label in one color. The result for $k = 1, 3, 10$ is shown below.




$k = 1$




$k = 3$




$k = 10$

Similar to the regression example, we observe that small neighborhood sizes lead to rather scattered prediction regions, while larger neighborhood sizes give a "smoother" predictor.

This classification task including the shown figures can be reproduced here: launch binder.

△

# 4. Linear regression

In this chapter, we will introduce the well-known linear regression by least squares. Most readers will have seen already linear regression for one-dimensional inputs and outputs. Even though the linear model, which is the basis for linear regression, is rather limited, it is still very often used in practical applications. Moreover, the simplicity of the model makes it a good test case to study some more theoretical properties of regression models in later chapters.

We start this chapter by a brief discussion of the relationship of the regressor and the linear model. Then, we formally introduce the linear model for $D$-dimensional inputs and one-dimensional outputs. This is followed by a discussion of the *training* of this model by *least squares parameter estimation*. Then we study the impact on noise in the training data on the constructed predictor. We conclude the chapter by the generalization of linear regression to $K$-dimensional outputs.

## 4.1 Linear model

In the previous chapter, we sharpened the notion of regression to a point, where we identified the regressor

$$\mathrm{E}\left(Y|X=x\right)$$

as the "best possible" choice for a predictor in the regression setup.[1)] Nevertheless, we also observed that the regressor only gives a "recipe" to construct actual data-based predictors or models. Let us briefly study two major classes of regression models that we will deal with.

**Remark** (Non-parametric vs. parametric models) Our first (data-based) regression model has been been the *kNN regression*. It could be immediately derived from the regressor. In the literature, this model is called a *non-parametric model*. A non-parametric model, roughly speaking, is a model that is fully characterized by the training data.

Since this characterization might not help to fully understand the "non-parametric" nature of a model, we should first start to introduce *parametric models*. A parametric model for regression describes the relationship between the input $X$ and the output $Y$ by

[1)] This was done under the assumption that the expected squared prediction error is the "right" error indicator.

a model, often a mathematical formula / function, with a finite number of parameters. To give an example, the function

$$f(x) = ax + b$$

is a (deterministic) parametric model for one-dimensional inputs and outputs with the two parameters $a, b$. Indeed, this is the *linear model*, for the one-dimensional case. In context of the regressor $\mathrm{E}(Y|X = x)$, a parametric model approximates the regressor, while making rather strong assumptions on the properties of the model. Graphically speaking, our one-dimensional linear model, just imposes the assumption that the predictor is a straight line through the data.

Moreover, after a training, which can be translated to the search for the best possible parameters, the evaluation of a parametric model no longer relies on the training data. Hence, the only thing that matters are the trained coefficients. This might make the model evaluation much cheaper, specifically, if we consider a growing training data size. On the other hand, non-parametric models rely on the training data in the evaluation, as seen for kNN regression, before.

So should we go for parametric or non-parametric models? There is no easy answer to this. Parametric models might be cheaper to evaluate but less flexible than non-parametric models. At the same time, the explicit choice of a parametric model might also help to impose structure on the predictor that might be necessary for some application. We will not go into further details here, but just observe that the majority of the models that we are going to discuss will be parametric models, like the linear model. △

In linear regression, we replace the regressor by the linear model:

**Definition 4.1** (Linear model) Let $\boldsymbol{X}$ be a $D$-dimensional input vector and $Y$ be a 1-dimensional quantitative output. The **linear model** is an approximation for the regressor $\mathrm{E}(Y|\boldsymbol{X} = \boldsymbol{x})$ of the form

$$f_\beta(\boldsymbol{X}) = \beta_0 + \sum_{j=1}^{D} X_j \beta_j \,,$$

where $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_D)^\top \in \mathbb{R}^{D+1}$ is a vector of unknown parameters or coefficients. $\beta_0$ is the **intercept** or **bias**. By considering the vector $\boldsymbol{Z} := (1, X_1, \ldots, X_D)^\top$ we can write the linear model as

$$f_\beta(\boldsymbol{Z}) = \boldsymbol{Z}^\top \beta \,.$$

We know some properties for this model, which are summarized in

**Lemma 4.1** The linear model $f_{\boldsymbol{\beta}}(\boldsymbol{X})$ is linear in its parameters $\boldsymbol{\beta}$ and affine in the input $\boldsymbol{X}$. For $\beta_0 = 0$ it is even linear in the input.

The proof is obvious.

**Remark** (Derivation of further models) In Chapter **??**, we will see that we can generalize the linear model to more complex non-linear models. The idea will be to replace the inputs $(1, X_1, \ldots, X_D)$ to the linear model by derived quantities of these inputs. We can e.g. build a quadratic model by using the inputs

$$(1, X_1, X_1^2, X_2, X_2^2, X_1 X_2)\,.$$

In that context, understanding the properties of the linear model will help us to understand more advanced (regression) models. △

## 4.2 Least squares parameter estimation

We now come to the *training* of the linear model. Hence, we want to find the coefficient vector $\hat{\boldsymbol{\beta}}$ such that the linear model "fits best" the given training data. We also call this process *parameter estimation*.

**Definition 4.2** Let $\boldsymbol{X}$ be an input vector and $Y$ be an output and let $f_{\boldsymbol{\gamma}} : \mathbb{R}^D \to \mathbb{R}$ be a parametric model / predictor that is supposed to describe the hidden relationship between the input and output and that is parametrized by a parameter vector $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_Q)^\top$. We are further given a training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ for $\boldsymbol{X}$,$Y$. A method to choose parameters $\boldsymbol{\gamma}$ is called **parameter estimator**.

There can be many different flavors of doing parameter estimation. In this section, we will introduce and use *least squares parameter estimation* that brings the idea of the expected squared prediction error to the discrete level:

**Definition 4.3** Let $\boldsymbol{X}$, $Y$, $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ and $f_{\boldsymbol{\gamma}}$ be given a before. The parameter estimator of form

$$\hat{\boldsymbol{\gamma}} = \arg\min_{\boldsymbol{\gamma}} \sum_{i=1}^{N} (y_i - f_{\boldsymbol{\gamma}}(\boldsymbol{x}_i))^2 = \arg\min_{\boldsymbol{\gamma}} \sum_{i=1}^{N} L_2(y_i, f_{\boldsymbol{\gamma}}(\boldsymbol{x}_i))$$

is called **least squares estimator**.

Indeed, the term that we minimize on the right-hand side is, besides of the normalizing constant $\frac{1}{N}$ (which anyway has no influence on the minimization), an empirical estimator for the mean over the $L_2$ loss between the training data and the predicted output. Hence, the least squares estimator finds the set of parameters $\hat{\boldsymbol{\gamma}}$ that minimizes the (empirical) mean over the $L_2$ loss between training data and predictor.

Note here, that the least squares estimator is not limited to the linear model. Indeed our definition is given for any parametric model $f_{\boldsymbol{\gamma}}$.

Next we combine the linear model with the least squares estimator, which gives us *linear regression by least squares* via

**Theorem 4.1** (Linear regression by least squares) Let $\boldsymbol{X}$, $Y$, $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ and the linear model $f_{\boldsymbol{\beta}}$ be given as before. We consider the matrix $\mathcal{X} \in \mathbb{R}^{N\times(D+1)}$

$$\mathcal{X} = \left(\begin{array}{c|c} \mathbf{1} & \begin{array}{c} \boldsymbol{x}_1^\top \\ \boldsymbol{x}_2^\top \\ \vdots \\ \boldsymbol{x}_N^\top \end{array} \end{array}\right)$$

containing the input measurements $\boldsymbol{x}_i$ as rows, where we prepend a 1 in each row. Let us further assume that $\mathcal{X}$ has full column rank. Then the least squares estimator $\hat{\boldsymbol{\beta}}$ for the parameters $\boldsymbol{\beta}$ in the linear model evaluates to

$$\hat{\boldsymbol{\beta}} = \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{y}\,,$$

where $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$.

We conclude that the *training* process in linear regression by least squares is the evaluation of the least squares estimator for the linear model $f_{\boldsymbol{\beta}}$. According to the theorem, this is achieved by evaluating $\hat{\boldsymbol{\beta}} = \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{y}$, which gives the coefficients $\hat{\boldsymbol{\beta}}$ and thus the predictor $f_{\hat{\boldsymbol{\beta}}}(\boldsymbol{X})$.

We skip the proof for now and first concentrate on an example:

**Example 4.1** Let $\boldsymbol{X} : \Omega \to \mathbb{R}^2$, $Y : \Omega \to \mathbb{R}$ be the input and output with associated training data

$$\mathcal{T} = \left\{\left((1,0)^\top, 1\right), \left((1,1)^\top, 2\right), \left((2,1)^\top, 4\right), \left((4,2)^\top, 5\right)\right\}\,.$$

We want to build a predictor $f$ using linear regression by least squares. We follow the steps outlined in Theorem 4.1 and define the matrix $\mathcal{X}$ and the vector $\boldsymbol{y}$ by

$$\mathcal{X} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 4 & 2 \end{pmatrix}, \quad \begin{pmatrix} 1 \\ 2 \\ 4 \\ 5 \end{pmatrix}$$

We further evaluate

$$\mathcal{X}^\top \mathcal{X} = \begin{pmatrix} 4 & 8 & 4 \\ 8 & 22 & 11 \\ 4 & 11 & 6 \end{pmatrix}, \quad \mathcal{X}^\top \boldsymbol{y} = \begin{pmatrix} 12 \\ 31 \\ 16 \end{pmatrix}.$$

Finally, we solve the linear system of equations[2)]

$$\mathcal{X}^\top \mathcal{X} \hat{\boldsymbol{\beta}} = \mathcal{X}^\top \boldsymbol{y}$$

and obtain

$$\hat{\beta} = \begin{pmatrix} \frac{2}{3} \\ \frac{2}{3} \\ 1 \end{pmatrix}.$$

The resulting predictor $f_{\hat{\boldsymbol{\beta}}}$ is given by

$$f_{\hat{\boldsymbol{\beta}}}(\boldsymbol{X}) = \frac{2}{3} + \frac{2}{3} X_1 + X_2 \,.$$

△

Next, we would like to prove Theorem 4.1. Depending on the mathematical background of the reader, knowledge on calculus in higher dimensions might be not (perfectly) present. Therefore, we briefly recall or introduce some basic knowledge in the field.

**Knowledge 4.1** Let a function $f : \mathbb{R}^D \to \mathbb{R}$ be given. Assuming that a certain number of conditions are fulfilled for $f$, we can introduce the **gradient** of $f$ as

$$\nabla f(\boldsymbol{x}) = \operatorname{grad} f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\boldsymbol{x}) \\ \vdots \\ \frac{\partial f}{\partial x_d}(\boldsymbol{x}) \end{pmatrix}.$$

It is simply the vector of first partial derivatives. Moreover, we can introduce the Hessian

$$\mathcal{H}_f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_D}(\boldsymbol{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_D \partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial^2 f}{\partial x_d \partial x_D}(\boldsymbol{x}) \end{pmatrix}$$

as the matrix of second partial derivatives. It is then well-known that we can find local extrema of $f$ by

1. computing those $\boldsymbol{x}$ such that
$$\nabla f(\boldsymbol{x}) = \operatorname{grad} f(\boldsymbol{x}) = \boldsymbol{0}\,,$$
2. computing the Hessian $\mathcal{H}_f(\boldsymbol{x})$ at these $\boldsymbol{x}$.

[2)] From numerical mathematics, we know that we should never explicitly invert a matrix and apply it to some vector but should always solve the corresponding linear system of equations.

If for an $\boldsymbol{x}$ the gradient is the zero vector and the Hessian is a positive / negative definite matrix, then $\boldsymbol{x}$ is the position of a **local minimum / maximum**.

We take this knowledge as given and finally enter the proof of Theorem 4.1.

Proof
Let $(\boldsymbol{x}_i, y_i) \in \mathbb{R}^D \times \mathbb{R}$ be the training samples with $i = 1, \dots, N$. We introduce the notation $\boldsymbol{z}_i = \begin{pmatrix} 1 \\ \boldsymbol{x}_i \end{pmatrix} \in \mathbb{R}^{D+1}$. Then we consider the linear model in the notation $f_{\boldsymbol{\beta}}(\boldsymbol{Z}) = \boldsymbol{Z}^\top \boldsymbol{\beta}$. Moreover, $\mathcal{X}$ is given by

$$\mathcal{X} = \begin{pmatrix} \boldsymbol{z}_1^\top \\ \vdots \\ \boldsymbol{z}_N^\top \end{pmatrix}.$$

Our objective is to evaluate the least squares estimator for the linear model, i.e.

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{N} (y_i - f_{\boldsymbol{\beta}}(\boldsymbol{x}_i))^2 = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{N} (y_i - \boldsymbol{z}_i^\top \boldsymbol{\beta})^2 .$$

Hence we need to minimize the function $g(\boldsymbol{\beta}) = (y_i - \boldsymbol{z}_i^\top \boldsymbol{\beta})^2$ with respect to $\boldsymbol{\beta}$, while treating $\boldsymbol{z}_i, y_i$ as constants. Since $g$ is a quadratic function, we know that there exists a minimum. To find it, we compute the gradient of $g$ as

$$\begin{aligned} \nabla g(\boldsymbol{\beta}) &= \begin{pmatrix} \frac{\partial g}{\partial \beta_1}(\boldsymbol{\beta}) \\ \frac{\partial g}{\partial \beta_2}(\boldsymbol{\beta}) \\ \vdots \\ \frac{\partial g}{\partial \beta_{D+1}}(\boldsymbol{\beta}) \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^N 2(y_i - \boldsymbol{z}_i^\top \boldsymbol{\beta})(-z_{i1}) \\ \sum_{i=1}^N 2(y_i - \boldsymbol{z}_i^\top \boldsymbol{\beta})(-z_{i2}) \\ \vdots \\ \sum_{i=1}^N 2(y_i - \boldsymbol{z}_i^\top \boldsymbol{\beta})(-z_{iD+1}) \end{pmatrix} \\ &= -2\,(\boldsymbol{z}_1 | \cdots | \boldsymbol{z}_N) \begin{pmatrix} y_1 - \boldsymbol{z}_1^\top \boldsymbol{\beta} \\ \vdots \\ y_N - \boldsymbol{z}_N^\top \boldsymbol{\beta} \end{pmatrix} = -2\mathcal{X}^\top (\boldsymbol{y} - \mathcal{X}\boldsymbol{\beta}) \end{aligned}$$

Next, we compute the Hessian of $g$:

$$\mathcal{H}_g(\boldsymbol{\beta}) = \begin{pmatrix} \frac{\partial^2 g}{\partial \beta_1 \partial \beta_1}(\boldsymbol{\beta}) & \cdots & \frac{\partial^2 g}{\partial \beta_1 \partial \beta_{D+1}}(\boldsymbol{\beta}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 g}{\partial \beta_{D+1} \partial \beta_1}(\boldsymbol{\beta}) & \cdots & \frac{\partial^2 g}{\partial \beta_{D+1} \partial \beta_{D+1}}(\boldsymbol{\beta}) \end{pmatrix}$$

The individual entries of the Hessian evaluate to

$$\frac{\partial^2 g}{\partial \beta_j \partial \beta_l} = \frac{\partial}{\partial \beta_l} \sum_{i=1}^{N} -2 z_{ij} (y_i - \boldsymbol{z}_i^\top \boldsymbol{\beta}) = \sum_{i=1}^{N} 2 z_{ij} z_{il}$$

leading to

$$\mathcal{H}_g(\boldsymbol{\beta}) = 2\mathcal{X}^\top \mathcal{X}\,.$$

From Knowledge 4.1, we know that we obtain a minimum for $\hat{\boldsymbol{\beta}}$ such that

$$\nabla_{\boldsymbol{\beta}} g(\hat{\boldsymbol{\beta}}) = \mathbf{0}$$

if $\mathcal{H}_g(\hat{\boldsymbol{\beta}})$ is symmetric positive definite. Since $\mathcal{H}_g(\hat{\boldsymbol{\beta}}) = 2\mathcal{X}^\top \mathcal{X}$ is always symmetric positive definite (since $\mathcal{X}$ has full column rank), we obtain the location $\hat{\boldsymbol{\beta}}$ of the minimum for

$$-2\mathcal{X}^\top(\boldsymbol{y} - \mathcal{X}\hat{\boldsymbol{\beta}}) = 0 \;\Leftrightarrow\; \mathcal{X}^\top\mathcal{X}\hat{\boldsymbol{\beta}} - \mathcal{X}^\top\boldsymbol{y} = 0 \;\Leftrightarrow\; \mathcal{X}^\top\mathcal{X}\hat{\boldsymbol{\beta}} = \mathcal{X}^\top\boldsymbol{y}\,.$$

Since $\mathcal{X}^\top\mathcal{X}$ is symmetric positive definite, it is invertible, the linear system of equations has exactly one solution and it holds

$$\mathcal{X}^\top\mathcal{X}\hat{\boldsymbol{\beta}} = \mathcal{X}^\top\boldsymbol{y} \;\Leftrightarrow\; \hat{\boldsymbol{\beta}} = \left(\mathcal{X}^\top\mathcal{X}\right)^{-1}\mathcal{X}^\top\boldsymbol{y}\,.$$

□

**Remark** Depending on the training data, e.g. in case of correlated input variables, the matrix $\mathcal{X}$ might not have full column rank. In this case, the above theorem no longer holds and linear regression by least squares might fail. In Chapter **??**, we will discuss a method to solve this issue. △

We would like to summarize the core ideas of linear regression by least squares as a new machine learning method:

**Method 3** (Linear regression by least squares) For given training data $\{\boldsymbol{x}_i, y_i)\}_{i=1}^N$, we compute the input data matrix $\mathcal{X}$ and the output data vector $\boldsymbol{y}$. Via least squares, we estimate the predictor $\hat{\boldsymbol{\beta}}$ for the coefficient vector of the linear model. Thereby, we obtain the predictor $\hat{Y}$ for the output as

$$\hat{Y}(\boldsymbol{X}) = f_{\hat{\boldsymbol{\beta}}}(\boldsymbol{X}) = \hat{\beta}_0 + \sum_{i=1}^{N} \boldsymbol{x}_i^\top \boldsymbol{\beta}_{1\ldots N}\,.$$

This is what is classically called *linear regression*. △

We immediately derive an algorithm for linear regression by least squares.

**Algorithm 3** (Linear regression by least squares)
**input:** $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ training data, $\boldsymbol{x}$ evaluation point
**output:** linear regression prediction $\hat{Y}(\boldsymbol{x})$

1. build matrix $\mathcal{X}$, vector $\boldsymbol{y}$

2. $\boldsymbol{b} \leftarrow \mathcal{X}^\top \boldsymbol{y}$
3. $\boldsymbol{A} \leftarrow \mathcal{X}^\top \mathcal{X}$
4. solve $\mathcal{A}\hat{\boldsymbol{\beta}} = \boldsymbol{b}$ using Cholesky factorization
5. $\boldsymbol{z} \leftarrow \begin{pmatrix} 1 \\ \boldsymbol{x} \end{pmatrix}$
6. $\hat{Y}(\boldsymbol{x}) \leftarrow \boldsymbol{z}^\top \hat{\boldsymbol{\beta}}$
7. return $\hat{Y}(\boldsymbol{x})$

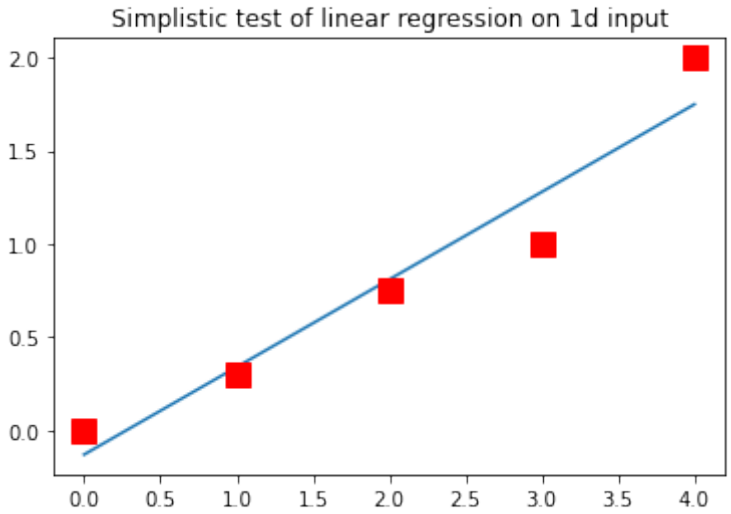
The computational complexity to compute $\hat{\boldsymbol{\beta}}$, i.e. for the *training*, is $O(D^3 + D^2 N)$. The evaluation of the linear model has a complexity of $O(D)$.

We give further examples.

**Example 4.2** In this example, we consider linear regression on one-dimensional input data. The training data for this task is

$$\mathcal{T} = \{(0,0), (1,0.3), (2,0.75), (3,1), (4,2)\}\,.$$

In the following figure, the training data is indicated by red boxes, while the predictor found using linear regression by least sqares is given as the blue graph.

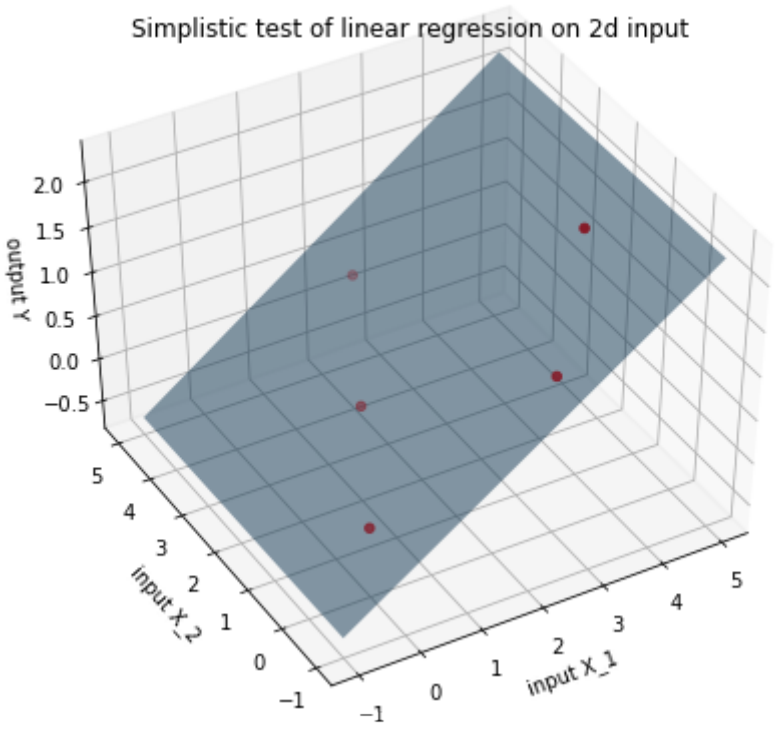



This linear regression task including the shown figure can be reproduced and modified here: launch binder. △

**Example 4.3** This example shows linear regression on two-dimensional input data. The training data is prescribed as

$$\mathcal{T} = \left\{\left((0,0)^\top, 0\right), \left((1,2)^\top, 0.3\right), \left((2,4)^\top, 0.75\right), \left((3,0)^\top, 1\right), \left((4,1)^\top, 2\right)\right\}$$

Below, the training data is given as red boxes, while the predictor computed via linear regression is the blue plane.




This linear regression task including the shown figures can be reproduced and modified here: launch binder. △

### 4.2.1 Statistical analysis of linear regression

In machine learning, we will in almost all cases have imperfect training data. A typical example for this is training data that is collected using some measurement device. Such data will always have a measurement error. More complex examples exist. This fundamental observation induces the treatment of inputs and outputs as random variables. In some sense, we need to accept that all our predictions will be influenced by the noise in the training data and other factors.

This section analyses the impact of statistical variations in the training data on the predictor constructed in linear regression by least squares. More specifically, we study a very strongly simplified model for such statistical varaiations: We start from a set of $N$ fixed measurements $\{\boldsymbol{x}_i\}_{i=1}^N$, i.e. the inputs will be assumed to be perfectly given. Moreover, we assume that the output is generated by a model of the form

$$Y = \boldsymbol{Z}^\top \boldsymbol{\beta} + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2)\,.$$

This means that we, in principle, assume that the output is exactly given by a linear model with coefficient vector $\boldsymbol{\beta}$, however, we have an additive, normally distributed noise

on the output. Using this model, we then generate the training data as

$$\mathcal{T} = \{(\boldsymbol{z}_i, \boldsymbol{z}_i^\top \boldsymbol{\beta} + \varepsilon_i)_{i=1}^N\}$$

with $\boldsymbol{z}_i = (1, \boldsymbol{x}_i^\top)^\top$. The result is training data with a very simple to study variation on it. The result of our analysis is given by

**Theorem 4.2** Let $\boldsymbol{X}$, $Y$ be in- and output and $\{\boldsymbol{x}_i\}_{i=1}^N$ be a set of fixed measurements. We assume the outputs $y_i$ to be given by a linear model with additive noise of the form

$$y_i = \beta_0 + \sum_{j=1}^{D} x_{ij}\beta_j + \varepsilon_i, \quad \text{for all } i = 1, \ldots, N$$

where we assume for $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_N)^\top$ that it holds

$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\boldsymbol{0}, \sigma^2 \mathcal{I}) \,.$$

By drawing a sample of $\mathcal{N}(\boldsymbol{0}, \sigma^2\mathcal{I})$ and evaluating the model above we obtain a training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$. Applying linear regression by least squares to the training set, we obtain a predictor $\hat{\boldsymbol{\beta}}$ for the linear regression coefficients that follows

$$\hat{\boldsymbol{\beta}} \sim \mathcal{N}(\boldsymbol{\beta}, (\mathcal{X}^\top \mathcal{X})^{-1}\sigma^2) \,,$$

i.e. it is normally distributed with mean $\boldsymbol{\beta}$ and covariance matrix $(\mathcal{X}^\top \mathcal{X})^{-1}\sigma^2$.

We immediately come to an example, to gain a deeper understanding of the discussed setting.

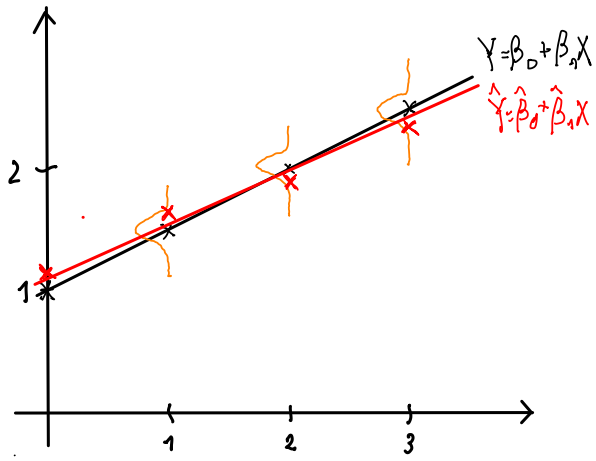
**Example 4.4** Let us consider one-dimensional input measurements

$$x_1 = 0, x_2 = 1, x_3 = 2, x_4 = 3$$

and we assume to have a given linear model with coefficients $\beta_0$ and $\beta_1 = 0.5$. Then we build output data via the model

$$Y = 1 + 0.5x + \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. The below figure demonstrates the situation.




The black graph corresponds to the original predictor. The data that we obtain and which is shown by the red crosses includes a small perturbation. Then, we use linear regression for that data and obtain a new (perturbed) predictor

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 x\,.$$

The red graph represents one such perturbed predictor. It is only *one such* predictor, since by sampling other noise values, we could certainly get other predictors. The above theorem now tells us, how the new coefficients $\hat{\boldsymbol{\beta}}$ will look like. Specifically, we do not need to go the detour via sampling the noise, but can immediately – from given input measurements and original coefficient vector $\boldsymbol{\beta}$ – sample possible $\hat{\boldsymbol{\beta}}$s. Later, we will use this to visualized the uncertainty in the predictor. △

Now that we have gained an intuition for the statement, we would like to proceed to the proof. Note that the proof might look complex due to its length, however it is just a composition of very simple small calculations:

Proof

Let the situation described in the statement of Theorem 4.2 be given. Starting from the training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ (with noise in the outputs), we compute the predictor $\hat{\boldsymbol{\beta}}$ for the coefficients in the linear model as

$$\hat{\boldsymbol{\beta}} = \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{y}\,,$$

where $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$, as before. However, we also know that the output vector $\boldsymbol{y}$ is computed from the assumed to be given linear model with coefficients $\boldsymbol{\beta}$, i.e. we have

$$\boldsymbol{y} = \mathcal{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}\,.$$

Combining the previous statements we thus obtain

$$\begin{aligned}
\hat{\boldsymbol{\beta}} &= \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{y} = \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \left(\mathcal{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}\right) \\
&= \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \left(\mathcal{X}^\top \mathcal{X}\right) \boldsymbol{\beta} + \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon} \\
&= \boldsymbol{\beta} + \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon}\,.
\end{aligned}$$

Next we evaluate the expectation of $\hat{\boldsymbol{\beta}}$ to

$$\begin{aligned} \mathrm{E}\left(\hat{\boldsymbol{\beta}}\right) &= \mathrm{E}\left(\boldsymbol{\beta} + \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon}\right) = \mathrm{E}\left(\boldsymbol{\beta}\right) + \mathrm{E}\left(\left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon}\right) \\ &= \boldsymbol{\beta} + \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \mathrm{E}\left(\boldsymbol{\varepsilon}\right) = \boldsymbol{\beta} + \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \mathbf{0} = \boldsymbol{\beta}\,. \end{aligned}$$

The second equality uses the linearity of the expectation. Then, we observe that both $\boldsymbol{\beta}$ and $\left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top$ are constants, i.e. are not random. Finally, we use the known zero-expectation of the noise term $\boldsymbol{\varepsilon}$. Our first result is thus that the mean of the predictor for the coefficients is equal to the original coefficient vector that we assumed to be given to create the outputs. In other words, the noise does not affect the mean of the coefficient predictor.

Next, we evaluate the covariance matrix of $\hat{\boldsymbol{\beta}}$. As we know, it is given by

$$\mathcal{C} = \mathrm{E}\left(\left(\hat{\boldsymbol{\beta}} - \mathrm{E}\left(\hat{\boldsymbol{\beta}}\right)\right)\left(\hat{\boldsymbol{\beta}} - \mathrm{E}\left(\hat{\boldsymbol{\beta}}\right)\right)^\top\right).$$

We first work a bit on the term $\hat{\boldsymbol{\beta}} - \mathrm{E}\left(\hat{\boldsymbol{\beta}}\right)$:

$$\hat{\boldsymbol{\beta}} - \mathrm{E}\left(\hat{\boldsymbol{\beta}}\right) = \boldsymbol{\beta} + \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon} - \boldsymbol{\beta} = \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon}\,.$$

The first equality goes back to the two derivations made before. The second equality is obvious. Then we use this result to further compute the covariance matrix as

$$\begin{aligned} \mathcal{C} &= \mathrm{E}\left(\left(\left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon}\right)\left(\left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon}\right)^\top\right) \\ &= \mathrm{E}\left(\left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top \mathcal{X}\left(\mathcal{X}^\top \mathcal{X}\right)^{-1}\right) \\ &= \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \mathrm{E}\left(\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\right) \mathcal{X}\left(\mathcal{X}^\top \mathcal{X}\right)^{-\top} \end{aligned}$$

The first equality simply inserts the previous result in the definition. The second equality uses basic laws of matrix calculus. The last step uses that $\mathcal{X}$ is not random thus can be "teared" out of the expectation. On the notation side "$-\top$" is the combination of transpose and inverse.

Next, we need to deal further with the term $\mathrm{E}\left(\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\right)$. We use a small trick and "add a zero" in the sense of $\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon} - \mathrm{E}\left(\boldsymbol{\varepsilon}\right)$ and get

$$\mathrm{E}\left(\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\right) = \mathrm{E}\left(\left(\boldsymbol{\varepsilon} - \mathrm{E}\left(\boldsymbol{\varepsilon}\right)\right)\left(\boldsymbol{\varepsilon} - \mathrm{E}\left(\boldsymbol{\varepsilon}\right)\right)^\top\right) = \sigma^2 \mathcal{I}\,,$$

where $\mathcal{I}$ is the identiy matrix. Here, the last equality uses the fact that $\mathrm{E}\left(\left(\boldsymbol{\varepsilon} - \mathrm{E}\left(\boldsymbol{\varepsilon}\right)\right)\left(\boldsymbol{\varepsilon} - \mathrm{E}\left(\boldsymbol{\varepsilon}\right)\right)^\top\right)$ is just the covariance of $\boldsymbol{\varepsilon}$, which is known to be $\sigma^2\mathcal{I}$, by the definition of $\boldsymbol{\varepsilon}$.

We continue our calculation of the covariance matrix $\mathcal{C}$ with

$$\begin{aligned}\mathcal{C} &= \left(\mathcal{X}^\top\mathcal{X}\right)^{-1}\mathcal{X}^\top \mathrm{E}\left(\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top\right)\mathcal{X}\left(\mathcal{X}^\top\mathcal{X}\right)^{-\top} \\ &= \left(\mathcal{X}^\top\mathcal{X}\right)^{-1}\mathcal{X}^\top\left(\sigma^2\mathcal{I}\right)\mathcal{X}\left(\mathcal{X}^\top\mathcal{X}\right)^{-\top} \\ &= \sigma^2\left(\mathcal{X}^\top\mathcal{X}\right)^{-1}\mathcal{X}^\top\mathcal{X}\left(\mathcal{X}^\top\mathcal{X}\right)^{-\top} \\ &= \sigma^2\left(\mathcal{X}^\top\mathcal{X}\right)^{-\top} = \sigma^2\left(\mathcal{X}^\top\mathcal{X}\right)^{-1}\end{aligned}$$

In the third equality, we use that $\sigma^2$ is a scalar value. The rest is basic matrix calculus. We briefly summarize the results found so far by

$$\mathrm{E}\left(\hat{\boldsymbol{\beta}}\right) = \boldsymbol{\varepsilon}, \quad \mathcal{C} = \sigma^2\left(\mathcal{X}^\top\mathcal{X}\right)^{-1}.$$

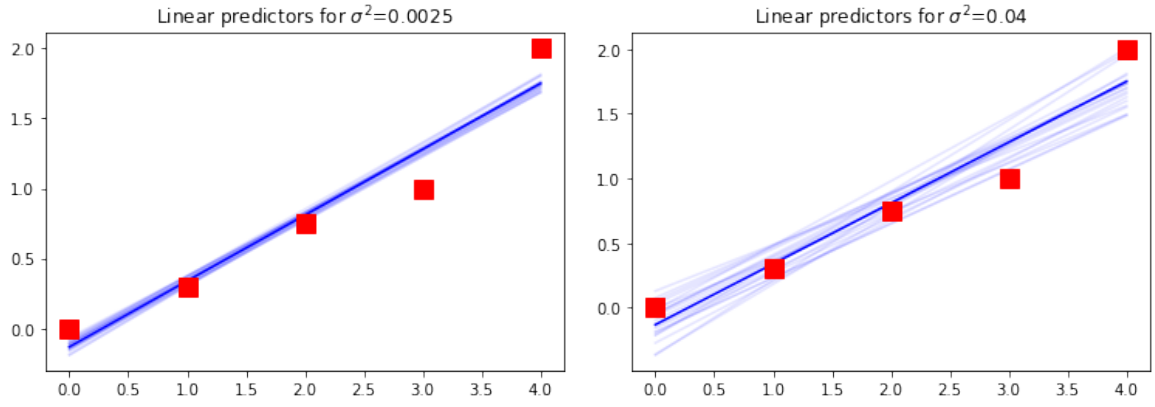
So we know the expectation and covariance of $\hat{\boldsymbol{\beta}}$. From probability theory, we further know that since $\hat{\boldsymbol{\beta}}$ is a linear function in $\boldsymbol{\varepsilon}$ and $\boldsymbol{\varepsilon}$ folows a normal distributed, also the linearly transformed $\hat{\boldsymbol{\beta}}$ follows the normal distribution, which is fully characterized by the information given above, i.e. we have

$$\hat{\boldsymbol{\beta}} \sim \mathcal{N}\left(\boldsymbol{\beta}, \left(\mathcal{X}^\top\mathcal{X}\right)^{-1}\sigma^2\right).$$

□

Let us now bring our new knowledge into action:

**Example 4.5** We continue Example 4.2. In this example we consider the linear model that we fitted into the given training data from that previous example as the *truth*. Hence, we assume that it is the best that we can do for that data. Then we further assume that we have noise with variance $\sigma^2$ on that data. In the below figures, we "sample" twenty predictors according to Theorem 4.2 once for $\sigma = 0.05$ and for $\sigma = 0.2$.




The figures nicely show, how the noise-driven predictors fluctuate around the assumed to be exact predictor. For higher variance, the fluctuations are stronger. This noise-

induced regression task including the shown figures can be reproduced and modified here: launch binder.[3] △

This should give us a feeling for the uncertainty that we should always expect in the construction of predictors. Hence, predictors never give the "truth" however incorporate several types of noises and errors. The noise studied here is the noise in the outputs. In Chapters ?? and ??, we will study other types of errors.

## 4.3 Linear regression with multiple outputs

We finish this chapter by generalizing linear regression from one-dimensional outputs to $K$-dimensional outputs. Essentially this means that we have to build $K$ linear models

$$f_{\boldsymbol{\beta}^{(k)}}(\boldsymbol{X}) = \beta_0^{(k)} + \sum_{j=1}^{D} X_j \beta_j^{(k)} \qquad\qquad k = 1, \dots, K$$

for training data

$$\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N},$$

where we specifically have $\boldsymbol{y}_i = (y_{i1}, \dots, y_{iK})^\top$.
To build such models, we first need to generalize from the simple least squares estimator, that we have seen before.

**Definition 4.4** Let $\boldsymbol{X}$ be a $D$-dimensional input, $\boldsymbol{Y}$ be a $K$-dimensional output, $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=i}^{N}$ and $f_{\boldsymbol{\gamma}} : \mathbb{R}^D \to \mathbb{R}^K$ be a prametric model. Furthermore we are given a loss function $L : \mathbb{R}^K \times \mathbb{R}^K \to \mathbb{R}$. Then a general parameter estimator is given by

$$\hat{\boldsymbol{\gamma}} = \arg\min_{\boldsymbol{\gamma}} \sum_{i=1}^{N} L(\boldsymbol{y}_i, f_{\boldsymbol{\gamma}}(\boldsymbol{x}_i))$$

We introduce the loss that we will use for linear regression with multiple outputs as an example.

**Example 4.6** (Loss for $K$-dimensional outputs) For $\boldsymbol{Y} \in \mathbb{R}^K$ we can define the loss

$$L : \mathbb{R}^K \times \mathbb{R}^K \to \mathbb{R}$$

by

$$L(\boldsymbol{y}, \boldsymbol{y}') = \sum_{k=1}^{K} (y_k - y'_k)^2 .$$

[3] The Jupyer notebook will be available as soon as the programming task for linear regression is completed.

△

We then immediately obtain the following result:

**Theorem 4.3** (Linear regression with multiple outputs) Let $\boldsymbol{X}$ be a $D$-dimensional input, $\boldsymbol{Y}$ be a $K$-dimensional output, $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=i}^N$ and $f_{\mathcal{B}} : \mathbb{R}^D \to \mathbb{R}^K$ be the linear model for $K$ outputs of the form

$$f_{\mathcal{B}}(\boldsymbol{X}) = \left(f_{\boldsymbol{\beta}^{(1)}}(\boldsymbol{X}), \ldots, f_{\boldsymbol{\beta}^{(K)}}(\boldsymbol{X})\right)^\top,$$

$$f_{\boldsymbol{\beta}^{(k)}}(\boldsymbol{X}) = \beta_0^{(k)} + \sum_{j=1}^K X_j \beta_j^{(k)}.$$

With the loss from Example 4.4, we obtain the least squares estimator

$$\hat{\mathcal{B}} = \arg\min_{\mathcal{B}} \sum_{i=1}^N \sum_{k=1}^K (y_{ik} - f_{\boldsymbol{\beta}^{(k)}}(\boldsymbol{x}_i))^2,$$

where the optimal coefficient matrix $\hat{\mathcal{B}} = \left(\boldsymbol{\beta}^{(1)} | \cdots | \boldsymbol{\beta}^{(K)}\right)$ is given with $\mathcal{Y} = (\boldsymbol{y}_1 | \cdots | \boldsymbol{y}_N)^\top$ by

$$\hat{\mathcal{B}} = (\mathcal{X}^\top \mathcal{X})^{-1} \mathcal{X}^\top \mathcal{Y}.$$

The statement is an obvious extension of Theorem 4.1 and is given without proof.

**Remark** Analysing the above theorem, we observe that we indeed can solve the full $K$-dimensional linear regression problem by solving $K$ one-dimensional linear regression problems

$$\hat{\boldsymbol{\beta}}^{(k)} = (\mathcal{X}^\top \mathcal{X})^{-1} \begin{pmatrix} y_{1k} \\ \vdots \\ y_{Nk} \end{pmatrix}.$$

△

This concludes our initial discussion on linear regression. Nevertheless, we will continue to use this method for studies of properties of machine learning methods.

# 5. (Stochastic) Gradient descent

Computing the estimator for the coefficients of a parametric model is usually based on the minimization of a functional $J : \mathbb{R}^S \to \mathbb{R}$. In case of the estimator that we introduced for linear regression by least squares (for one-dimensional outputs), the minimization task was

$$\hat{\boldsymbol{\omega}} = \arg\min_{\boldsymbol{\omega}} J(\boldsymbol{\omega}) = \arg\min_{\boldsymbol{\omega}} \sum_{i=1}^{N} L_2(y_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i)) \,,$$

thus we had to minimize the functional

$$J(\boldsymbol{\omega}) = \sum_{i=1}^{N} L_2(y_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i)) \,.$$

Since we applied this estimator to the very simple linear model and since the least squares loss has nice mathematical properties, we could find the unique global minimum by an explicit solution of linear system of equations.

In many other state of the art machine learning models, including even simple classification methods or the widely used artificial neural networks, we will no longer be able to solve the respective minimization problem in an exact way. Reasons for this are the strong non-linearity of some of the models that we are going to study, a strong underdetermination of the model paramters (i.e. we often have more parameters than training data) and as a result the existence of many, many local minima of the functional $J$. Even worse, finding the actual global minimum is usually impossible. As a consequence, we will have to accept that solving the minimization problem exactly will usually be impossible.

This chapter will be concerned with a very simplistic but (surprisingly) very powerful approach to minimize functionals. *Gradient descent* methods only requires the existence of a gradient of the functional.[1)] Based on this information, these methods provide a greedy approach to look for the minimum.

We will start by introducing the core technique, namely *gradient descent* with its counterpart *batch gradient descent*. Then we will extend this approach to *stochastic gradient descent* and *mini-batch gradient descent*.

[1)] In practical applications, like neural networks with non-smooth *activations*, one usually even does not require to have the gradient been properly defined everywhere.

## 5.1 Gradient descent

The starting point for the introduction of the gradient descent approach is the following theorem.

**Theorem 5.1** Let $J : \mathbb{R}^S \to \mathbb{R}$ be a functional, i.e. (here) a mapping from a higher-dimensional vector to a real number. We assume that $J(\boldsymbol{\omega})$ ($\boldsymbol{\omega} = (\omega_1, \ldots, \omega_S)^\top$) is differentiable in each component $\omega_i$ of its input, such that the gradient

$$\nabla J(\boldsymbol{\omega}) := \begin{pmatrix} \frac{\partial J}{\partial \omega_1} \\ \vdots \\ \frac{\partial J}{\partial \omega_S} \end{pmatrix} (\boldsymbol{\omega})$$

exists. The gradient at a specific location $\boldsymbol{\omega}'$, i.e. $\nabla J(\boldsymbol{\omega}')$, is the vector that points at $\boldsymbol{\omega}'$ into the direction in which the function $J$ has its strongest increase (when "moving" away from $\boldsymbol{\omega}'$).

This theorem is a classical result from calculus for higher-dimensional functions. Therefore, we won't give a proof here.

Nevertheless, knowing this result, we can immediately come up with a simplistic iterative greedy strategy to look for a (local) minimum of a given functional $J$.

**Method 4** (Gradient / steepest descent) We assume that we are given a functional $J$ and an initial guess $\boldsymbol{\omega}^{(0)}$ for the location of a (local) minimum of $J$ and we look for the location $\hat{\boldsymbol{\omega}}$ of a local minimum. In *gradient* or *steepest descent*, we iteratively apply updates

$$\boldsymbol{\omega}^{(n+1)} = \boldsymbol{\omega}^{(n)} - \eta \nabla J(\boldsymbol{\omega}^{(n)}) \,.$$

What are we doing here? We gradually change the position of the assumed local minimum and go in direction $-\nabla J(\boldsymbol{\omega})$, which is the direction of steepest *decent* of $J$. Moreover, we have to decide on the size of that update step. This step size $\eta$, which is a positive real number, is often called *learning rate* in the machine learning context.

Ideally, the method terminates when further updates don't decrease the functional anymore or, equivalently, if the magnitude of the gradient becomes very small or even zero. In that case, we have (approximately) found a *local* minimum. △

Note that many open questions remain for now: How to choose $\eta$? When to actually terminate the update process? Is is enough to have a *local* minimum, but not a *global* minimum? Let us keep the suspense of these open questions for now and continue to the algorithm.

**Algorithm 4** (Gradient decent)

**input:** $J(\boldsymbol{\omega})$ functional with existing $\nabla J(\boldsymbol{\omega})$, $\boldsymbol{\omega}^{(0)}$ initial guess, $\eta$ learning rate
$tol$ tolerance (stopping criterion)

**output:** $\hat{\boldsymbol{\omega}}$ approximated local minimizer, i.e. $\hat{\boldsymbol{\omega}} \approx \arg\min_{\boldsymbol{\omega}} J(\boldsymbol{\omega})$

1. $n \leftarrow 0$
2. while $\left\|\nabla J\left(\boldsymbol{\omega}^{(n)}\right)\right\| > tol$:
   a) $\boldsymbol{\omega}^{(n+1)} \leftarrow \boldsymbol{\omega}^{(n)} - \eta \nabla J(\boldsymbol{\omega}^{(n)})$
   b) $n \leftarrow n + 1$
3. return $\boldsymbol{\omega}^{(n)}$

The algorithm strictly follows our method description from above. In addition we have chosen the stopping criterion of comparing the norm of the gradient with a fixed tolerance.

In the following example, we apply gradient descent to linear regression by least squares.

**Example 5.1** (Gradient descent for linear regression by least squares) Let $\mathcal{T} = \{\boldsymbol{x}_i, y_i)\}_{i=1}^N$ be training data and $f_{\boldsymbol{\beta}}$ be the linear model. As discussed before, the functional for least squares parameter estimation is then given by

$$J(\boldsymbol{\beta}) = \sum_{i=1}^{N} L_2(y_i, f_{\boldsymbol{\beta}}(\boldsymbol{x}_i)) \,.$$

From the proof of Theorem 4.1, we further know that its gradient is given by

$$\nabla J(\boldsymbol{\beta}) = -2\mathcal{X}^\top (\boldsymbol{y} - \mathcal{X}\boldsymbol{\beta}) \,,$$

with $\mathcal{X}$ and $\boldsymbol{y}$ as in Theorem 4.1. As a consequence, gradient descent for linear regression by least squares has the update rule

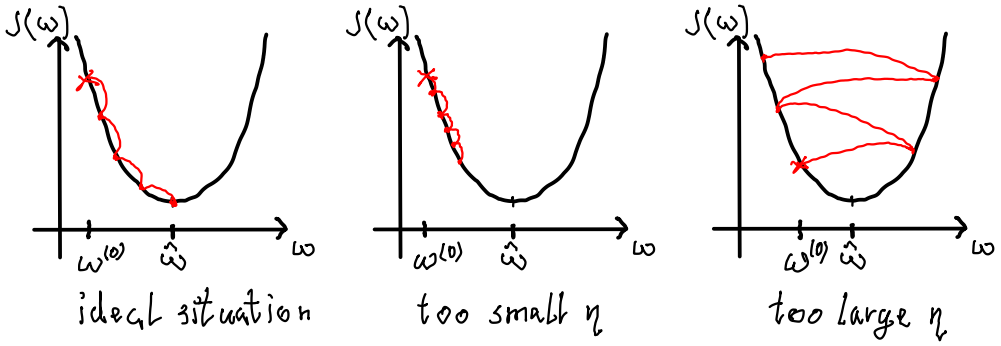
$$\boldsymbol{\beta}^{(n+1)} = \boldsymbol{\beta}^{(n)} + \eta 2\mathcal{X}^\top \left(\boldsymbol{y} - \mathcal{X}\boldsymbol{\beta}^{(n)}\right)$$

and we still have to choose some initial guess $\boldsymbol{\beta}^{(0)}$, the learning rate $\eta$ and the tolerance $tol$. △

So why did we not use gradient descent for training in linear regression by least squares? As discussed before, in linear regression we know the exact solution with the *global* minimum. Since the minimum is unique in linear regression, gradient descent would ultimately also find it, however it is often much slower and more unreliable than the direct approach.

Some reasons for these issues are summarized in the following remark, which goes (including the figures) back to [5].

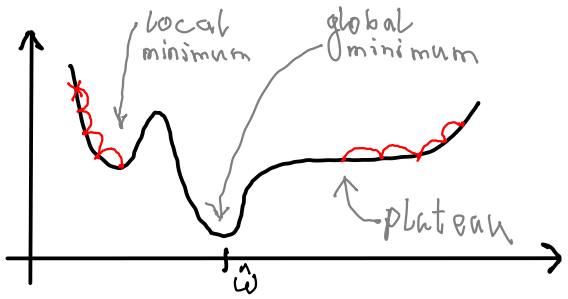
**Remark** (Issues of gradient descent) One big challenge in gradient descent is the choice of a proper learning rate $\eta$. This is illustrated by the three figures below.




On the left-hand side, we have the ideal situation. Within a few steps, the gradient descent method finds even the global minimum. In the center, we see what happens if we choose a too small learning rate. While we might find the minimum at some point, we will require many steps to get there. On the right-hand side, we see what happens for a too large learning rate. Indeed the process can even diverge completely.

One solution to overcome these issues is by adapting the learning rate during the training process. Here, a typical heuristics is to first start with a larger learning rate and then to reduce the learning rate over time when coming closer to a (local) minimum.

While the above figures give a first intuition, we should be aware that the situation is even more complex. Indeed, for non-linear, non-convex functionals $J(\boldsymbol{\omega})$, we can easily have many local minima and even plateaus:




As shown above, the gradient descent would both terminate in a local minimum and in a plateau, while both results might be far off the actual global minimum. Even worse, we should be aware that in models with 10000s of parameters, as easily found in e.g. artificial neural networks, we can also have 10000s or much more local minima.

The above proposed heuristics does not really help to overcome the issue of many local minima. We will for now keep that in the back of our mind. △

## 5.2 Batch gradient descent

In nowadays literature on machine learning, there is often the notion of *batch gradient descent*. Even though gradient descent and batch gradient descent, are the same algo-

rithm, it is still helpful to discuss the small conceptual change that is applied when going to *batch* gradient descent.

Indeed, the notion "batch" gives a connection to the training data that is used in gradient descent:

**Remark** (From gradient descent to batch gradient descent) To better understand this connection we assume that most training (i.e. parameter estimation) problems can be expressed in terms of the parameter estimation that we introduced in Definition 4.4,

$$\hat{\boldsymbol{\omega}} = \arg\min_{\boldsymbol{\omega}} \sum_{i=1}^{N} L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i)) \,.$$

Here, the functional to be minimized is

$$J(\boldsymbol{\omega}) = \sum_{i=1}^{N} L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i))$$

with a general loss $L$ and we can derive (by linearity of the gradient) the update step as

$$\boldsymbol{\omega}^{(n+1)} = \boldsymbol{\omega}^{(n)} - \eta \nabla J(\boldsymbol{\omega}^{(n)}) = \boldsymbol{\omega}^{(n)} - \eta \sum_{i=1}^{N} \nabla L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}^{(n)}}(\boldsymbol{x}_i)) \,.$$

By this, we have just introduced a connection between the minimization and the training data $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$. Essentially, a single update step can be understood as $N$ small update steps that use the gradient of the loss term evaluated at each training sample. Since classical gradient descent uses *all* training samples, it is called *batch* gradient descent, i.e. it uses the full batch of training samples. △

Let us introduce the batch gradient descent algorithm.

**Algorithm 5** (Batch gradient decent)

**input:** $f_{\boldsymbol{\omega}}$ parametric model, $\boldsymbol{\omega}^{(0)}$ initial guess, $\eta$ learning rate, $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N}$ training data, *tol* tolerance (stopping criterion)

**output:** $\hat{\boldsymbol{\omega}}$ approximated local minimizer, i.e. $\hat{\boldsymbol{\omega}} \approx \arg\min_{\boldsymbol{\omega}} J(\boldsymbol{\omega})$

1. $n \leftarrow 0$
2. while $\left\|\nabla J\left(\boldsymbol{\omega}^{(n)}\right)\right\| > tol$:
   a) $\boldsymbol{\omega}^{(n+1)} \leftarrow \boldsymbol{\omega}^{(n)} - \eta \sum_{i=1}^{N} \nabla L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}^{(n)}}(\boldsymbol{x}_i))$
   b) $n \leftarrow n + 1$
3. return $\boldsymbol{\omega}^{(n)}$

The attentive reader will immediately conclude from this terminology discussion that there might be gradient descent methods that will not use the "full batch". And this is what we discuss next.

## 5.3 Stochastic gradient descent

As we learned before, (batch) gradient descent might "get stuck" in a local minimum or even in a plateau. Moreover, the method is rather expensive since (as visible in batch gradient descent) we have to evaluate the gradient of the loss for each training sample. Hence, with growing training sample size, this approach might become intractable.

In this section, we are going into the other extreme, were we only use one (random) training sample per update step, giving rise to the *stochastic gradient descent*.

**Method 5** (Stochastic gradient descent) We are given a training data set $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, a loss $L$, a parametric model $f_{\boldsymbol{\omega}}$, a learning rate $\eta$ and an initial guess $\boldsymbol{\omega}^{(0)}$ and are again looking for an approximation to the minimizer $\hat{\boldsymbol{\omega}}$ of

$$J(\boldsymbol{\omega}) = \sum_{i=1}^{N} L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i)) \,.$$

In stochastic gradient descent, in comparison to batch gradient descent, we randomly pick per update step *one* training sample $(\boldsymbol{x}_r, \boldsymbol{y}_r)$ and perform the update

$$\boldsymbol{\omega}^{(n+1)} = \boldsymbol{\omega}^{(n)} - \eta \nabla L(\boldsymbol{y}_r, f_{\boldsymbol{\omega}^{(n)}}(\boldsymbol{x}_r)) \,.$$

△

We immediately give the algorithm.

**Algorithm 6** (Stochastic gradient decent)

**input:** $\mathcal{T} \in \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ training data, $f_{\boldsymbol{\omega}}$ parametric model, $L$ loss, $\boldsymbol{\omega}^{(0)}$ initial guess, $\eta$ learning rate, $tol$ tolerance

**output:** $\hat{\boldsymbol{\omega}}$ approximated minimizer

1. $n \leftarrow 0$
2. while $\left\|\nabla J\left(\boldsymbol{\omega}^{(n)}\right)\right\| > tol$:
   a) pick $(\boldsymbol{x}_r, \boldsymbol{y}_r) \in_{rnd} \mathcal{T}$
   b) $\boldsymbol{\omega}^{(n+1)} \leftarrow \boldsymbol{\omega}^{(n)} - \eta \nabla L(\boldsymbol{y}_r, f_{\boldsymbol{\omega}^{(n)}}(\boldsymbol{x}_r))$
   c) $n \leftarrow n + 1$
3. return $\boldsymbol{\omega}^{(n)}$

In step 2. a) "$\in_{rnd}$" stands for a random selection procedure. Note, that it is advisable to perform this randomization such that the whole data set is walked through before some training sample is used again. This is also how stochastic gradient descent is usually implemented.

Let us close this section by briefly discussing the advantages and disadvantages of stochastic gradient descent.

**Remark** (Discussion of stochastic gradient descent) The obvious advantage of stochastic gradient descent is that it is cheap in each update step. Moreover, the randomization allows us to "escape" local minima. To understand this, we need to analyze, what happens in one update step: By only picking one training sample, we do not necessarily go into the direction from the actual steepest descent but only into the decent direction that is implied by that training sample. Therefore, we always deviate from the classical path of gradient descent. Roughly speaking, we somewhat "jump around". This, however, helps us often to "escape" the maybe suboptimal trajectory of classical gradient descent.

While decreasing the problem of local minima, we, on the other hand, might need much more steps (in terms of the number of training samples used) than the (batch) gradient descent, since the steps no longer give the clear locally optimal direction. Moreover, when coming close to an actual minimum, we might not finally reach it due to the "jumping" nature of the method.[2)] △

## 5.4 Mini-batch gradient descent

The last remark of the previous section showed that stochastic gradient descent helps to overcome the local minima issue, but might be too extreme, when it comes to the number of required steps and the inaccuracy of each step. With the *mini-batch gradient descent*, we introduce a hybrid approach between batch and stochastic gradient descent.

**Method 6** (Mini-batch gradient descent) In comparison to stochastic gradient descent, *mini-batch gradient descent* does not only pick one single random training sample but $N_b$ random training samples from the training set $\mathcal{T}$. The resulting random sub-sample of $\mathcal{T}$ is called *mini-batch*. Correspondingly, $N_b$ is the *(mini-)batch size*. As update step, we take a modified batch gradient descent update step

$$\boldsymbol{\omega}^{(n+1)} = \boldsymbol{\omega}^{(n)} - \eta \sum_{(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{T}_{rnd}} \nabla L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}^{(n)}}(\boldsymbol{x}_i))$$

△

We immediately proceed to the algorithm.

**Algorithm 7** (Mini-batch gradient decent)
**input:** $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ training data, $f_{\boldsymbol{\omega}}$ parametric model, $L$ loss, $\boldsymbol{\omega}^{(0)}$ initial guess, $N_b$ batch size, $\eta$ learning rate, *tol* tolerance
**output:** $\hat{\boldsymbol{\omega}}$ approximated minimizer

1. $n \leftarrow 0$

[2)] Note that this last issue can be reduced by the earlier proposed heuristics to decrease the learning rate during the gradient descent application.

2. while $\left\|\nabla J\left(\boldsymbol{\omega}^{(n)}\right)\right\| > tol$:
   a) pick $\mathcal{T}_{rnd} \subset_{rnd} \mathcal{T}$, such that $|\mathcal{T}_{rnd}| = N_b$
   b) $\boldsymbol{\omega}^{(n+1)} \leftarrow \boldsymbol{\omega}^{(n)} - \eta \sum_{(\boldsymbol{x}_i, \boldsymbol{y}_i) \in \mathcal{T}_{rnd}} \nabla L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}^{(n)}}(\boldsymbol{x}_i))$
   c) $n \leftarrow n + 1$
3. return $\boldsymbol{\omega}^{(n)}$

In 2. a), the notation "$\subset_{rnd}$" stands for a random *subset* selection. As for stochastic gradient descent, this random subset selection is usually implemented such that sequential subset selections of this kind "traverse" the full training data set.

**Remark** (Discussion of mini-batch gradient descent) Mini-batch gradient descent provides a trade-off between stochastic gradient descent and batch gradient descent. Ideally, it combines the advantages of both methods. It should also be clear that for $N_b = N$ we obtain batch gradient descent and for $N_b = 1$ we obtain stochastic gradient descent. From an implementation perspective, mini-batch gradient descent also has the advantage that the $N_b$ updates in one descent step can be computed in parallel. △

In the literature, the notion of *training epochs* has been established.

**Remark** (Training epochs) One *training epoch* in batch / stochastic / mini-batch gradient descent corresponds to one sweep through the full training data set. Thereby we obtain that

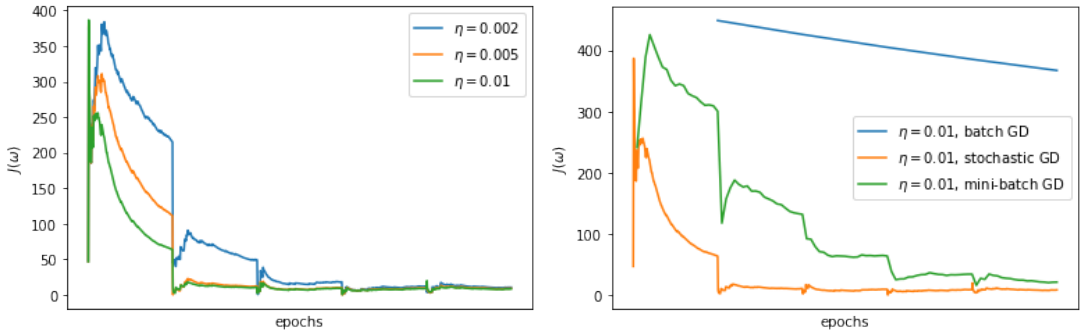
- 1 step of batch gradient descent corresponds to 1 training epoch,
- $N$ steps of stochastic gradient descent correspond to 1 training epoch, and
- $\frac{N}{N_b}$ steps of mini-batch gradient descent correspond to 1 training epoch.

This notion is thus helpful to make training in terms of its cost in the training data set size comparable between the methods. △

As a practical side remark, one should be aware that the scaling of the different inputs might have an impact on the convergence of gradient descent methods. Therefore, as always, it is advisable to rescale the inputs, before using gradient descent methods, see Section 2.4.

We finish this chapter by a practical application example that compares all gradient descent methods studied in this chapter.

**Example 5.2** (Gradient descent comparison for linear regression) We consider the first 200 training samples from the energy efficiency data set discussed in Example 2.2. For this data set, we compute the predictor from linear regression by least squares using the gradient descent methods discussed so far.




The plot on the left hand side shows the decay of the functional $J(\boldsymbol{\omega})$ in stochastic gradient descent for a growing number of epochs and three different learning rates. Here, the larger the learning rate the more quickly the method converges towards the minimum. It should be noted, however, that the minimum search for models, which are not as simple as the linear model with least squares loss, could easily break down (i.e. explode) for such large training rates. [3)]

On the right-hand side, batch gradient descent, stochastic gradient descent and mini-batch gradient descent with $N_b = 10$ are compared. While in principle, batch gradient descent is giving the best possible (local) update direction. The purely randomized stochastic gradient descent outperforms batch gradient clearly. This is one of the interesting properties of stochastic gradient descent that shows up in practice: It is often much faster due to the randomization. Mini-batch gradient descent as a hybrid method between batch and stochastic gradient descent, gives also convergence results that are in between both methods.

This gradient descent example including the shown figures can be reproduced and modified here: launch binder.

△

[3)] Note that the figure also shows some sudden drops in the magnitude of the functional. This specifically happens once after each epoch. It seems like this is an artifact from a not documented functionality in the software (*Keras*) that is used to implement this example.

# 6. Estimation of prediction error

Until now, we have primarily delt with the introduction and training of predictors in supervised machine learning. However, we mostly ignored questions related to the "quality" of a given predictor. If we were to have only one single predictor type, asking for its quality for given training data would be mostly an academic question. However, in practice, we have many different predictor models among which we can choose. Moreover, many of these models have additional parameters (e.g. the $k$ in kNN regression) that further influence the model and have an impact on the prediction quality. Therefore, it is fundamental to develop a precise understanding of "quality" of a given machine learning model. This is achieved by studying the prediction error.

In this chapter, we develop the necessary terminology and practical tools to assess the quality of a given predictor model. Specifically, we will learn to distinguish between the *training error* and the *generalization error*, which are both fundamental notions in the field. After having studied these notions, we will develop practical tools and algorithms to explicitely measure these errors for given data.

## 6.1 Training error

Let us jump immediately into our first definition.

**Definition 6.1** Let $\boldsymbol{X}$, $\boldsymbol{Y}$ quantitative input and output, $\rho(\boldsymbol{x}, \boldsymbol{y})$ their joint density and $f$ be a function to predict $\boldsymbol{Y}$ from $\boldsymbol{X}$. Moreover, we are given a training set $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ and a loss function $L : R_Y \times R_Y \to \mathbb{R}$. The **training error** is given by

$$TE(f, \mathcal{T}) = \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{y}_i, f(\boldsymbol{x}_i)) \,.$$

For a given training set $\mathcal{T}$, the training error is the average loss between predictions on the input measurements of the training set and the output measurements of the training data set. Alternatively, we could say that it is the average deviation of the predictor from the given training data.

Where have we seen the training error before? Besides of the normalizing constant $\frac{1}{N}$, the training error is exactly the term that we minimize in the estimators from Definitions 4.3 and 4.4 studied, so far. Thus essentially, these estimators look for coefficient vectors such that the resulting model will minimize the training error.[1)]

At this point we need to make an important

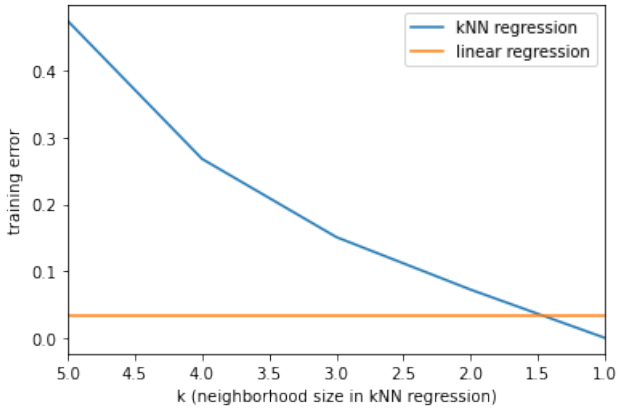
**Remark** (Knowledge gained from training error) It is crucial to notice that the training error makes *no* statement on predictions that are made for points that are *not* part of the training set. Hence, even though we might see a very low training error for some model, its predictions might be far off from what we expect.[2)] △

We should give an example for training error evaluation and for the just made remark. Also, we will use the example to introduce an important notion.

**Example 6.1** (Training error vs. model complexity) In this example, start with the very simple training data

$$\mathcal{T} = \{(0,0), (1,0.3), (2,0.75), (3,1), (4,2)\}\,.$$

that we used to create a kNN regression model and a linear regression model in Examples 3.4 and 4.2. We again build these models. However, this time, we evaluate their training error over the full training set $\mathcal{T}$. The following figure gives the results of this evaluation.




How to read and interpret this plot? First of all, we need to notice that the linear model of linear regression is, after training, free of any further parameter. In contrast, the kNN regression predictor still has the neighborhood size $k$ as a parameter to choose. Therefore, we plot the kNN training error depending on the neighborhood size, while the training error of linear regression is just a constant.

---

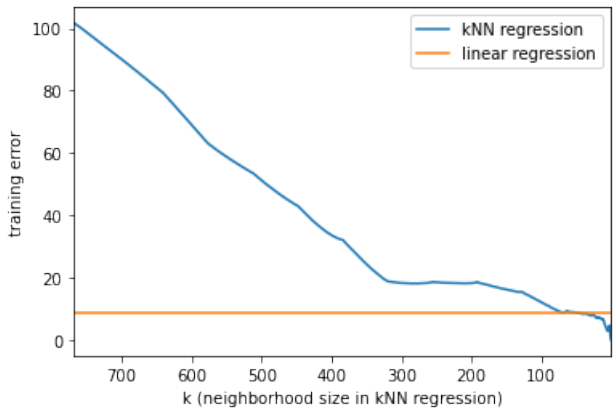
[1)] In fact, we could also use exactly the training error in these estimators, as the normalizing constant does not have an effect on the minimization process. We only stick to the estimators without the normalizing constant to comply with literature in the field, e.g. [6].

[2)] This observation will soon lead to the discussion of the complementing *generalization error* that exactly studies this situation.

The attentive reader will also have notice that the axis labels on the horizontal axis *de*crease instead of increasing. The reason for this choice goes back to a concept that we did not discuss in detail before, namely the *model complexity*. With this notion, we, roughly speaking, describe the ability of a given predictor model to adapt to the training data. Intuitively, kNN regression shows a low "adaptation" to the data for large neighborhood sizes (i.e. there is a lot of averaging happening), while it shows a strong "adaptation" to the data for small neighborhood sizes. This also becomes apparent when looking at the results of the training data evaluation. For large neighborhood sizes, we obtain a large training error, while for small neighborhood sizes, we obtain a very small error (hence we adapt much better). Even more, for $k = 1$, the training error is exactly zero. However, why does that make us flipping the horizontal axis? Essentially, it is quite common to show from left to right growing model complexity, which is what we obtain for decreasing $k$ in kNN regression.

Now that we can read the results in the above plot, how to interpret them? Moreover, is it just this data set that gives us these results, hence is there some consistent observation that we can derive from that figure?

To answer first the second question, we repeat our experiment for a real world data set, the Energy Efficiency Data Set discussed in Example 2.2. The result of this study is given below.




Indeed, there seems to be a first indication that the previous result is not found "by chance". Seemingly, for growing model complexity, the training error becomes consistently smaller. At the same time, the linear model always gives us the same training error due to its constant model complexity.

While there are certainly many open questions on this observation, we for now close this example with the remark that the first example including the shown figure can be reproduced and modified here launch binder, while the second example including the shown figure can be reproduced and modified here launch binder. △

One easy conclusion from the above example would be to always pick the model with the highest model complexity, giving us the smallest possible training error. This leads us immediately to

**Remark** (Overfitting) Picking the model with the lowest training error does not have to give us the best predictions for unseen input data. Let us briefly consider the kNN regression model for this discussion. For a neighborhood size of $k = 1$ it will always give us a training error of zero, as it exactly picks the training sample as prediction. However, should we therefore always pick $k = 1$? The answer is "No!". Indeed, for $k = 1$ the kNN model does not at all consider any relationship within the data. As we will see, this makes the kNN model with $k = 1$ a very bad predictor.

With this simple argument, we might (for now) accept that it can happen that we have models that have a very small training error, since they fit very well to the training data. However, at the same time, these models are very bad predictors for data that is not part of the training set. We call this situation *overfitting*. △

Note that we are far away from being done with the discussion of *overfitting*, *model complexity*, etc. Indeed, we will even not only discuss this empirically, as in this chapter, but will also give theory on that in the next chapter. Later chapters will also cover techniques to overcome overfitting and related issues.

**Remark** (Is training error irrelevant?) We have established that the training error might give us wrong indications towards the predictive properties of a model, since it only measures the error on the training set. Our natural conclusion could be to simply ignore the training error as a measure for the quality of a model.

However, that would also be a wrong approach, since, indeed, the training error is the only information that is present during training time. Hence, whenever we e.g. estimate the coefficients of a model, we can only do this estimation on the training data and only the training error will guide us to an optimal choice of coefficients. Therefore, the training error is far away from being irrelevant. Still, we need to *additionally* consider the error that a model makes on data that is not part of the training data set. △

This remark brings us to the introduction of the *generalization error*.

## 6.2 Generalization error

In this section, we would like to formally introduce the *generalization error* as a measure for the deviation of the predictor model from the "ground truth" outside of the training data locations. This introduction is still on the theoretical side as an extension to notions that we know from statistical decision theory. The practical implementation of the evaluation of the generalization error is discussed in the subsequent section.

Let us first recall that we introduced the expected (squared) prediction error in regression as

**Definition** (Expected (squared) predition error in regression) Let $\boldsymbol{X}$, $\boldsymbol{Y}$ be input and quantitative output with joint density $\rho(\boldsymbol{x}, \boldsymbol{y})$ and $f : \mathbb{R}^D \to \mathbb{R}^K$ a function to predict

$\boldsymbol{Y}$ from $\boldsymbol{X}$. For the squared loss, the **expected (squared) prediction error** or **risk of** $f$, $\text{EPE}(f)$, is defined as

$$\begin{aligned}\text{EPE}(f) &= \text{E}_{\boldsymbol{X},\boldsymbol{Y}}\left(L_2(\boldsymbol{Y}, f(\boldsymbol{X}))\right) \\ &= \int_{\mathbb{R}^K}\int_{\mathbb{R}^D} L_2(\boldsymbol{y}, f(\boldsymbol{x}))\rho(\boldsymbol{x},\boldsymbol{y})\, d\boldsymbol{x}d\boldsymbol{y}\,.\end{aligned}$$

Certainly, for a general loss function $L$, we similarly can describe the general expected prediction error (i.e. not just for the $L_2$ loss) by

$$\text{EPE}(f) = \text{E}_{\boldsymbol{X},\boldsymbol{Y}}\left(L(\boldsymbol{Y}, f(\boldsymbol{X}))\right)\,.$$

As discussed before, this definition of a prediction error only considers a fixed, given function $f$, while the real regression task would specifically consider a predictor that is built from training data.

The following definition closes this gap.

**Definition 6.2** (Generalization error) Let $\boldsymbol{X}$, $\boldsymbol{Y}$ quantitative input and output, $\rho(\boldsymbol{x},\boldsymbol{y})$ their joint density and $L : \mathbb{R}^K \times \mathbb{R}^K \to \mathbb{R}$ a loss function. Moreover, we are given a training set $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ that is a sample of the random variable $T$. Finally $f_{\mathcal{T}}$ is a function to predict $\boldsymbol{Y}$ from $\boldsymbol{X}$ that was trained using $\mathcal{T}$.

The **generalization / test error** is given by

$$\begin{aligned}GE(f,\mathcal{T}) &= \text{E}_{\boldsymbol{X},\boldsymbol{Y}|T}\left(L(\boldsymbol{Y}, f_T(\boldsymbol{X}))|T=\mathcal{T}\right) \\ &= \int_{\mathbb{R}^K}\int_{\mathbb{R}^D} L(\boldsymbol{y}, f_{\mathcal{T}}(\boldsymbol{x}))\rho(\boldsymbol{x},\boldsymbol{y}|\mathcal{T})d\boldsymbol{x}d\boldsymbol{y}\,.\end{aligned}$$

If we look carefully into the definition, we immediately observe that the generalization error is just the expected prediction error extended by a notion to have the predictor being dependent on some given training data. As we are in a statistical setting, we also no longer just consider the expectation over $\boldsymbol{X}$ and $\boldsymbol{Y}$, however we have to further understand $\mathcal{T}$ as a sample from some random variable $T$ and have to condition the expectation on the explicit choice of this sample $\mathcal{T}$. The random variable $T$ itself describes the set of $N$ random vector tuples $\{(\boldsymbol{X}_i, \boldsymbol{Y}_i)\}_{i=1}^N$, where each of these tuples independently follows the joint density $\rho(\boldsymbol{x},\boldsymbol{y})$.

Some reader might observe that the proposed generalization error is not just evaluated in points that are not part of the training set, but everywhere. We briefly discuss this in

**Remark** ($GE(f, \mathcal{T})$ definition vs. training error) In the last section, we stressed a lot the differentiation between an error that is only evaluated on the training data, i.e. the *training error* and an error that is evaluated on points that are not part of the training set. We anticipated that this would be the *generalization error*.

Now, the reader might get confused by the above definition of a generalization error that indeed evaluates the error as a mean over over all points, including the training points. The reason for this lies in the continuous / statistical nature of the above definition. While we in principle could remove the training point from the integral evaluation, we know from calculus / probability theory / measure theory that they anyway are a set of measure zero and therefore do not contribute to the multiple integral.

Again, this is only true in the continuous setting from statistical learning theory in which the definition of the generalization error is given. In the next section, we will introduce concrete algorithms that approximate the generalization error using true discrete (test/validation) data. In these algorithms, we will exclude training samples from the evaluation of the generalization error. △

The just introduced generalization error sticks to the use of a single training set. However, for a real assessment of the quality of a given model, one should come up with a training set independent error measure. We obtain such an error measure by further averaging over all possible training sets in

**Definition 6.3** With the same requirements as in Definition 6.2, we define the **expected generalization / test error** by

$$\begin{aligned} EGE(f) &= \mathrm{E}_T\left(\mathrm{E}_{\boldsymbol{X},\boldsymbol{Y}|T}\left(L(\boldsymbol{Y}, f_T(\boldsymbol{X}))|T\right)\right) \\ &= \int_{R_T}\left(\int_{\mathbb{R}^K}\int_{\mathbb{R}^D} L(\boldsymbol{y}, f_{\mathcal{T}}(\boldsymbol{x}))\rho(\boldsymbol{x}, \boldsymbol{y}|\mathcal{T})d\boldsymbol{x}d\boldsymbol{y}\right)\rho(\mathcal{T})d\mathcal{T}\,. \end{aligned}$$

This is the quantity that we want to measure in order to understand the generalization properties of machine learning models.

## 6.3 Empirical error estimation

Certainly, solving the integrals in the definition of the expected generalization error exactly will never be possible if we do not know the joint density $\rho(\boldsymbol{x}, \boldsymbol{y})$. However, if were at least able to sample from that joint density, we could use the Monte Carlo method to approximate these integrals.

**Knowledge 6.1** (Monte Carlo method) According to the strong law of large numbers from probability, we have:

Let $X_1, \ldots, X_n$ be independent, equally distributed RVs with $\mathrm{E}(|X_i|) = \mu < \infty$ and $\mathrm{Var}(X_i) = \sigma^2 < \infty$. We define

$$\bar{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i \,.$$

It holds

$$P\left( \lim_{n \to \infty} \bar{X}_n = \mu \right) = 1 \,.$$

This gives rise to the following approximation: Let us assume that we are given a random varialble $X$ with density $\rho(x)$. We can approximate the mean over some function of $X$, $\mathrm{E}(f(X))$, by

$$\mathrm{E}(f(X)) = \int_{\mathbb{R}} f(x)\rho(x)dx \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \,,$$

where the $\{x_i\}_{i=1}^N$ are $N$ samples from $X$. The given approximation method for the integral is often called **Monte Carlo method**.

Let us formulate the approximation of $EGE(f)$ by the Monte Carlo method as

**Remark** (Approximation of the expected generalization error) To apply the Monte Carlo method, we would sample independently $N_T$ many training sets $\{\mathcal{T}_i\}_{i=1}^{N_T}$ and again independently $N_{XY}$ tuples $\{(\boldsymbol{x}_j, \boldsymbol{y}_j)\}_{j=1}^{N_{XY}}$ and would, with a bit of work, obtain the approximation

$$EGE(f) \approx \frac{1}{N_T} \frac{1}{N_{XY}} \sum_{i=1}^{N_T} \sum_{j=1}^{N_{XY}} L(\boldsymbol{y}_j, f_{\mathcal{T}_i}(\boldsymbol{x}_j))$$

Roughly speaking, a good estimate for the expected generalization error is thus given by independently choosing training sets for the predictor construction and independently choosing "correct" input-to-output pairs $(\boldsymbol{x}_j, \boldsymbol{y}_j)$, i.e. a *validation set*, to measure the error in the the constructed predictor, combined with a final averaging over all cases. $\triangle$

As mentioned before, in practice, we cannot sample from the joint distribution over $\boldsymbol{X}, \boldsymbol{Y}$. The only data that we have access to is (training) data that we are given as the input to the supervised machine learning task. All following standard approaches for evaluating the (expected) generalization error use this data to build approximations, which somewhat are simplified versions of the Monte Carlo approximation seen above.

Our first practical method is the *validation set approach*.

**Method 7** (Validation set approach) With the observations made so far, we start from given data $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ and randomly split this data set into a training set $\mathcal{T}_{train}$ of size $N_{train}$ and a validation set $\mathcal{T}_{val}$ of size $N_{val}$, thus

$$\mathcal{T} = \mathcal{T}_{train} \cup \mathcal{T}_{val}, \quad \mathcal{T}_{train} \cap \mathcal{T}_{val} = \emptyset \,.$$

Then the *validation set approach* trains a given model on the training data and evaluates the empirical error estimator for the generalization error as

$$\frac{1}{N_{val}} \sum_{i=1}^{N_{val}} L(\boldsymbol{y}_i, f_{\mathcal{T}_{train}}(\boldsymbol{x}_i)) \,.$$

△

Comparing the resulting estimator for the expected generalization error with the Monte Carlo approximation, we observe that it matches the case of using exactly one training set, i.e. $N_T = 1$ and a validation set of size $N_{XY} = N_{val}$ for a concretely given data set. We proceed to the resulting algorithm.

**Algorithm 8** (Error estimation by validation set approach)
**input:** $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ data, $f$ predictor model, $L$ loss
**output:** $\mathcal{E}$ approximation to expected generalization error

1. $\mathcal{T} =_{rand} \mathcal{T}_{train} \dot\cup \mathcal{T}_{val}$ *(randomly split into training and validation set)*
2. $f_{\mathcal{T}_{train}} \leftarrow \texttt{train}(f, \mathcal{T}_{train})$ *(train model $f$ using $\mathcal{T}_{train}$)*
3. $\mathcal{E} \leftarrow \frac{1}{|\mathcal{T}_{val}|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}_{val}} L(\boldsymbol{y}, f_{\mathcal{T}_{train}}(\boldsymbol{x}))$ *(estimate error)*
4. return $\mathcal{E}$

If $t(N)$ is the number of operations/time required to carry out the training and assuming that the training dominates the evaluation time for the predictor, the complexity of the validation set approach is $O(t(N))$.

It should be rather obvious that the validation set approach has its weakness in the number of training sets, namely *one*, that are used to approximated the expected generalization error. Translated to practical observations, this results in an error approximation that strongly depends on the split. Another practical disadvantage is that for small amounts of given data, the validation set approach even further reduces the amount of data used for the training and thereby might further lead to an over-estimation of the error.

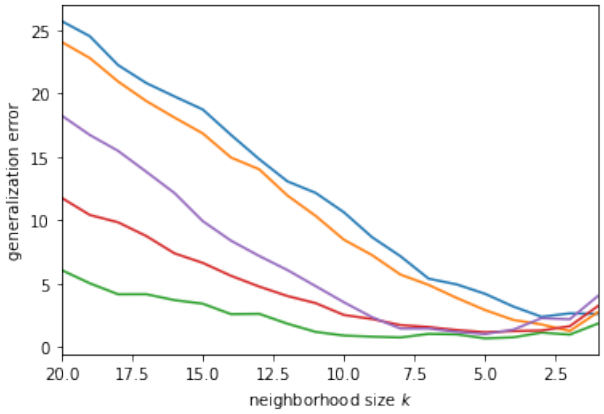
The following example highlights the issue of the dependency of the error on the (random) split.

**Example 6.2** (Validation set approach) We artificially generate training data by first choosing $N = 51$ input samples $\{x_i\}_{i=1}^N$ that are equally distributed over the interval $[-4, 4]$. Then we obtain outputs using

$$y_i = x_i^2 + \varepsilon_i \,,$$

where each $\varepsilon_i$ is drawn from a normal distribution with zero mean and $\sigma^2$ variance with $\sigma = 1.15$. We apply the validation set approach to measure the generalization

error of kNN regression for changing neighborhood size $k$ on this data set. To this end, we repeatedly randomly split the data into 40 training samples and 11 validation samples and compute the generalization error accordingly. In the below plot, each curve corresponds to a different split into training and validation set.




As discussed before, it becomes apparent that the calculated prediction error strongly depends on the split between training and validation set.

This example including the shown figure can be reproduced and modified here: launch binder [3)] △

As we have seen, the validation set approach uses exactly one training set and some validation samples, leading to the discussed the disadvantages of strong dependency on the split and over-estimation of the error due to smaller training set size. The *leave-one-out cross validation* overcomes these issues.

**Method 8** (Leave-one-out cross validation) In *leave-one-out cross validation* we consider the original data set $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$. This data set is then subsequently divided $N$ times into a training set of $N-1$ samples and a validation set of one, i.e. the remaining, sample. Each of these $N$ training and validation set pairs gives rise to a single error evaluation, which is averaged over all $N$ evaluations. △

As we only remove one training sample from the data set, there will be almost no error over-estimation. Moreover, the averaging process solves the issue of the strong fluctuations for different splits. Therefore, this approach outperforms the validation set approach.

Comparing this leave-one-out cross validation further to the Monte Carlo method approximation, we observe that it is somewhat the other extreme to the validation set approach. In the leave-one-out cross validation, we use $N_T = N$ (compared to $N_T = 1$) different training sets, however we only use a validation set of size $N_{XY} = 1$ (compared to $N_{XY} = N_{val}$).

[3)] The Jupyer notebook will only be available after the homework assignments on this topic were submitted.

We give the exact algorithm for leave-one-out cross validation below.

**Algorithm 9** (Error estimation by leave-one-out cross validation)
**input:** $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ data, $f$ predictor model, $L$ loss
**output:** $\mathcal{E}$ approximated expected generalization error

1. *for* $i \in \{1, \dots, N\}$
   a) $\mathcal{T}_{train} \leftarrow \mathcal{T} \setminus \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$
   b) $f_{\mathcal{T}_{train}} \leftarrow \texttt{train}(f, \mathcal{T}_{train})$ *(train model $f$ using $\mathcal{T}$ without $(\boldsymbol{x}_i, \boldsymbol{y}_i)$)*
   c) $\mathcal{E}_i \leftarrow L(\boldsymbol{y}_i, f_{\mathcal{T}_{train}}(\boldsymbol{x}_i))$ *(single-sample error)*
2. $\mathcal{E} \leftarrow \frac{1}{N} \sum_{i=1}^N \mathcal{E}_i$
3. return $\mathcal{E}$

The complexity of the algorithm is $O(N \cdot t(N))$. However, this complexity is also its biggest disadvantage as we indeed would have to train a given model $N$ times, which is often way too expensive.
A good compromise between the two just introduced approaches is given by

**Method 9** ($\mathcal{K}$-fold cross validation) In *$\mathcal{K}$-fold cross validation*, we split the data $\mathcal{T}$ into $\mathcal{K}$ (almost) identically sized subsets $\mathcal{T}_1, \dots, \mathcal{T}_\mathcal{K}$. Then, we train the model on the union over $\mathcal{K} - 1$ subsets and compute the error on the remaining subset. While keeping the same splitting, we repeat the same errror evaluation idea over all possible combinations of subsets of size $\mathcal{K} - 1$. Finally, we average over all $\mathcal{K}$ computed errors. △

We immediately see that $\mathcal{K}$-fold cross validation becomes leave-one-out cross validation for $\mathcal{K} = N$. Moreover, for $\mathcal{K} = 2$, it becomes very similar to the validation set approach.
All remaining details of the approach are summarized in

**Algorithm 10** (Error estimation by $\mathcal{K}$-fold cross validation)
**input:** $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$ data, $f$ predictor model, $L$ loss, $\mathcal{K}$ subset count
**output:** $\mathcal{E}$ approximated expected generalization error

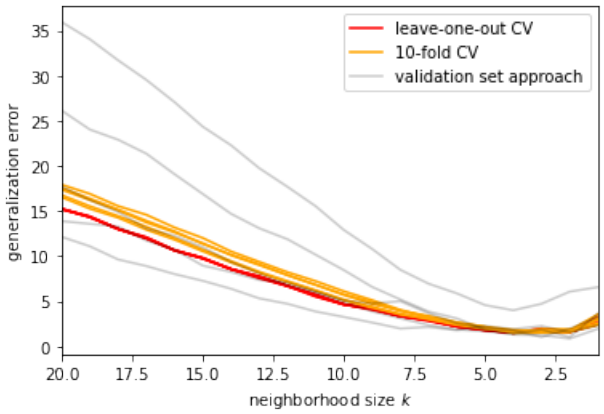
1. split $\mathcal{T}$ into equally sized sets $\mathcal{T}_1, \dots, \mathcal{T}_\mathcal{K}$
2. *for* $i \in \{1, \dots, \mathcal{K}\}$
   a) $\mathcal{T}_{train} \leftarrow \mathcal{T} \setminus \mathcal{T}_i$
   b) $f_{\mathcal{T}_{train}} \leftarrow \texttt{train}(f, \mathcal{T}_{train})$
   c) $\mathcal{E}_i \leftarrow \frac{1}{|\mathcal{T}_i|} \sum_{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{T}_i} L\left(\boldsymbol{y}, f_{\mathcal{T}_{train}}(\boldsymbol{x})\right)$
3. $\mathcal{E} \leftarrow \frac{1}{\mathcal{K}} \sum_{i=1}^N \mathcal{E}_i$
4. return $\mathcal{E}$

Its computational complexity is $O(\mathcal{K} \cdot t(N(\mathcal{K}-1)/\mathcal{K}))$, assuming that the training part dominates the evaluation part. In some sense, the choice of $\mathcal{K}$ allows us to have a "slider" between the very expensive leave-one-out cross validation and the cheap but high-variance validation set approach.

Putting the new method again in perspective to the Monte Carlo approximation of the expected generalization error, this method, roughly speaking, uses $N_T = \mathcal{K}$ training sets of size $N - \frac{N}{\mathcal{K}}$ and $N_{XY} = \frac{N}{\mathcal{K}}$ validation samples.

Let us compare the different generalization error estimation approaches in

**Example 6.3** (Method comparison) We use the artificial data from Example 6.2. However, this time, we five times evaluate each of the three different error estimators from this section, namely the validation set approach with $N_{train} = 40$ training samples, leave-one-out cross validation and $\mathcal{K}$-fold cross validation with $\mathcal{K} = 10$. The results are given below.



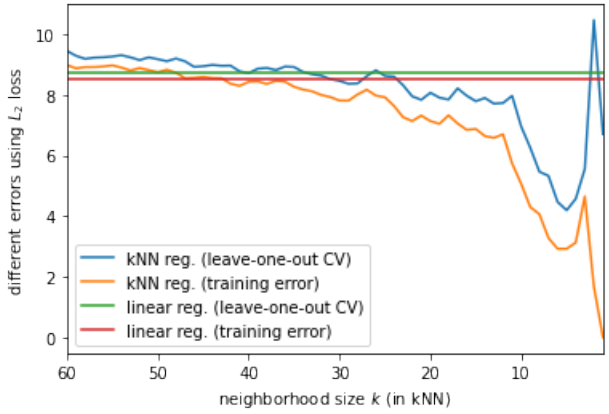

As assumed, the validation set approach has the strongest variance in the error predictions, while the leave-one-out cross validation has the smallest variance. $\mathcal{K}$-fold cross validation is a good compromize in between.

This example including the shown figure can be reproduced and modified here: launch binder [4)] △

We finally would like to continue our discussion on the training error, model complexity and overfitting. To this end, we have a look at

**Example 6.4** As in the second part of Example 6.1, we again use the Energy Efficiency Data Set and compute the training error for linear regression and kNN regression for a (now reduced) range of $k \in [1, 40]$. In addition, we compute the generalization error for both methods by leave-one-out cross validation. The results of this study are given below.

[4)] The Jupyer notebook will only be available after the homework assignments on this topic were submitted.




In the given example, the training errors are consistently smaller than the generalization errors. This is a very typical behavior, as we would usually expect that predictors perform worse outside of the training set than exactly on the training set, for which they have been optimized.

Moreover, we can indeed confirm the expected *overfitting* for growing model complexity. Hence, we see a decreasing training error, while the prediction error becomes quite large. At the same time, linear regression seems to perform worse then kNN regression on the given data set. △

In supervised machine learning tasks, we often have models with some additional (hyper-)parameters that influence the outcome of the model. So far, we have seen that for kNN regression with the neighborhood size $k$. In these cases, cross validation approaches are, besides of their use for comparing models and giving a generalization error estimation, also often used to search for optimal parameters. It is e.g. possible to sample a series of possible parameter choices, compute the error by cross validation and then pick the parameter set with the lowest generalization error.

# 7. Bias – variance tradeoff

Already in the previous chapter, we had a careful look at the somewhat complex interdependencies of training error, generalization error and model complexity. In this chapter, we would like to put this discussion on solid theoretical grounds. To this end, we will introduce the *expected (squared) generalization error at location* $\boldsymbol{x}_0$ and will evaluate it for a generic noisy data model. The outcome of this analysis is that the (expected) generalization error decomposes as

$$\text{Error} = \text{Irreducible Error} + \text{Bias}^2 + \text{Variance}\,.$$

Bias and variance are important properties of a given model, which finally connect back to the observed behavior of training error vs. generalization error for models of changing model complexity.

In the following, we will start by introducing the necessary notions. Then, we give the fundamental theorem on the *bias-variance decomposition*. This theorem is afterwards applied to the predictor constructed in kNN regression. In the analysis of the bias-variance decomposition of kNN regression, we will make important structural observations that will lead to the introduction of the very important notion *bias-variance tradeoff* and a sharpening of the notion of *overfitting*.

## 7.1 Bias-variance decomposition

Let us briefly recall that we introduced in the previous chapter the expected generalization error as

$$\begin{aligned}
EGE(f) &= \mathrm{E}_T\left(\mathrm{E}_{\boldsymbol{X},\boldsymbol{Y}|T}\left(L(\boldsymbol{Y}, f_T(\boldsymbol{X}))|T\right)\right) \\
&= \int_{R_T}\left(\int_{\mathbb{R}^K}\int_{\mathbb{R}^D} L(\boldsymbol{y}, f_{\mathcal{T}}(\boldsymbol{x}))\rho(\boldsymbol{x},\boldsymbol{y}|\mathcal{T})d\boldsymbol{x}d\boldsymbol{y}\right)\rho(\mathcal{T})d\mathcal{T} \\
&= \int_{R_T}\left(\int_{\mathbb{R}^K}\int_{\mathbb{R}^D} L(\boldsymbol{y}, f_{\mathcal{T}}(\boldsymbol{x}))\frac{\rho(\boldsymbol{x},\boldsymbol{y},\mathcal{T})}{\rho_T(\mathcal{T})}d\boldsymbol{x}d\boldsymbol{y}\right)\rho(\mathcal{T})d\mathcal{T} \\
&= \int_{R_T}\int_{\mathbb{R}^K}\int_{\mathbb{R}^D} L(\boldsymbol{y}, f_{\mathcal{T}}(\boldsymbol{x}))\rho(\boldsymbol{x},\boldsymbol{y},\mathcal{T})d\boldsymbol{x}d\boldsymbol{y}d\mathcal{T} \\
&= \mathrm{E}_{\boldsymbol{X},\boldsymbol{Y},T}\left(L(\boldsymbol{Y}, f_T(\boldsymbol{X}))\right)
\end{aligned}$$

The last three steps are new, here. They show that the expected generalization error is nothing else than the expectation of the loss between predictor and output over the training data, inputs and outputs.

We keep this observation in mind and thereby now define an expected (squared) generalization error for a specific location. For simplicity, we stick in this full chapter to the case of $K = 1$, i.e. a one-dimensional output.

**Definition 7.1** Let $\boldsymbol{X}$, $Y$ be quantitative input and output ($K = 1$), $\rho(\boldsymbol{x}, y)$ be their joint density and $L_2$ be the squared loss. Moreover, we are given a training set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ that is a sample of the random variable $T$. Finally $f_{\mathcal{T}}$ is a function to predict $Y$ from $\boldsymbol{X}$ that was trained using $\mathcal{T}$. For the squared loss, the **expected (squared) generalization error of $f$ at location $\boldsymbol{x}_0$** is defined as

$$\begin{aligned} EGE(f, \boldsymbol{x}_0) &= \mathrm{E}_{Y,T|X}\left(L_2(Y, f_T(\boldsymbol{X}))|\boldsymbol{X} = \boldsymbol{x}_0\right) \\ &= \int_{R_T}\int_{\mathbb{R}^D} L(y, f_{\mathcal{T}}(\boldsymbol{x}_0))\rho(y, \mathcal{T}|\boldsymbol{x}_0)dyd\mathcal{T} \end{aligned}$$

Hence this expected generalization error computes the mean over all outputs and possible training data, while being conditioned to a fixed input. In other words, we get the (expected) generalization error for a fixed location. Note that $\boldsymbol{x}_0$ is chosen independently of the training data. We need this knowledge at a later stage.

Now, we state our main result of this section.

**Theorem 7.1** (Bias–variance decopmposition) With the requirements of Definition 7.1 and the assumption that the output is given by

$$Y = f_{exact}(\boldsymbol{X}) + \varepsilon, \quad \text{with } \mathrm{E}(\varepsilon) = 0, \mathrm{Var}(\varepsilon) = \sigma_\varepsilon^2,$$

where $f_{exact} : \mathbb{R}^D \to \mathbb{R}$ is known, we can decompose $EGE(f, \boldsymbol{x}_0)$ as

$$EGE(f, \boldsymbol{x}_0) = \sigma_\varepsilon^2 + \left[\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - f_{exact}(\boldsymbol{x}_0)\right]^2 + \mathrm{E}_T\left(\left(f_T(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right).$$

The first term is an error that is intrinsic to the data. Moreover,

- $\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - f_{exact}(\boldsymbol{x}_0)$ is the **bias** of the predictor, and
- $\mathrm{E}_T\left(\left(f_T(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right)$ is the **variance** of the predictor.

Let us analyze the statement of the theorem by

**Remark** (Interpretation of the bias-variance descomposition theorem) In the above theorem, we start with the assumption that we, in principle, would know the exact de-

pendency between our input $\boldsymbol{X}$ and our output $Y$. This exact dependency is given by the function $f_{exact}$. However, we are interested, how the constructed predictor reacts to perturbations in that model. Therefore, we replace the exact dependency

$$Y = f_{exact}(\boldsymbol{X})$$

by a perturbed one, in which we simply use an additive noise term $\varepsilon$ with zero expectation and $\sigma_\varepsilon^2$ variance,

$$Y = f_{exact}(\boldsymbol{X}) + \varepsilon\,.$$

With a little work, which we will do in a moment, we can then derive that the expected prediction error at a location $\boldsymbol{x}_0$ of some predictor $f$ for the modified $Y$ decomposes into the additive components of an irreducible error, the squared *bias* and the *variance* of the predictor.

The term $\sigma_\varepsilon^2$ is considered an *irreducible* error, as we cannot assume to eliminate the intrinsic noise in the data. The term $\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - f_{exact}(\boldsymbol{x}_0)$ is the *bias* of the predictor. The bias describes the deviation of the mean (with respect to the training data) of the predictor from the exact model. In some sense, one can interpret this as the error that the predictor would have without additional noise. Finally, the term $\mathrm{E}_T\left(\left(f_T(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right)$ is the *variance*, i.e. the expected squared deviation around the mean (all with respect to the training data), of the predictor. In other words this term measures, how much the originally induced noise (that goes into the training data) will lead to fluctuations in the predictor. △

The bias-variance decomposition is one of the "big" statements in machine learning. It characterizes the errors that we will experience in the predictor. However, at this stage, the statement still seems to be rather abstract. In the next section, we will apply the decomposition to a concrete predictor. The curious reader might therefore immediately skip forward to that section and come back to this point, later.

For all others, we now discuss the proof of the bias-variance decomposition theorem.

Proof

The proof is mostly a sequence of applications of definitions and small calculations. Let us just start from the localized expected (squared) prediction error as

$$\begin{aligned}
&EGE(f, \boldsymbol{x}_0) = \mathrm{E}_{Y,T|\boldsymbol{X}}\left((Y - f_T(\boldsymbol{X}))^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&= \mathrm{E}_{Y,T|\boldsymbol{X}}\left((Y + \mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - f_T(\boldsymbol{X}))^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&= \mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(Y - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 + 2\left[Y - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - f_T(\boldsymbol{X})\right]\right) \\
&\qquad + \left(\mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - f_T(\boldsymbol{X})\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\Big) \\
&= \mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(Y - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&\quad + 2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left[Y - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - f_T(\boldsymbol{X})\right] | \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&\quad + \mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(f_T(\boldsymbol{X}) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&= \mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(Y - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) && (7.1) \\
&\quad + 2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left[Y - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - f_T(\boldsymbol{X})\right] | \boldsymbol{X} = \boldsymbol{x}_0\right) && (7.2) \\
&\quad + \mathrm{E}_T\left(\left(f_T(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right) && (7.3)
\end{aligned}$$

In the first line, we just apply the definition. Afterwards, we "add a zero", i.e. we add $+\mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)$. Moving to the third line, we multiply out the square. Next, we use the linearity of the expectation. Finally, we observe that

$$\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(f_T(\boldsymbol{X}) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) = \mathrm{E}_T\left(\left(f_T(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right)$$

holds, where we apply the conditioning and observe that the only random part that we need to cover by the expectation is the training data $T$.

Next, we analyze the term in (7.1) by

$$\begin{aligned}
(7.1) &= \mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(Y - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&= \mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(\varepsilon + f_{exact}(\boldsymbol{X}) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&= \mathrm{E}_{Y,T|X}\left(\varepsilon^2 | \boldsymbol{X} = \boldsymbol{x}_0\right) + 2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\varepsilon\left(f_{exact}(\boldsymbol{X}) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right) | \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&\qquad + \mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left(f_{exact}(\boldsymbol{X}) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&= \mathrm{E}_\varepsilon\left(\varepsilon^2\right) + 2\,\mathrm{E}_\varepsilon\left(\varepsilon\left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\right) + \mathrm{E}_{Y,T}\left(\left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right) \\
&= \sigma_\varepsilon^2 + 2\,\mathrm{E}_\varepsilon\left(\varepsilon\right)\left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right) + \left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2 \\
&= \sigma_\varepsilon^2 \qquad\qquad + \left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2 .
\end{aligned}$$

Indeed, some of the steps are not fully obvious. Therefore, we slowly discuss them. In the first transformation, we use the definition of the output, i.e. $Y = f_{exact}(\boldsymbol{X}) + \varepsilon$. In the second step, we multiply out the term by grouping together the two terms $f_{exact}(\boldsymbol{X}) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)$. At the same time, we apply the linearity of the expectation. The next transformation is a bit tricky. We first consider the term $\mathrm{E}_{Y,T|X}\left(\varepsilon^2 | \boldsymbol{X} = \boldsymbol{x}_0\right)$ for which we claim

$$\mathrm{E}_{Y,T|X}\left(\varepsilon^2 | \boldsymbol{X} = \boldsymbol{x}_0\right) = \mathrm{E}_\varepsilon\left(\varepsilon^2\right) .$$

If we carefully study the term $\varepsilon^2$, we observe that its "randomness" is independent of the choice of the input and the training data. In principle, it only implicitly depends on the random variable $Y$. However, in fact, $Y$ is even derived from the random variable $\varepsilon$, therefore, it is enough to consider the expectation in that random variable, which gives the claimed equality.

Similarly, we claim that

$$2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\varepsilon\left(f_{exact}(\boldsymbol{X})-\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right)|\boldsymbol{X}=\boldsymbol{x}_0\right)=2\,\mathrm{E}_\varepsilon\left(\varepsilon\left(f_{exact}(\boldsymbol{x}_0)-\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\right)\,.$$

Here, we have to additionally study the term $f_{exact}(\boldsymbol{X})-\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)$. By conditioning it to $\boldsymbol{X}=\boldsymbol{x}_0$ we simply get the term evaluated at $\boldsymbol{x}_0$. At the same time, we observe that $f_{exact}$ is independent of the training data $T$ and of the output $Y$ as it has been given as a fixed, deterministic function. Similarly, $\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)$ is independent of the training data $T$ as the dependency has already been removed by the mean over $T$. Moreover, this term is anyway not dependent on $Y$. Hence it only remains $\varepsilon$ as random variable and we get the above result.

In the next step of the derivation, with the fourth equality, we use in the first term a zero summation argument, that we have seen before, namely knowing $\mathrm{E}_\varepsilon(\varepsilon)=0$ we get

$$\mathrm{E}_\varepsilon\left(\varepsilon^2\right)=\mathrm{E}_\varepsilon\left((\varepsilon-\mathrm{E}_\varepsilon(\varepsilon))^2\right)=\mathrm{Var}_\varepsilon(\varepsilon)=\sigma_\varepsilon^2\,.$$

We get the second term in that line via

$$2\,\mathrm{E}_\varepsilon\left(\varepsilon\left(f_{exact}(\boldsymbol{x}_0)-\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\right)=2\,\mathrm{E}_\varepsilon(\varepsilon)\left(f_{exact}(\boldsymbol{x}_0)-\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\,,$$

hence we essentially move the difference term out of the expectation. We can do that, since, as discussed before, both terms in that difference are even independent of $\varepsilon$ and thereby no random variables but just constants. Again via linearity of the expectation, we can move constants outside of the expectation. Finally, we use

$$\mathrm{E}_{Y,T}\left(\left(f_{exact}(\boldsymbol{x}_0)-\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right)=\left(f_{exact}(\boldsymbol{x}_0)-\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2$$

to get the last term in that line. Here, the same argument of both terms in the difference being constants, i.e. no random variables, holds. Thereby, we can simply drop the expectation.

The final transformation is obtained, since the second term in the sum becomes zero as $\mathrm{E}_\varepsilon(\varepsilon)=0$.

As a next step, we analyze equation (7.2). We thus compute

$$\begin{aligned}
(7.2)=&2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left[Y-\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)-f_T(\boldsymbol{X})\right]|\boldsymbol{X}=\boldsymbol{x}_0\right)\\
=&2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left[\varepsilon+f_{exact}(\boldsymbol{X})-\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)-f_T(\boldsymbol{X})\right]|\boldsymbol{X}=\boldsymbol{x}_0\right)\\
=&2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\varepsilon\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)-f_T(\boldsymbol{X})\right]\right.\\
&\qquad\left.+\left[f_{exact}(\boldsymbol{X})-\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)-f_T(\boldsymbol{X})\right]|\boldsymbol{X}=\boldsymbol{x}_0\right)\\
=&2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\varepsilon\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)-f_T(\boldsymbol{X})\right]|\boldsymbol{X}=\boldsymbol{x}_0\right) &(7.4)\\
&+2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left[f_{exact}(\boldsymbol{X})-\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)-f_T(\boldsymbol{X})\right]|\boldsymbol{X}=\boldsymbol{x}_0\right) &(7.5)\\
& &(7.6)
\end{aligned}$$

The first transformation is the application of the definition of $Y$, the second transformation is achieved by multiplying out the terms and the third transformation goes back to the linearity of the expectation.

We now develop the terms from equations (7.4) and (7.5) independently, starting with (7.4):

$$\begin{aligned}
(7.4) =& 2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\varepsilon\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - f_T(\boldsymbol{X})\right]|\boldsymbol{X}=\boldsymbol{x}_0\right)\\
=& 2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\varepsilon\,\mathrm{E}_T\left(f_T(\boldsymbol{X})\right)|\boldsymbol{X}=\boldsymbol{x}_0\right) - 2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\varepsilon f_T(\boldsymbol{X})|\boldsymbol{X}=\boldsymbol{x}_0\right)\\
=& 2\,\mathrm{E}_{\varepsilon}\left(\varepsilon\,\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right) - 2\,\mathrm{E}_{\varepsilon,T}\left(\varepsilon f_T(\boldsymbol{x}_0)\right)\\
=& 2\,\mathrm{E}_{\varepsilon}\left(\varepsilon\right)\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - 2\,\mathrm{E}_{\varepsilon,T}\left(\varepsilon\right)\mathrm{E}_{\varepsilon,T}\left(f_T(\boldsymbol{x}_0)\right)\\
=& 0\cdot\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - 0\cdot\mathrm{E}_{\varepsilon,T}\left(f_T(\boldsymbol{x}_0)\right)\\
=& 0
\end{aligned}$$

The first tranformation multiplies out the inner term in the expectation and applies linearity. For the first term in the second line, we then apply the conditioning and observe that the only remaining random component for the outer expectation is $\varepsilon$, leading to the given transformation. For the second term, we also apply the conditioning and identify that $\varepsilon$ and the predictor have still a dependency on $\varepsilon$ and $T$ as random variables. The transformation from the fourth to the fifth line uses first

$$2\,\mathrm{E}_{\varepsilon}\left(\varepsilon\,\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right) = 2\,\mathrm{E}_{\varepsilon}\left(\varepsilon\right)\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\,,$$

where we have simply observed that the term $\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)$ is not a random variable and can thereby using linearity moved outside of the expectation. We then use

$$2\,\mathrm{E}_{\varepsilon,T}\left(\varepsilon f_T(\boldsymbol{x}_0)\right) = 2\,\mathrm{E}_{\varepsilon,T}\left(\varepsilon\right)\mathrm{E}_{\varepsilon,T}\left(f_T(\boldsymbol{x}_0)\right)\,,$$

which looks very similar, however uses a different argument. Here, we use that $\varepsilon$ and $f_T$ considered as random variables are independent, since we have modeled the situation such that the sampling of the training data is independent of the sampling of $Y$ and thereby the sampling of $\varepsilon$. As this is the case, we can apply that the expectation of the product of two independent random variables is the product of the expectations of the two random variables and get the above result.

Let us next consider the term from (7.5) by

$$\begin{aligned}
(7.5) =& 2\,\mathrm{E}_{Y,T|\boldsymbol{X}}\left(\left[f_{exact}(\boldsymbol{X}) - \mathrm{E}_T\left(f_T(\boldsymbol{X})\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{X})\right) - f_T(\boldsymbol{X})\right]|\boldsymbol{X}=\boldsymbol{x}_0\right)\\
=& 2\,\mathrm{E}_{Y,T}\left(\left[f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right]\left[\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - f_T(\boldsymbol{x}_0)\right]\right)\\
=& 2\left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\mathrm{E}_{Y,T}\left(\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - f_T(\boldsymbol{x}_0)\right)\\
=& 2\left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\mathrm{E}_T\left(\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - f_T(\boldsymbol{x}_0)\right)\\
=& 2\left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\left(\mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\\
=& 2\left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)\cdot 0\\
=& 0
\end{aligned}$$

Above, we start by applying the conditioning. Then, we identify that $[f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)]$ is not a random variable and move it out of the expectation. Further, we observe that the second term of the resulting big product in the third line is independent of $Y$, hence we can skip the averaging over that random variable. Afterwards, in the transformation from line four to five, we apply the expectation in the second term of the product by linearity to both sub-terms. These, however, then cancel out and we are done.

Finally, collecting all partial results that we have derived so far, we obtain

$$\begin{aligned}
EGE(f, \boldsymbol{x}_0) &= \mathrm{E}_{Y,T|\boldsymbol{X}}\left((Y - f_T(\boldsymbol{X}))^2 \middle| \boldsymbol{X} = \boldsymbol{x}_0\right) \\
&= (7.1) + (7.2) + (7.3) \\
&= \sigma_\varepsilon^2 + \left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2 + (7.4) + (7.5) + (7.3) \\
&= \sigma_\varepsilon^2 + \left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2 + 0 + \mathrm{E}_T\left(\left(f_T(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right) \\
&= \sigma_\varepsilon^2 + \left(f_{exact}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2 + \mathrm{E}_T\left(\left(f_T(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_T(\boldsymbol{x}_0)\right)\right)^2\right) ,
\end{aligned}$$

which is the statement of the theorem. □

## 7.2 Bias-variance decomposition for kNN regression

In this section, we would like to have a more practical look at the bias-variance decomposition discussed before. We do this, by applying the theorem to kNN regression. This will give us a rather technical, but precise statement on what is the bias and the variance of the kNN regression predictor. We will then use this mathematical result in an example, where we analyze the special cases of kNN regression for $k = 1$ and $k = N$, which correspond to maximal and minimal model complexity. These special cases will finally show us, what is the very important *bias-variance trade-off* that we very often have to consider in supervised learning. This central notion is the main contribution of this section.

We start by formulating the statement on the bias-variance decomposition for kNN regression predictors.

**Theorem 7.2** (Bias–variance decomposition for kNN regression) Let the setting of Theorem 7.1 be given and let $T = \{(\mathscr{X}_i, \mathscr{Y}_i)\}_{i=1}^N$ be the i.i.d. random variable describing a training set with $\mathscr{Y}_i = f_{exact}(\mathscr{X}_i) + \mathscr{E}_i$, i.e. $T$ itself is build from the random variables $\mathscr{X}_i, \mathscr{E}_i$ for $1 \leq i \leq N$. If $f_{kNN}$ is the kNN regression predictor, then its bias and variance are given by

$$\mathrm{E}_T\left(\frac{1}{k}\sum_{\mathscr{X}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right) - f_{exact}(\boldsymbol{x}_0) \quad \text{and} \quad \mathrm{Var}_T\left(\frac{1}{k}\sum_{\mathscr{X}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right) + \frac{\sigma_\varepsilon^2}{k} .$$

The bias–variance decomposition of $f_{kNN}$ is hence given by

$$EGE(f_{kNN}, \boldsymbol{x}_0)$$
$$= \sigma_\varepsilon^2 + \left[ \mathrm{E}_T \left( \frac{1}{k} \sum_{\mathscr{X}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i) \right) - f_{exact}(\boldsymbol{x}_0) \right]^2 + \mathrm{Var}_T \left( \frac{1}{k} \sum_{\mathscr{X}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i) \right) + \frac{\sigma_\varepsilon^2}{k} .$$

($N_k(\boldsymbol{x}_0)$ is the set of the $k$ nearest training inputs wrt. $\boldsymbol{x}_0$.)

The statement itself is not fully trivial to read, therefore, we should discuss first the different objects that are described by the theorem. The proof is postponed to the end of this section.

**Remark** (Discussion of the bias-variance decomposition of kNN regression) We should first recall the statement from the general Theorem 7.1 on bias-variance decomposition. There, we assume that there is an exactly known relationship between the input and the output, which is however perturbed by an additive noise. This noise then propagates into the localized expected generalization error of the resulting predictor, where the predictor always depends on the training data that was derived from the input and the output. Specifically, all expectations in the terms of the general bias-variance decomposition are expectations with respect to the random variable $T$, from which we draw randomly the training data.

As we now want to explicitly analyze a given method, namely kNN regression, we (unfortunately) have to further specify, what this random variable $T$ means. This is done at the beginning of Theorem 7.2, where we introduce it as

$$T = \{(\mathscr{X}_i, \mathscr{Y}_i)\}_{i=1}^N .$$

Hence, this $T$ itself is set of $N$ tuples of i.i.d. random variables $\mathscr{X}_i$ and i.i.d. random variables $\mathscr{Y}_i$.[1)] However, since we already know the assumed dependency between the $\mathscr{X}_i$ and $\mathscr{Y}_i$ via

$$\mathscr{Y}_i = f_{exact}(\mathscr{X}_i) + \mathscr{E}_i ,$$

we actually know that the "randomness" comes from the input sample random variables $\mathscr{X}_i$ and the noise random variables $\mathscr{E}_i$.

Now that we understand what the random variable $T$ *is*, we can continue and analyze the bias and variance term. The bias of kNN regression is

$$\mathrm{E}_T \left( \frac{1}{k} \sum_{\mathscr{X}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i) \right) - f_{exact}(\boldsymbol{x}_0) .$$

[1)] For some reader, it might feel like an additional burden to even introduce random variables that describe training sets. Nevertheless, a mathematically sound analysis of the bias-variance decomposition of kNN regression without this construction is impossible.

To better understand this term, we should recall that the kNN regression predictor for a neighborhood size of $k$ and a training set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ is

$$f_{kNN}(\boldsymbol{x}_0) = \frac{1}{k} \sum_{\boldsymbol{x}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} y_i \,,$$

hence for a given input $\boldsymbol{x}_0$ it is the average over the output of those $k$ training samples that are closest to the input $\boldsymbol{x}_0$. The bias above thus gives the deviation between the evaluation of the exactly known dependency between input and output and the training-sample-averaged kNN predictor that considers the exactly known function $f_{exact}$ (without noise!). Even though we construct the predictor from noisy data, its bias is thus *independent* of this noise.

Let us next discuss the variance of kNN regression, which is given by

$$\mathrm{Var}_T \left( \frac{1}{k} \sum_{\mathscr{X}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i) \right) + \frac{\sigma_\varepsilon^2}{k} \,.$$

Here, we have first a term that is the variance (with respect to random training data) of the kNN regression predictor on the exact function $f_{exact}$. The second term describes the contribution of the noise to the variance of the kNN regression predictor. Indeed, it is the variance of the noise, scaled by $\frac{1}{k}$. △

Given that some reader might simultaneously refer to other literature, we should make the following additional

**Remark** (Relation to other literature) In other literature, e.g. [6], the bias-variance decomposition of kNN regression is reported to be

$$EGE(f_{kNN}, \boldsymbol{x}_0) = \sigma_\varepsilon^2 + \left[ \left( \frac{1}{k} \sum_{\boldsymbol{x}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\boldsymbol{x}_i) \right) - f_{exact}(\boldsymbol{x}_0) \right]^2 + \frac{\sigma_\varepsilon^2}{k} \,.$$

This result can be derived by making the simplifying assumption that the input samples $\boldsymbol{x}_i$ do not come from a random sampling, but are deterministically given. This, however, is in conflict with our definition of the (localized) expected generalization error. Therefore, we stick to the result given in Theorem 7.2. △

As stated at the beginning of this section, the main contribution here is not the theorem on the bias-variance decomposition itself, but its two special cases $k = 1$ and $k = N$ which lead to the important *bias-variance trade-off*. We formulate these observations as

**Example 7.1** (Bias-variance trade-off) We would like to analyze the bias-variance decomposition for kNN regression for the two extremal cases of $k = 1$, i.e. only the nearest neighbor in the training set is taken as predictor, and $k = N$, i.e. all training points are

considered in the prediction. For the case of $k = 1$, the bias-variance decomposition for kNN regression becomes

$$\begin{aligned} &EGE(f_{kNN}, \boldsymbol{x}_0) \\ &= \sigma_\varepsilon^2 + \left[\mathrm{E}_T\left(f_{exact}(\mathscr{X}_{nearest})\right) - f_{exact}(\boldsymbol{x}_0)\right]^2 + \mathrm{Var}_T\left(f_{exact}(\mathscr{X}_{nearest})\right) + \sigma_\varepsilon^2, \end{aligned}$$

where $\mathscr{X}_{nearest}$ refers to that random variable $\mathscr{X}_i$ that is closest to the location $\boldsymbol{x}_0$. We can see, that the variance that is induced by the noise in the data becomes maximal (i.e. $\sigma_\varepsilon^2$ instead of $\frac{\sigma_\varepsilon^2}{k}$). At the same time, the bias becomes rather small, as always the closest training samples are considered for a given $\boldsymbol{x}_0$. This becomes particularly clear for the case in which we would always use a $\boldsymbol{x}_0$ that comes from the (random) training set. In that case, the bias even becomes zero. To summarize, we have low bias but high (noise-induced) variance.

In the other extremal case of $k = N$, we obtain

$$\begin{aligned} &EGE(f_{kNN}, \boldsymbol{x}_0) \\ &= \sigma_\varepsilon^2 + \left[\mathrm{E}_T\left(\frac{1}{N}\sum_{i=1}^{N} f_{exact}(\mathscr{X}_i)\right) - f_{exact}(\boldsymbol{x}_0)\right]^2 + \mathrm{Var}_T\left(\frac{1}{N}\sum_{i=1}^{N} f_{exact}(\mathscr{X}_i)\right) + \frac{\sigma_\varepsilon^2}{N}\,. \end{aligned}$$

It should be noted here that the sums indeed go over all training samples, since we take as "neighborhood" all samples from the training set. We immediately observe that the noise-induced variance $\frac{\sigma_\varepsilon^2}{N}$ is minimal. Even more, if we consider the case of large $N$ or even the limit case $N \to \infty$, the noise-induced variance vanishes. In that limit case, recalling Knowledge 6.1, we also have

$$\frac{1}{N}\sum_{i=1}^{N} f_{exact}(\mathscr{X}_i) \to \mathrm{E}_T\left(f_{exact}(\mathscr{X})\right)$$

and we further conclude, since this expectation in the limit is a constant that

$$\mathrm{Var}_T\left(\frac{1}{N}\sum_{i=1}^{N} f_{exact}(\mathscr{X}_i)\right) \to 0\,.$$

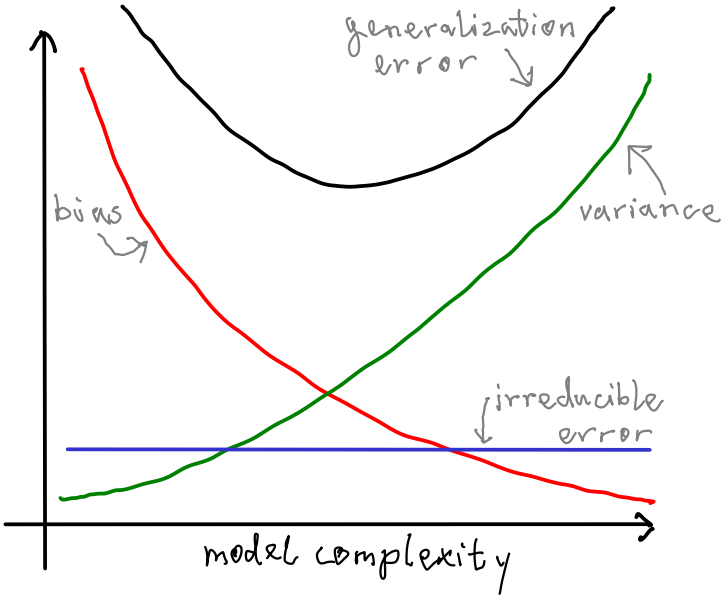
Hence in the limit case for $N \to \infty$, all variance goes to zero.

If we analyze the bias term, first for finite $N$, we see that $\frac{1}{N}\sum_{i=1}^{N} f_{exact}(\mathscr{X}_i)$ becomes independent of the choice of $\boldsymbol{x}_0$. Thereby, we notice that the bias becomes maximal for $k = N$, since the predictor will never follow the choice of $\boldsymbol{x}_0$. In the limit case of $N \to \infty$, this becomes even more pronounced as the bias then behaves like

$$\left[\mathrm{E}_T\left(\frac{1}{N}\sum_{i=1}^{N} f_{exact}(\mathscr{X}_i)\right) - f_{exact}(\boldsymbol{x}_0)\right]^2 \to \left[\mathrm{E}_T\left(f_{exact}(\mathscr{X})\right) - f_{exact}(\boldsymbol{x}_0)\right]^2\,.$$

Summarizing this second part of the analysis, we see that for $k = N$ we get small variance and large bias.

To conclude, we have the general observation that in kNN regression, we obtain for a high model complexity, i.e. $k$ towards 1, a small bias but a large variance, while we obtain for a small model complexity, i.e. $k$ towards $N$, a small variance but a large bias. Both errors, together with the irreducible error, contribute independently from each other to the overall generalization error of the predictor. Qualitatively, our analysis thus gives the below general picture.




The *bias-variance trade-off* in kNN regression corresponds to the observation that we will never be able to minimize both, bias *and* variance. Instead, we have to make a trade-off between both error contributions and might want to choose the model complexity such that the overall generalization error becomes small. △

While we do this kind of analysis only for kNN regression, it should be noted that the same observation holds more generally for models in supervised machine learning. Therefore, we might want to formulate it one more time, in something like a definition.

**Definition 7.2** (Bias-variance tradeoff) The **bias-variance trade-off** in supervised machine learning describes the observation that a model with minimal bias *and* minimal variance usually does not exist. However, we need to make a trade-off between both errors, which might depend on the data, the model and other influencing factors.

In the previous chapter, we have already introduced the notion of *overfitting*. We roughly analyzed it as the situation in which a model is too strongly "adapted" to given data, which then might lead to an increased generalization error. With the notions that we have developed now, we can make the statement more precise by

**Definition 7.3** (Overfitting) In supervised machine learning, we say that *overfitting* happens, if a given predictor (model) has a too low bias. Then the bias-variance trade-off implies that the variance of the model is rather large, leading to a suboptimal generalization error.

A notion, which describes the opposite situation, is given by

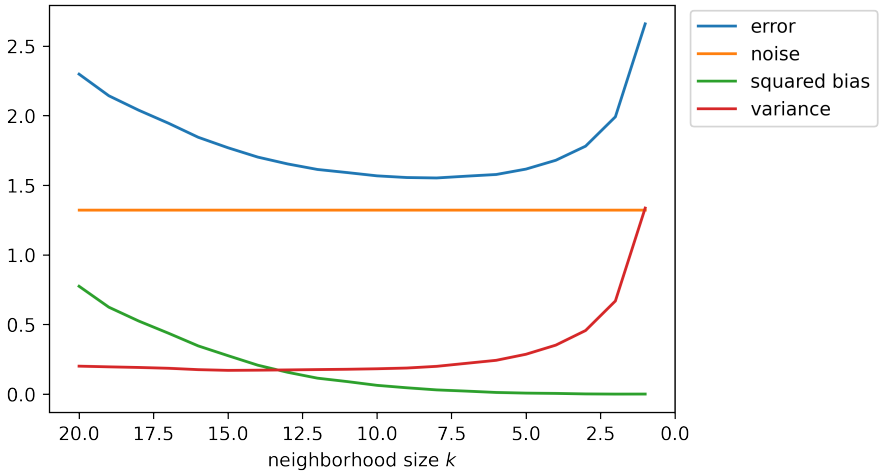
**Definition 7.4** (Underfitting) In supervised machine learning, we say that **underfitting** happens, if a given predictor (model) has a too high bias. Then the bias-variance trade-off implies that the variance of the model is rather small. Still, the high bias leads to a suboptimal generalization error.

Ideally, we thus neither would like to be in the overfitting nor in the underfitting regime, which is, again, the bias-variance trade-off.

After the qualitative, asymptotic Example 7.1 for the bias-variance trade-off in kNN regression, we would also like to give an empirical example for the statement of Theorem 7.2.

**Example 7.2** (Empirical analysis of bias-variance decomposition in kNN regression) In this example, we exactly reproduce the setup of the bias-variance decomposition for kNN regression in Theorem 7.2. To this end, we start by choosing $f_{exact}(X) = X^2$. The training data random variable $T$ is modeled such that we have $N = 51$ training samples and the input sample random variables $\{\mathscr{X}_1, \dots \mathscr{X}_N\}$ all follow a uniform distribution on $[-4, 4]$. Moreover, the noise samples all have a variance of $\sigma_\varepsilon^2 = 1.15^2$. The expectation and variance with respect to the training sets is approximated over $N_T = 100$ samples of training sets using the classical unbiased estimators for expectation and variance known from statistics.

Below, we give the irreducible error, squared bias, variance and the resulting total error for an evaluation point of $\boldsymbol{x}_0 = 0.5$.




This empirical study very nicely complements the asymptotic analysis from the previous example. Indeed, the (squared) bias decays while the variance grows for growing model complexity. For a neighborhood size of $k = 1$ we very clearly see overfitting. For large neighborhood sizes, we notice some underfitting.
This example including the shown figure can be reproduced and modified here: launch binder.
△

We conclude this section with the proof of Theorem 7.2. While on first sight, the proof might look very technical, it is indeed only the concatenation of many small and simple ideas. Therefore, even the mathematically less interested reader is encouraged, to still try to follow the line of arguments in the proof.

Proof of Theorem 7.2
The proof simply applies Theorem 7.1 to the kNN regression predictor, which is, for given training data $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ given by

$$f_{kNN,\mathcal{T}}(\boldsymbol{x}_0) = \frac{1}{k} \sum_{\boldsymbol{x}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} y_i \,.$$

We recall here, that $\mathcal{N}_k(\boldsymbol{x}_0)$ gives the $k$ nearest neighbors $\boldsymbol{x}_i$ from the training set $\mathcal{T}$. If we consider the training set random variable $T = \{(\mathscr{X}_i, \mathscr{Y}_i)\}_{i=1}^N$ instead of the realization $\mathcal{T}$, we obtain

$$f_{kNN,T}(\boldsymbol{x}_0) = \mathrm{E}_T \left( \frac{1}{k} \sum\nolimits_{\mathscr{X}_i \in \mathcal{N}_k(\boldsymbol{x}_0)} \mathscr{Y}_i \right) ,$$

while recalling that the outputs are given by

$$\mathscr{Y}_i = f_{exact}(\mathscr{X}_i) + \mathscr{E}_i \,.$$

We start the derivation by calculating the bias as

$$\begin{aligned}
&\mathrm{E}_T\left(f_{kNN,T}(\boldsymbol{x}_0)\right) - f_{exact}(\boldsymbol{x}_0) = \mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} \mathscr{Y}_i\right) - f_{exact}(\boldsymbol{x}_0)\\
=&\mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i) + \mathscr{E}_i\right) - f_{exact}(\boldsymbol{x}_0)\\
=&\frac{1}{k}\,\mathrm{E}_T\left(\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right) + \mathrm{E}_T\left(\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} \mathscr{E}_i\right) - f_{exact}(\boldsymbol{x}_0)\\
=&\frac{1}{k}\,\mathrm{E}_T\left(\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right) - f_{exact}(\boldsymbol{x}_0)
\end{aligned}$$

Here, we use the definition of the outputs and the linearity of the expectation. The only non-trivial transformation from the third to the fourth line uses

$$\mathrm{E}_T\left(\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} \mathscr{E}_i\right) = \mathrm{E}_T\left(\sum\nolimits_{i=1}^{k} \mathscr{E}_i\right) = \sum\nolimits_{i=1}^{k} \mathrm{E}_T\left(\mathscr{E}_i\right) = 0\,. \tag{7.7}$$

In this chain of tranformations, the first one is a bit challenging. In principle, the sum over the nearest neighbors of $\boldsymbol{x}_0$ would explicitly select those noise variables $\mathscr{E}_i$ that are associated to these nearest neighbors. However, these noise variables are random variables that are identically distributed and independent from the inputs $\mathscr{X}_i$. Therefore, we can select *any* subset of $k$ noise variables and will obtain the same mean result. As we can select any subset of $k$ noise variables, we can just select the first $k$ of them, which leads to the given first transformation. The rest is linearity and the definition of the noise.

The next part of the proof computes the variance term for the kNN predictor. We start with a simple standard transformation for the variance.

$$\mathrm{E}_T\left(\left(f_{kNN,T}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_{kNN,T}(\boldsymbol{x}_0)\right)\right)^2\right) = \mathrm{E}_T\left(f_{kNN,T}(\boldsymbol{x}_0)^2\right) - \left(\mathrm{E}_T\left(f_{kNN,T}(\boldsymbol{x}_0)\right)\right)^2 \tag{7.8}$$

To keep the further derivation readable, we first evaluate the first term on the right-hand

side of the above equation by

$$\begin{aligned}
&\mathrm{E}_T\left(f_{kNN,T}(\boldsymbol{x}_0)^2\right) = \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)}\mathscr{Y}_i\right]^2\right) = \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)+\mathscr{E}_i\right]^2\right)\\
=&\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]^2 + 2\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right] + \left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)}\mathscr{E}_i\right]^2\right)\\
=&\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]^2\right) + \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]^2\right) + 2\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]\right)\\
=&\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]^2\right) + \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]^2\right) + 2\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]\right)\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]\right)\\
=&\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]^2\right) + \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]^2\right) + 2\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]\right)\cdot 0\\
=&\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]^2\right) + \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]^2\right).
\end{aligned}$$

The first row ist just the application of the definitions, while the second and third row multiply out the square and apply linearity and some reordering. (Starting from the second row, we abbreviate the summation index.) In the fourth row, we claim that

$$2\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]\right) = 2\,\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i} f_{exact}(\mathscr{X}_i)\right]\right)\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i}\mathscr{E}_i\right]\right),$$

which is true since the $\mathscr{X}_i$ and the $\mathscr{E}_i$ are independent and the mean of the product of independent random variables is the product of the means of the indedependent random variables. In the fifth row, we again apply the observation from equation (7.7). The rest is trivial. In the last row, the second term can be further simplified to

$$\begin{aligned}
&\mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)}\mathscr{E}_i\right]^2\right) = \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{i=1}^{k}\mathscr{E}_i\right]^2\right) = \mathrm{E}_T\left(\frac{1}{k^2}\sum\nolimits_{i=1}^{k}\sum\nolimits_{j=1}^{k}\mathscr{E}_i\mathscr{E}_j\right)\\
=&\frac{1}{k^2}\sum\nolimits_{i=1}^{k}\sum\nolimits_{j=1}^{k}\mathrm{E}_T\left(\mathscr{E}_i\mathscr{E}_j\right) = \frac{1}{k^2}\sum\nolimits_{i=1}^{k}\mathrm{E}_T\left(\mathscr{E}_i^2\right) = \frac{1}{k^2}\sum\nolimits_{i=1}^{k}\mathrm{E}_T\left(\left(\mathscr{E}_i-\mathrm{E}_T\left(\mathscr{E}_i\right)\right)^2\right)\\
=&\frac{1}{k^2}\sum\nolimits_{i=1}^{k}\sigma_\varepsilon^2 = \frac{1}{k^2}\sum\nolimits_{i=1}^{k}\sigma_\varepsilon^2 = \frac{k}{k^2}\sigma_\varepsilon^2 = \frac{1}{k}\sigma_\varepsilon^2.
\end{aligned}$$

In the first transformation, we use again the argument from the derivation in equation (7.7). Most steps are applications of basic calculations and definitions. A not immediately obvious step is however

$$\frac{1}{k^2}\sum\nolimits_{i=1}^{k}\sum\nolimits_{j=1}^{k}\mathrm{E}_T\left(\mathscr{E}_i\mathscr{E}_j\right) = \frac{1}{k^2}\sum\nolimits_{i=1}^{k}\mathrm{E}_T\left(\mathscr{E}_i^2\right).$$

In that step, we observe that due to the independence of the noise variables, we have

$$\mathrm{E}_T\left(\mathscr{E}_i\mathscr{E}_j\right) = \begin{cases}\mathrm{E}_T\left(\mathscr{E}_i^2\right) & \text{if } i=j\\ 0 & \text{if } i\neq j\end{cases}.$$

This completes the derivation of the first term of the right-hand side of equation (7.8). Next, we evaluate the second term on the right-hand side of that equation to

$$
\begin{aligned}
\left(\mathrm{E}_T\left(f_{kNN,T}(\boldsymbol{x}_0)\right)\right)^2 &= \left(\mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)}\mathscr{Y}_i\right)\right)^2 \\
&= \left(\mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i) + \mathscr{E}_i\right)\right)^2 \\
&= \left(\mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right) + \mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)}\mathscr{E}_i\right)\right)^2 \\
&= \left(\mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right)\right)^2,
\end{aligned}
$$

using the arguments that we have seen before. Finally, we "plug" together all results obtained so far, and get the variance of the kNN regression predictor as

$$
\begin{aligned}
&\mathrm{E}_T\left(\left(f_{kNN,T}(\boldsymbol{x}_0) - \mathrm{E}_T\left(f_{kNN,T}(\boldsymbol{x}_0)\right)\right)^2\right) \\
&= \mathrm{E}_T\left(\left[\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right]^2\right) + \frac{1}{k}\sigma_\varepsilon^2 - \left(\mathrm{E}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right)\right)^2 \\
&= \mathrm{Var}_T\left(\frac{1}{k}\sum\nolimits_{\mathscr{X}_i\in\mathcal{N}_k(\boldsymbol{x}_0)} f_{exact}(\mathscr{X}_i)\right) + \frac{1}{k}\sigma_\varepsilon^2\,.
\end{aligned}
$$

This concludes the proof. □

# 8. Linear classification

So far, we have dealt mostly with the supervised machine learning task of regression, where we seek the hidden relationship between inputs and continuous outputs. *Classification*, instead, seeks the hidden relationship between inputs and one or more discrete outputs in a qualitative variable. Thereby, classification tries to assign inputs to classes or labels, leading to a different application field than regression. Indeed, many of the more well-known modern machine learning tasks like SPAM filtering, handwriting recognition, image classificaiton, etc. are classification tasks.

So why have we not considered classification in more detail, earlier? As a matter of fact, classification tends to be slightly more complicated, when it comes to the construction of even rather simple methods. Therefore, we first established many core notions from machine learning for the regression task. These can often be carried over to the classification task, noting that a proper mathematical analysis tends to be also more involved for classification.

In this chapter, we make a slow start by re-motivating the concept of qualitative variables and by reminding the reader of a few essential concepts that have been discussed before. This will result in a first proposed naive classification method, which however turns out to be suboptimal. Afterwards, *Linear Discriminant Analysis* and *Logistic Regression* are introduced, which are both linear methods to carry out classification.

## 8.1 Introduction

Let us first shed some more light on *qualitative variables*.

**Remark** (Qualitative variables and their encoding) Qualitative variables have a finite range of classes. These could represent, among others

- success and failure,
- a finite number of properties (e.g. {red green, yellow}),
- a finite sequence of numbers (e.g. $\{0, 1, 2, \ldots, 9\}$), or
- ordered categories (e.g. {small, medium, large}).

To allow for a treatment of such classes by mathematical techniques, we usually have to encode these classes into numbers. The actual way, how this is done, is often dependent

of the application, classification method, etc. However, one obvious way to encode a finite number of $r$ classes is to simply replace the classes / class labels by values in $\{1, \dots, r\}$. Another approach goes via *dummy variables* and is often also called *one-hot encoding*. Let us pick the example of {red,green,yellow}. Instead of using values in $\{0, 1, 2\}$, we can also introduce three indicator variables $G_1$, $G_2$, $G_3$, where each variable can take a value in $\{0, 1\}$, such that

- $G_1 = 1$, $G_2 = 0$, $G_3 = 0$ corresponds to "red"
- $G_1 = 0$, $G_2 = 1$, $G_3 = 0$ corresponds to "green", and
- $G_1 = 0$, $G_2 = 0$, $G_3 = 1$ corresponds to "yellow".

From the view point of statistical learning, we would thus have a random output vector $G : \Omega \to \{0, 1\}^3$. △

Earlier on, we have seen that a meaningful way to express the (abstract) prediction error in classification is by

**Definition** (Expected (0-1) predition error in regression) Let $\boldsymbol{X}$, $G$ be input and a one-dimensional output with mixed joint density $\rho(\boldsymbol{x}, g)$ and $f : \mathbb{R}^D \to R_G$ a function to predict $G$ from $\boldsymbol{X}$. For the 0-1 loss, the **expected prediction error of** $f$, $\mathrm{EPE}(f)$, is defined as

$$\mathrm{EPE}(f) = \mathrm{E}\left(L_{0-1}(G, f(\boldsymbol{X}))\right) .$$

The $L_{0-1}$ loss is given by

$$L_{0-1}(g, g') = \begin{cases} 0 & g = g' \\ 1 & g \neq g' \end{cases} .$$

This construction, as seen before, leads to the "best possible" predictor (give the mentioned setup) via

**Theorem** Let $\boldsymbol{X}$, $G : \Omega \to R_G$, $R_G = \{1, \dots, r\}$ be input and qualitative output with joint mixed density $\rho(\boldsymbol{x}, g)$. The function $f : \mathbb{R}^D \to R_G$ that minimizes (under appropriate conditions) the expected prediction error $\mathrm{EPE}(f)$ with respect to the 0-1 loss is given by

$$f(\boldsymbol{x}) = \underset{g \in R_G}{\arg\min}[1 - p(g|\boldsymbol{x})] ,$$

which can be simplified to

$$f(\boldsymbol{x}) = \mathfrak{g} \quad \text{if} \quad p(\mathfrak{g}|x) = \max_{g \in R_G} p(g|\boldsymbol{x})$$

This minimizer is the **Bayes classifier**.

Even though this statement still seems to be quite abstract, it should be noted that

many practical classification methods are derived from the *Bayes classifier*, as we have seen it already with kNN classification.

**Remark** (Construction idea of new classification methods) Recalling regression, where we build new methods by approximating the regressor $\mathrm{E}(Y|\boldsymbol{X} = \boldsymbol{x})$, classification methods are derived by approximating the Bayes classifier. Indeed, a different way to write the Bayes classifier is by writing

$$f(\boldsymbol{x}) = \arg\max_{g \in \mathbb{R}_G} p(g|\boldsymbol{x})\,.$$

Hence, the obvious idea is to replace $p(g|\boldsymbol{x})$ by some approximation, while keeping the maximization. △

In this chapter, we will only consider *linear* methods for classification. However, what makes a classification method linear? To clarify this, we first need an important notion in classification.

**Definition 8.1** (Decision boundary) Let $\boldsymbol{X}$, $G : \Omega \to R_G$, $R_G = \{1, \ldots, r\}$ be input and qualitative output with mixed joint density $\rho(\boldsymbol{x}, g)$. In the case of $r = 2$, i.e. two classes, the **decision boundary** between these two classes is given as the set of points $\boldsymbol{x} \in \mathbb{R}^D$ at which the probability of $\boldsymbol{x}$ to be in either of the classes is equal, i.e.

$$\{\boldsymbol{x} \in \mathbb{R}^D \,|\, p(1|\boldsymbol{x}) = p(2|\boldsymbol{x})\}\,.$$

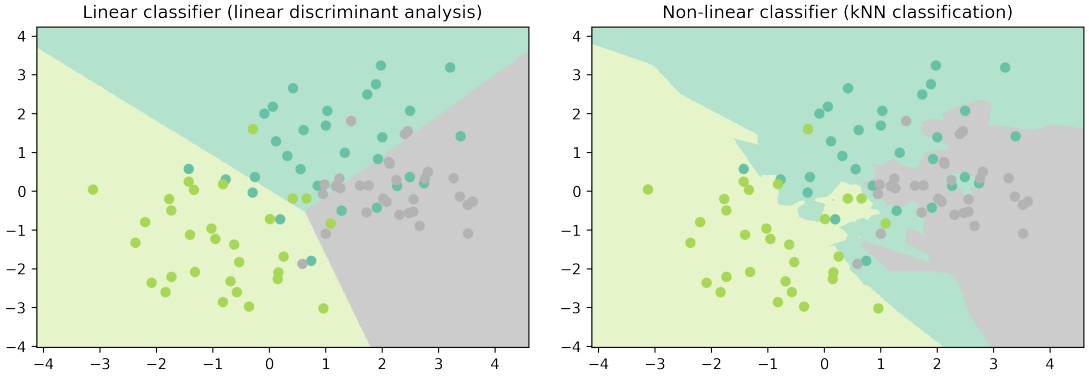
For $r > 2$, we consider the mutual decision boundaries between pairs of classes.

We will soon give a visual idea of a decision boundary. Before that, we however state what makes a classification method *linear*.

**Definition 8.2** We call a method for classification **linear** if the constructed predictor has linear decision boundaries between the classes.

Let us clarify the visual interpretation of a linear decision boundary in

**Example 8.1** (Linear vs. non-linear decision boundaries) We reconsider the data from Example 3.5. For this data set, we build a linear classifier via the soon to be introduced *linear discriminant analysis*. Moreover, we again build the non-linear kNN classifier with $k = 3$. A visualization of the predictors obtained with both methods is given below.




For the linear method, the decision boundary, which gives the location of the change between two classes, is a straight line. In the non-linear case, it can be arbitrary.
This example including the shown figures can be reproduced and modified here: launch binder.
△

We come to our first proposal for a linear classifier.

**Method 10** (Classification by linear regression of indicator matrix) As discussed before, the construction of a classification method implies finding an approximation to the conditional PMF $p(g|\boldsymbol{x})$. A naive approach to do this is by least squares linear regression. To this end, we are given training data $\{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$, $g_i \in \{1, \dots, r\}$. The idea of the *linear regression of indicator matrix* approach is to build for each choice of $g \in \{1, \dots, r\}$ a linear model that approximates $p(g|\boldsymbol{x})$ as

$$f^{(g)}(\boldsymbol{x}) \approx p(g|\boldsymbol{x})\,.$$

Hence we build a training set

$$\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$$

with

$$\boldsymbol{y}_i = \begin{pmatrix} y_{i1} \\ \vdots \\ y_{ir} \end{pmatrix}, \quad y_{ij} = p(j|\boldsymbol{x}_i) = P(G = j|\boldsymbol{X} = \boldsymbol{x}_i) = \begin{cases} 1 & g_i = j \\ 0 & g_i \neq j \end{cases}\,.$$

The rationale behind this construction is that we certainly have $p(g_i|\boldsymbol{x}_i) = 1$ and $p(g|\boldsymbol{x}_i) = 0$ for all other $g \neq g_i$ at the location of the training samples. In other words, we certain that at location $x_i$ the class label is $g_i$ (i.e. have a probability of 1 for this) and are also certain that at this location the class label is not any of the other class labels (i.e. have a probability of 0 for the other cases).

Then, we apply linear regression by least squares for an output dimension of $K = r$ to this training set and obtain the $r$ predictors

$$f(\boldsymbol{x}) = \begin{pmatrix} f^{(1)}(\boldsymbol{x}) \\ \vdots \\ f^{(r)}(\boldsymbol{x}) \end{pmatrix}\,.$$

Finally, the predictor for $G$ is given as

$$\widehat{G} = \arg\max_{g \in \{1,\ldots,r\}} f^{(g)}(\boldsymbol{x})\,.$$

△

The just constructed method is linear, since the decision boundary between two classes $g$ and $g'$ is the solution of the *linear* system of equations

$$f^{(g)}(\boldsymbol{x}) = f^{(g')}(\boldsymbol{x})\,.$$

Let us consider a brief

**Example 8.2** We are given the training data $\mathcal{T} = \{(-1,1),(-0.5,1),(1,2)\}$, hence we have the classes $R_G = \{1,2\}$. The resulting training data for linear regression with $K = 2$ is

$$\mathcal{T}' = \left\{\left(-1, \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right)\left(-0.5, \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right)\left(1, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right)\right\}$$

Two linear models $f^{(1)}$ and $f^{(2)}$, approximating $p(1|x)$ and $p(2|x)$ are built from the training sets $\mathcal{T}^{(1)} = \{(-1,1),(-0.5,1),(1,0)\}$ and $\mathcal{T}^{(2)} = \{(-1,0),(-0.5,0),(1,1)\}$. The remaining steps are obvious. △

We present an algorithm for the this approach.

**Algorithm 11** (Classification by linear regression of indicator matrix)
**input:** $\{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$ training data ($g_i \in \{1,\ldots,r\}$), $\boldsymbol{x}$ evaluation point
**output:** linear regression prediction $\hat{G}(\boldsymbol{x})$

1. build matrix $\mathcal{X}$ *(see linear regression)*
2. build matrix $\mathcal{Y} \in \mathbb{R}^{N\times r}$ with entries $y_{ij} = \begin{cases} 1 & g_i = j \\ 0 & g_i \neq j \end{cases}$
3. $\mathcal{C} \leftarrow \mathcal{X}^\top \mathcal{Y}$
4. $\boldsymbol{A} \leftarrow \mathcal{X}^\top \mathcal{X}$
5. solve $\mathcal{A}\hat{\mathcal{B}} = \boldsymbol{C}$ using Cholesky factorization
6. $\boldsymbol{z} \leftarrow (1, \boldsymbol{x}^\top)$
7. $\hat{\boldsymbol{y}} \leftarrow \boldsymbol{z}\hat{\mathcal{B}}$
8. return $\arg\max_{j\in\{1,\ldots,r\}} \hat{y}_j$

The complexity of this algorithm can easily be derived as $O(D^3 + D^2 \cdot N + D \cdot N \cdot r)$. Looking at the steps discussed so far, the introduced method is the obvious linear classifier. However, in practice, the approach has some issues.

**Remark** (Issues of classification by linear regression of indicator matrix) The individual approximations $f^{(g)}(\boldsymbol{x})$ to the conditional PMF $p(g|\boldsymbol{x})$ do not fulfill the statistical properties of a conditional PMF. This means that it may hold

$$f^{(g)}(\boldsymbol{x}) \notin [0,1] \quad \text{for } g \in \{1,\dots,r\} \qquad \text{or} \qquad \sum_{g=1}^{r} f^{(g)}(\boldsymbol{x}) \neq 1\,.$$

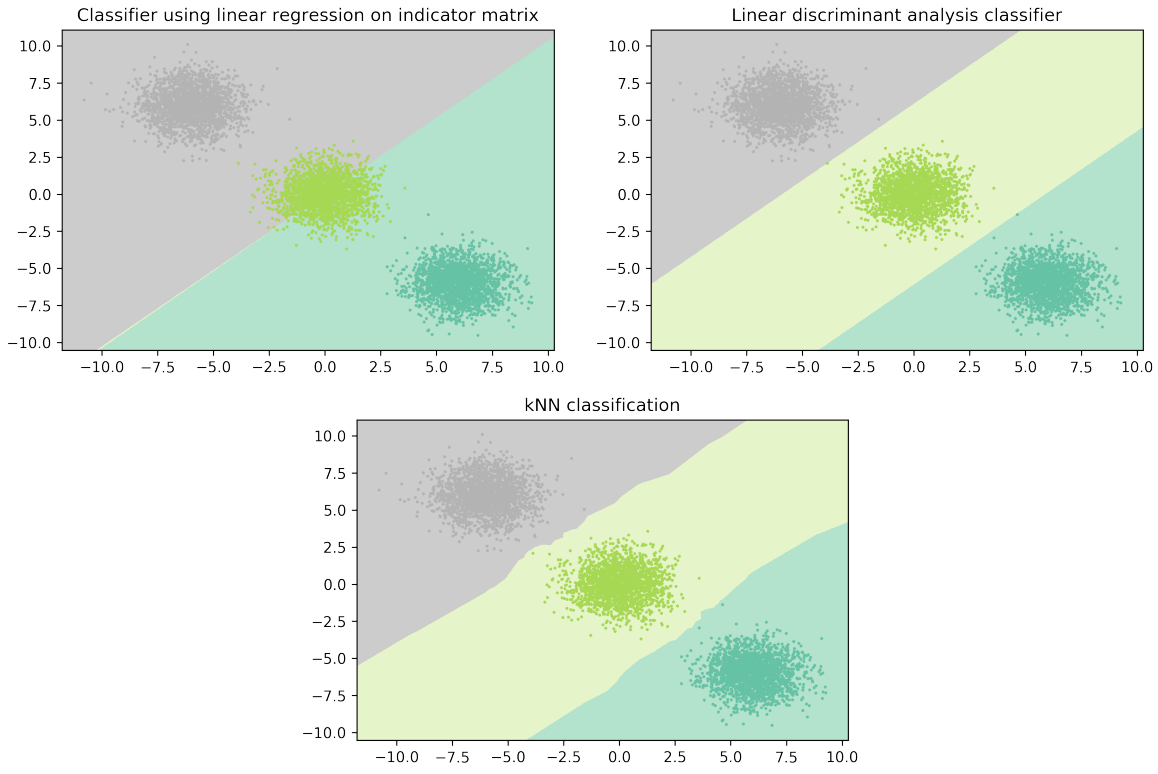
One of the consequences of this issue is a phenomenon called *masking* that can happen for this classification approach. If masking happens, a classifier might never predict a given class, even though training data indicated the existence of that class. △

We give give a classical example for masking.

**Example 8.3** (Masking in classification by linear regression of the indicator matrix) In this example, we slightly modify the data from Example 3.5 and take of each of the following distributions (giving inputs to the three labels) 2000 samples.

$$\mathcal{N}\left(\begin{pmatrix} 6 \\ -6 \end{pmatrix}, \begin{pmatrix} 1.2 & 0 \\ 0 & 1.2 \end{pmatrix}\right), \quad \mathcal{N}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1.2 & 0 \\ 0 & 1.2 \end{pmatrix}\right), \quad \mathcal{N}\left(\begin{pmatrix} -6 \\ 6 \end{pmatrix}, \begin{pmatrix} 1.2 & 0 \\ 0 & 1.2 \end{pmatrix}\right)$$

Below are the visualizations for the previously introduced classifier, a "proper" linear classifier and the kNN classifier.




We observe that for the linear regression classification approach, the class for the central points is not given at all, even though the labels from the different classes are well distinct. Both other methods work as expected on the given data.

This example including the shown figures can be reproduced and modified here: launch binder. △

In Section 8.3, we will introduce the *logistic regression*, which is very similar to our just discussed naive approach. However, it will be built such that the necessary statistical properties of the approximation to the conditional PMF $p(g|\boldsymbol{x})$ are enforced. As a result that method will also have no masking problem.

Before, we come to *logistic regression*, we will however introduce *linear discriminant analysis*, which, from a technical perspective, is a bit simpler to consider.

## 8.2 Linear discriminant analysis

In the classification method *linear discriminant analysis*, we again approximate the conditional PMF $p(g|\boldsymbol{x})$, which is often called *class posterior*. However, this time, we first transform $p(g|\boldsymbol{x})$ using Bayes theorem for mixed joint densities, which we give here without proof.

**Theorem 8.1** (Bayes' theorem in case of mixed joint densities) Let $(\boldsymbol{X}, G)$ be a continuous random vector and a discrete random variable with joint mixed density $\rho(\boldsymbol{x}, g)$. For all $\boldsymbol{x} \in \mathbb{R}^D$ and all $g \in R_G$, such that $\rho_{\boldsymbol{X}}(\boldsymbol{x}) > 0$ and $p_G(g) > 0$, it holds

$$p(g|\boldsymbol{x}) = \frac{\rho(\boldsymbol{x}|g)p_G(g)}{\rho_{\boldsymbol{X}}(\boldsymbol{x})}\,.$$

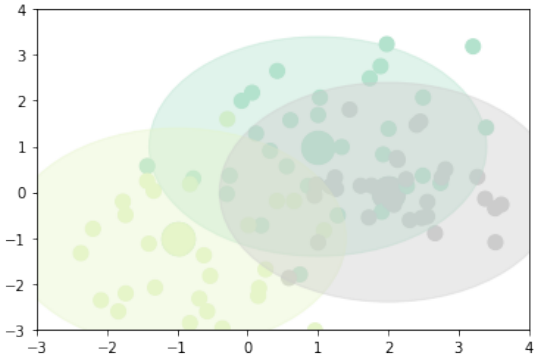
Using Bayes' theorem, we can replace the class posterior $p(g|\boldsymbol{x})$ by a combination of the conditional density $\rho(\boldsymbol{x}|g)$, the marginal density $\rho_{\boldsymbol{X}}(\boldsymbol{x})$ and the marginal PMF $p_G(g)$. The conditional density $\rho(\boldsymbol{x}|g)$ is often called *likelihood*. $p_G(g)$ gives the *class probability*. Moreover, $\rho_{\boldsymbol{X}}(\boldsymbol{x})$ is sometimes called *marginal likelihood*.

However, how will this help us to build a classifier? We get a partial answer by revisiting Examples 3.5 and 8.1 and thereby getting a more intuitive idea for the likelihood $\rho(\boldsymbol{x}|g)$.

**Example 8.4** (Likelihood) In Examples 3.5 and 8.1, we obtained the training data by sampling from three two-dimensional random variables $\boldsymbol{X}_{purple}$, $\boldsymbol{X}_{green}$, $\boldsymbol{X}_{turquoise}$ that follow the Gaussian distributions with means

$$\{(1,1)^\top, (-1,-1)^\top, (2,0)^\top\}$$

and the identity covariance matrix. The below figure tries to explicitly visualize the three different distributions from which the training data is taken.




By mixing the samples from all three random variables with appropriate labels into one training set, we thus generated a training set, in which we exactly know the likelihoods

$$\rho(\boldsymbol{x}|g_{purple}), \rho(\boldsymbol{x}|g_{green}), \rho(\boldsymbol{x}|g_{turquoise})\,,$$

as these are exactly the associated densities of the three different Gaussian distributions. In other words, a single likelihood $\rho(\boldsymbol{x}|g)$ describes the distribution of the input samples that have the label $g$.
This example including the shown figure can be reproduced and modified here: launch binder.
△

*Linear discriminant analysis* (LDA) picks up this observation and uses it to build a classification method.

**Method 11** (Linear discriminant Analysis (LDA)) In linear discriminant analysis, we start from the class posterior $p(g|\boldsymbol{x})$, which we replace using Bayes' theorem by $\frac{\rho(\boldsymbol{x}|g)p_G(g)}{\rho_{\boldsymbol{X}}(\boldsymbol{x})}$. As always, we are given training data $\mathcal{T} = \{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$, $g_i \in \{1, \ldots, r\}$. In a next step, we make the assumption (and thereby make the approximation) that the training input samples for each class $g$, i.e.

$$\mathcal{T}^{(g)} = \{\boldsymbol{x}_i | (\boldsymbol{x}_i, g_i) \in \mathcal{T}, g_i = g\}\,,$$

have each been drawn from a random variable $\boldsymbol{X}^{(g)}$ that follows a normal distribution with mean $\boldsymbol{\mu}_g$ and some $g$-independent covariance $\boldsymbol{\Sigma}$, hence

$$\boldsymbol{X}^{(g)} \sim \mathcal{N}(\boldsymbol{\mu}_g, \boldsymbol{\Sigma})\,.$$

Thereby, we can explicitly give the likelihood $\rho(\boldsymbol{x}|g)$ as

$$\rho(\boldsymbol{x}|g) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_g)^\top \Sigma^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_g)\right), \quad \text{for } 1, \ldots, r\,.$$

Note that this assumption is a crucial part of the LDA method. Other assumptions, e.g. a different type of distribution or the choice of a *g-dependent* covariance lead to other methods.

Having "re-written" the conditional PMF $p(g|\boldsymbol{x})$ via likelihood, class probability and marginal likelihood, the actual fitting / training / optimization part in LDA is the search for the means $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_r \in \mathbb{R}^d$, for the covariance $\Sigma \in \mathbb{R}^{D \times D}$ and for the class probabilities $p_G(1), \dots, p_G(r)$ based on the given training data.[1)] △

The actual technical details of linear discriminant analysis are summarized by

**Theorem 8.2** (Linear Discriminant Analysis) Let $\boldsymbol{X}$, $G \in \{1, \dots, r\} =: R_G$ be input and qualitative output with mixed joint density $\rho(\boldsymbol{x}, g)$ and $\rho_{\boldsymbol{X}}(\boldsymbol{x}) > 0$, $p_G(g) > 0$. By approximating the conditional density $\rho(\boldsymbol{x}|g)$ using the density of the multivariate normal distribution $\mathcal{N}(\boldsymbol{\mu}_g, \boldsymbol{\Sigma})$, i.e.

$$\rho(x|g) \approx \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_g)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_g)\right), \quad \text{for all } g = 1, \dots, r\,,$$

the Bayes classifier $f_b$ is approximated as

$$f_b(\boldsymbol{x}) \approx \underset{g \in R_G}{\arg\max} \left(\log p_G(g) + \boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g - \frac{1}{2}\boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g\right).$$

Proof
We start with the Bayes classifier $f_b$, which is given by

$$f_b(\boldsymbol{x}) = \underset{g \in \mathbb{R}_G}{\arg\max}\, p(g|\boldsymbol{x})\,.$$

Let us focus for now on the conditional PMF $p(g|\boldsymbol{x})$ and apply the transformations and approximations stated in Method 11 with

$$\begin{aligned} p(g|\boldsymbol{x}) &= \frac{\rho(\boldsymbol{x}|g)p_G(g)}{\rho_{\boldsymbol{X}}(\boldsymbol{x})} = p_G(g)\frac{1}{\rho_{\boldsymbol{X}}(\boldsymbol{x})}\rho(\boldsymbol{x}|g) \\ &\approx p_G(g)\frac{1}{\rho_{\boldsymbol{X}}(\boldsymbol{x})}\ \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_g)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_g)\right) \\ &= p_G(g)\, c\, \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_g)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_g)\right). \end{aligned}$$

Note that we are overall only interested to find $g$, such that the above term is maximized. Therefore, the term $\frac{1}{\rho_{\boldsymbol{X}}(\boldsymbol{x})}\ \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}}$, which is independent of $g$, can safely be replaced by a constant $c$ in the last step.

Next, we transform the above equation using the logarithm function. Since the logarithm is a monotone function, the final maximization will not be affected by that transforma-

[1)] Why do we not have to also find $\rho_{\boldsymbol{X}}(\boldsymbol{x})$? We see that in the soon to come theorem and proof.

tion. We thus further obtain

$$\begin{aligned}\log(p(g|\boldsymbol{x})) &\approx \log\left(p_G(g)\, c\, \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_g)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_g)\right)\right)\\ &= \log(p_G(g)) + \log(c) - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_g)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_g)\end{aligned}$$

using basic calculation rules for the logarithm. For the last term, we briefly observe

$$\begin{aligned}-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_g)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_g) &= \frac{1}{2}\left(\boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{x} - \boldsymbol{x}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g - \boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{x} + \boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g\right)\\ &= -\frac{1}{2}\boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{x} + \boldsymbol{x}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g - \frac{1}{2}\boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g\,,\end{aligned}$$

where we use that $\boldsymbol{x}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g = \boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{x}$, since $\boldsymbol{\Sigma}^{-1}$ is symmetric. With this, we continue

$$\begin{aligned}\log(p(g|\boldsymbol{x})) &\approx \log(p_G(g)) + \log(c) - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_g)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_g)\\ &= \log(p_G(g)) + \log(c) - \frac{1}{2}\boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{x} + \boldsymbol{x}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g - \frac{1}{2}\boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g\\ &= \log(p_G(g)) + c' + \boldsymbol{x}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g - \frac{1}{2}\boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g\,.\end{aligned}$$

In the last step, we have taken the $g$-independent terms $\log(c) - \frac{1}{2}\boldsymbol{x}^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{x}$ and have replaced them by the constant $c'$.
We conclude by carrying out the minimization process as

$$\begin{aligned}f_b(\boldsymbol{x}) &= \operatorname*{arg\,max}_{g\in R_G} p(g|\boldsymbol{x}) = \operatorname*{arg\,max}_{g\in R_G} \log(p(g|\boldsymbol{x}))\\ &\approx \operatorname*{arg\,max}_{g\in R_G}\left(\log(p_G(g)) + c' + \boldsymbol{x}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g - \frac{1}{2}\boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g\right)\\ &= \operatorname*{arg\,max}_{g\in R_G}\left(\log(p_G(g)) + \boldsymbol{x}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g - \frac{1}{2}\boldsymbol{\mu}_g^\top \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_g\right)\,.\end{aligned}$$

The only new observation is the last step, in which we drop $c'$ due to its $g$-independence. This concludes the proof. □

The above theorem shows that the constructed classifier has as unknown quantities $\{\boldsymbol{\mu}_g\}_{g=1}^r$, $\boldsymbol{\Sigma}$ and $\{p_G(g)\}_{g=1}^r$. As said before, these need to be estimated from the data. The necessary estimators are introduced in

**Definition 8.3** (Estimators in LDA) With the setting of Theorem 8.2, we define, for

given training data $\mathcal{T} = \{(x_i, g_i)\}_{i=1}^N$, the following estimators:

$$p_G(g) \approx \widehat{p_G(g)} := \frac{N_g}{N}$$
$$\boldsymbol{\mu}_g \approx \widehat{\boldsymbol{\mu}_g} := \frac{1}{N_g} \sum_{(x,g)\in\mathcal{T}} x$$
$$\boldsymbol{\Sigma} \approx \widehat{\boldsymbol{\Sigma}} := \sum_{g=1}^{r} \sum_{(x,g)\in\mathcal{T}} (x - \widehat{\boldsymbol{\mu}_g})(x - \widehat{\boldsymbol{\mu}_g})^\top / (N - r)$$

where $N_g := \left|\left\{(x, g) \in \mathcal{T} \middle| x \in \mathbb{R}^D\right\}\right|$ is the number of training samples in class $g$.

The fact that $\widehat{\boldsymbol{\mu}_g}$ and $\widehat{\boldsymbol{\Sigma}}$ are *unbiased* estimators for the mean and the covariance goes back to basic knowledge from statistics.

As we now know how to determine the remaining unknown quantities from the data, we can state the full algorithm for linear discriminant analysis.

**Algorithm 12** (Linear discriminant analysis)

**input:** $\{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$ training data ($g_i \in \{1, \ldots, r\}$),
$\boldsymbol{x}$ evaluation point

**output:** linear regression prediction $\hat{g}$

1. $N_g \leftarrow \left|\left\{(\boldsymbol{x}, g) \in \mathcal{T} \middle| \boldsymbol{x} \in \mathbb{R}^D\right\}\right|$
2. $\widehat{p_G(g)} \leftarrow \frac{N_g}{N}$
3. $\widehat{\boldsymbol{\mu}_g} \leftarrow \frac{1}{N_g} \sum_{(\boldsymbol{x},g)\in\mathcal{T}} \boldsymbol{x}$
4. $\widehat{\boldsymbol{\Sigma}} \leftarrow \sum_{g=1}^{r} \sum_{(\boldsymbol{x},g)\in\mathcal{T}} (\boldsymbol{x} - \widehat{\boldsymbol{\mu}_g})(\boldsymbol{x} - \widehat{\boldsymbol{\mu}_g})^\top / (N - r)$
5. $\hat{g} \approx \arg\max_{g\in R_G} \left(\log \widehat{p_G(g)} + \boldsymbol{x}^\top \widehat{\boldsymbol{\Sigma}}^{-1} \widehat{\boldsymbol{\mu}_g} - \frac{1}{2} \widehat{\boldsymbol{\mu}_g}^\top \widehat{\boldsymbol{\Sigma}}^{-1} \widehat{\boldsymbol{\mu}_g}\right)$
6. return $\hat{g}$

It can be easily seen that the complexity of the linear discriminant analysis is $O(N \cdot D^2 + D^3)$. Furthermore, we can state

**Lemma 8.1** The linear discriminant analysis is a linear predictor.

Proof

We need to show that the decision boundary between two classes $g$, $g'$ is linear. The decision boundary is given by

$$\log \widehat{p_G(g)} + \boldsymbol{x}^\top \widehat{\boldsymbol{\Sigma}}^{-1} \widehat{\boldsymbol{\mu}_g} - \frac{1}{2} \widehat{\boldsymbol{\mu}_g}^\top \widehat{\boldsymbol{\Sigma}}^{-1} \widehat{\boldsymbol{\mu}_g} = \log \widehat{p_G(g')} + \boldsymbol{x}^\top \widehat{\boldsymbol{\Sigma}}^{-1} \widehat{\boldsymbol{\mu}_{g'}} - \frac{1}{2} \widehat{\boldsymbol{\mu}_{g'}}^\top \widehat{\boldsymbol{\Sigma}}^{-1} \widehat{\boldsymbol{\mu}_{g'}}$$

By carefully investigating this equation, we observe that it is indeed linear in $\boldsymbol{x}$. □

In earlier chapters, we spend quite some efforts to discuss training and generalization error of a given method. All that knowledge carries over to the case of classification, as further outlined in

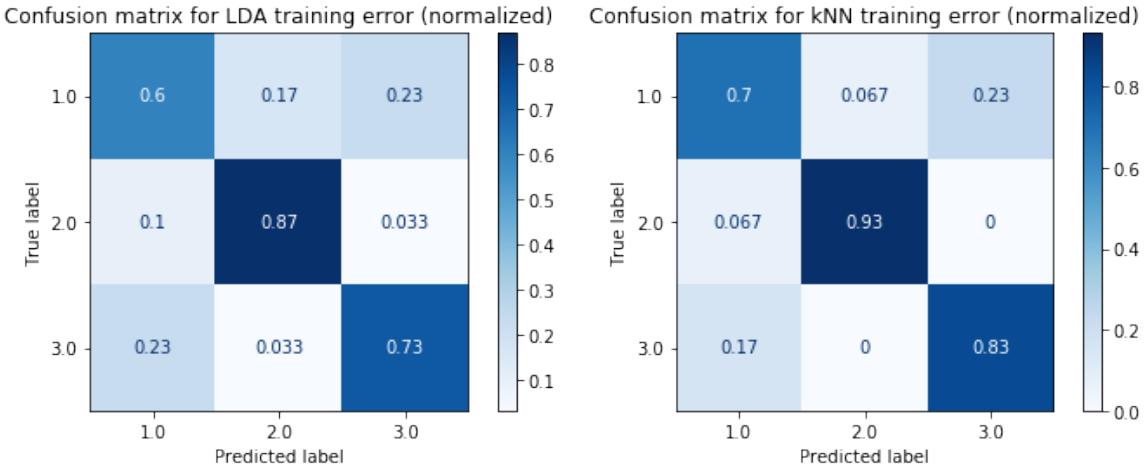
**Remark** (Training and generalization error estimation) We can use the same techniques for training and (expected) generalization error estimation as described in Chapter 6. Indeed, we only need a correspondingly adapted loss function. While we only considered the squared loss $L_2$ for the regression case, we often consider the 0-1-loss for classification. In classification we usually work with more than two classes. In this case the 0-1-loss gives no information on the "structure" of the mis-classifications, hence we cannot observe whether e.g. a given training sample always incorrectly gets classified as member of the class 2 but never as class 1, while the correct classification would be class 3. This is why it is often useful to introduce a *confusion matrix*, given below

| | predicted class | | | | |
|---|---|---|---|---|---|
| true class | 1 | 2 | 3 | $\cdots$ | $r$ |
| 1 | | | | | |
| 2 | | | | | |
| 3 | | | | | |
| $\vdots$ | | | | | |
| 5 | | | | | |

In cell $(g, g')$ of the above table, we thus give the absolute number or percentage of samples in e.g. a validation set $\mathcal{T}_{var}$ that do belong to class $g$, however are classified as being members of class $g'$. △

Let us give an example for confusion matrices in

**Example 8.5** We continue with the data from Example 8.1 and obtain for the *training error* the confusion matrices




for the classifiers built by linear discriminant analysis (left) and kNN classification (right) with $k = 3$. Similarly, we can build confusion matrices for the various generalization error estimations.

This example including the shown confusion matrices can be reproduced and modified here: launch binder. △

We finally note that the general construction idea of LDA can also lead to other classifiers.

**Remark** (Construction of further classification methods) In Method 11, we introduced the LDA by using Bayes' theorem and then approximating the likelihood $\rho(\boldsymbol{x}|g)$ by densities following the normal distribution with class-dependent mean and class-independent covariance.

If we replace in this construction the class-independent covariance by a class-*dependent* covariance, i.e. we estimate the different covariances only within the training samples of a given class, we obtain a new method. This new method is called *quadratic discriminant analysis* and is no longer a linear, but a quadratic method. It is a method of higher model complexity.

A very well-known, simple classification method is the *naive Bayes classifier*. We obtain this method, if we make the strongly simplifying assumption that the different dimension-wise random variables $\{X_1, \ldots, X_D\}$ that form the dimensions of the input $\boldsymbol{X}$, are independent. In that case we obtain that the likelihood can be expressed as

$$\rho(\boldsymbol{x}|g) = \rho(x_1|g) \cdot \ldots \cdot \rho(x_D|g)\,.$$

The naive Bayes classifier is found, if we approximate the $\rho(x_j|g)$ individually by the density of the univariate normal distribution $\mathcal{N}(\boldsymbol{\mu}_g, \sigma_g)$.

There are some other methods that can be derived using similar ideas. △

## 8.3 Logistic regression

At the beginning of the chapter, we introduced classification by linear regression for the indicator matrix, where we tried to directly approximate the class posterior $p(g|\boldsymbol{x})$ using a linear model. It turns our that this method is not ideal as the obtained predictor for the class posterior does not follow the typical laws from probability theory.

In this section, we introduce a modified approach of the earlier idea. It is called *logistic regression*. It fixes the earlier problems and provides a meaningful way to do linear classification. However, why did we not introduce this approach earlier? Indeed, as we will see, the training in logistic regression is somewhat more involved. Therefore, we first made the detour via LDA and only come back to this approach, now.

**Method 12** ((Multinomial) Logistic regression) We start from given training data $\{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$, $g_i \in \{1, \dots, r\}$. In *(multinomial)*[2)] *logistic regression*, we construct for $g = 1, \dots, r$ approximations to the class posterior $p(g|\boldsymbol{x})$. These are obtained by first introducing $r$ linear models

$$s_g(\boldsymbol{x}) = \beta_0^{(g)} + \sum_{i=1}^{D} \beta_i^{(g)} x_i \qquad \text{for } g = 1, \dots, r\,,$$

hence one model per class (posterior). Together, the set of coefficients of all models is

$$\mathcal{B} = \left\{ \beta_i^{(g)} \right\}_{g=1,\dots r, i=0,\dots,D}\,.$$

Then, we apply to each of the linear models the so-called *softmax function* and approximate the class posteriors by

$$p_{\mathcal{B}}(g|\boldsymbol{x}) \approx \frac{\exp(s_g(\boldsymbol{x}))}{\sum_{h=1}^{r} \exp(s_h(\boldsymbol{x}))} \qquad \text{for } g = 1, \dots, r\,.$$

It remains to find "appropriate" weights $\mathcal{B}$. △

In difference to linear regression for the indicator matrix, we have

**Theorem 8.3** Let $\boldsymbol{X}, G$ be quantitative input and qualitative output with $R_G = \{1, \dots, r\}$. Let further $p_{\mathcal{B}}(g|\boldsymbol{x})$ be the class posterior as introduced in Method 12 with $\mathcal{B} = \{\beta_i^{(g)}\}_{g=1,\dots r, i=0,\dots,D}$. It holds for all $\boldsymbol{x} \in \mathbb{R}^D$ and $g \in \{1, \dots, r\}$ that

$$\sum_{g \in \{1,\dots,r\}} p_{\mathcal{B}}(g|\boldsymbol{x}) = 1$$

and

$$p_{\mathcal{B}}(g|\boldsymbol{x}) \in [0, 1]\,.$$

[2)] *Multinomial* stands for having more than two classes.

To be concise, we skip the proof here. Indeed, summarizing the theorem, the approximation that we make in logistic regression for the class posterior fulfills the typical laws from probability for a conditional PMF. This is what we were lacking in our earlier attempt to construct a classifier from a linear model.

So far, we only identified our model, but did not indicate, how to obtain the coefficients $\mathcal{B} = \{\beta_i^{(g)}\}_{g=1,\dots r, i=0,\dots,D}$. Clearly, the approximated class posterior is no longer a linear function in the inputs (even though it was built from a linear model).[3] Therefore, a fit by linear regression by least squares following e.g. Theorem 4.3 is no longer possible. Intuitively, we would now tend to use the general estimator of the form

$$\mathcal{B} = \arg\min_{\mathcal{B}} \sum_{i=1}^{N} L(g_i, f_{\mathcal{B}}(\boldsymbol{x}_i))$$

with $f_{\mathcal{B}}(\boldsymbol{x}_i) = \arg\max_{g\in\{1,\dots,r\}} p_{\mathcal{B}}(g|\boldsymbol{x}_i)$ and maybe a 0-1 loss. Nevertheless, the resulting minimization problem is not that easy to solve, since we still have the discrete selection of the class of $f_{\mathcal{B}}$ in it.

Therefore, we instead minimize the error in the *class posterior*

$$\mathcal{B} = \arg\min_{\mathcal{B}} \sum_{i=1}^{N} L_{CE}(p(\cdot|\boldsymbol{x}_i), p_{\mathcal{B}}(\cdot|\boldsymbol{x}_i))\,.^{[4]}$$

and introduce the *cross entropy loss* by

**Definition 8.4** (Cross entropy loss) Let $\boldsymbol{X}$, $G$ be quantitative input and $r$-class qualitative output as before. Let further $p(g|\boldsymbol{x})$, $p'(g|\boldsymbol{x})$ be class posteriors or approximated class posteriors. Then the **cross entropy loss** is given by

$$L_{CE}(p(\cdot|\boldsymbol{x}), p'(\cdot|\boldsymbol{x}_i)) = -\sum_{g=1}^{r} p(g|\boldsymbol{x}) \log(p'(g|\boldsymbol{x}))\,.$$

Note that the cross entropy loss goes back to a statistical estimation method that is called *maximum likelihood estimation*. This is the dominant method that is used do parameter estimation for logistic regression. To keep the discussion on logistic regression (rather) short, we do not intend to discuss the idea behind the use of maximum likelihood estimation here, but simply accept that the loss function for logistic regression is the cross entropy loss and that it is applied to the class posterior.

The following theorem indicates, how to carry out training for logistic regression.

[3] Still, logistic regression is a linear classifier, as the decision boundary is the solution of the system of linear equations $\frac{\exp(s_g(\boldsymbol{x}))}{\sum_{h=1}^{r}\exp(s_h(\boldsymbol{x}))} = \frac{\exp(s_{g'}(\boldsymbol{x}))}{\sum_{h=1}^{r}\exp(s_h(\boldsymbol{x}))} \Leftrightarrow \exp(s_g(\boldsymbol{x})) = \exp(s_{g'}(\boldsymbol{x})) \Leftrightarrow s_g(\boldsymbol{x}) = s_{g'}(\boldsymbol{x})$.

[4] Note that the "$\cdot$" stands for the fact that the specific choice of $g$ is actually made by the loss function $L_{CE}$.

**Theorem 8.4** With the setting of Method 12, a given training set $\{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$ and the use of the cross entropy loss, the functional that we need to minimize to compute the coefficients $\mathcal{B}$ is given by

$$\begin{aligned} J(\mathcal{B}) &= \sum_{i=1}^{N} L_{CE}(p(\cdot|\boldsymbol{x}_i), p_{\mathcal{B}}(\cdot|\boldsymbol{x}_i)) \\ &= -\sum_{i=1}^{N}\sum_{g=1}^{r} p(g|\boldsymbol{x}_i)\log(p_{\mathcal{B}}(g|\boldsymbol{x}_i)) \\ &= -\sum_{i=1}^{N}\left[\left(\sum_{g=1}^{r} p(g|\boldsymbol{x}_i)s_g(\boldsymbol{x})\right) - \log\left(\sum_{h=1}^{r}\exp(s_h(\boldsymbol{x}))\right)\right] \end{aligned}$$

We skipt the proof, here.[5)] Note that the exact class posterior $p(g|\boldsymbol{x})$ for a given training sample is simply

$$p(g|\boldsymbol{x}_i) = \begin{cases} 1 & \text{if } g = g_i \\ 0 & \text{if } g \neq g_i \end{cases},$$

as we assume that the training samples give exact information.

To minimize the above derived functional, and thereby to train a logistic regression model, we can use stochastic gradient descent (or any other gradient descent method). As a hint towards the necessary gradient calculation for the functional, we note

**Lemma 8.2** With the setting from Theorem 8.4, the gradient of the functional $J(\mathcal{B})$ with respect to a fixed class label $g$, which we call "$\nabla_g$", is given by

$$\nabla_g J(\mathcal{B}) = \begin{pmatrix} \frac{\partial}{\partial\beta_0^{(g)}} J(\mathcal{B}) \\ \vdots \\ \frac{\partial}{\partial\beta_D^{(g)}} J(\mathcal{B}) \end{pmatrix} = \frac{1}{N}\sum_{i=1}^{N}(p_{\mathcal{B}}(g|\boldsymbol{x}_i) - p(g|\boldsymbol{x}_i)) \begin{pmatrix} 1 \\ \boldsymbol{x}_i \end{pmatrix}.$$

We skipt the proof, here.[6)] The full gradient $\nabla_{\mathcal{B}}$ is obtained by "stacking on top of each other" the above gradients for all $g = 1, \ldots, r$. This leads to the following (high-level) algorithm for the training of logistic regression.

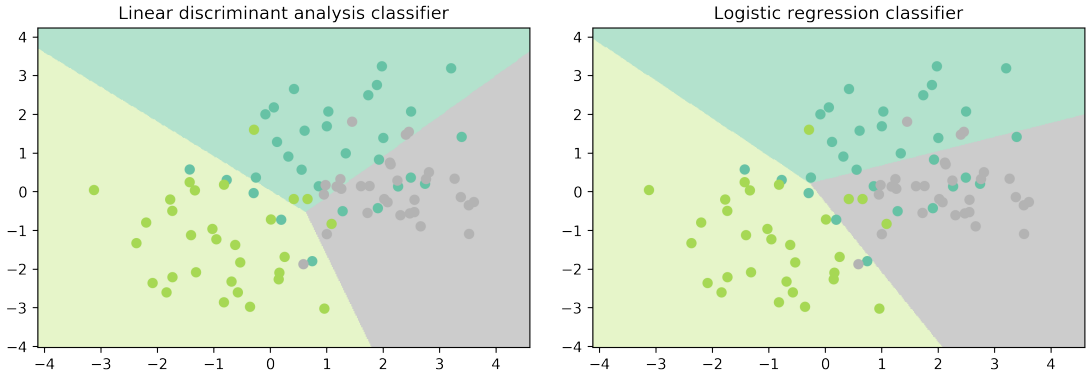
[5)] It will be added to the lecture notes at a later stage.
[6)] It will be added to the lecture notes at a later stage.

**Algorithm 13** (Logistic regression)
**input:** $\{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$ training data ($g_i \in \{1, \dots, r\}$), $\boldsymbol{x}$ evaluation point
**output:** logistic regression prediction $\hat{g}$

1. build the models $s_g(\boldsymbol{x})$ for $g = 1, \dots, r$
2. derive the predictor models $p_{\mathcal{B}}(g|\boldsymbol{x})$ for $g = 1, \dots, r$
3. $p(g|\boldsymbol{x}_i) \leftarrow \begin{cases} 1 & \text{if } g = g_i \\ 0 & \text{if } g \neq g_i \end{cases}$, for $g = 1, \dots, r$
4. compute the gradient $\nabla_{\mathcal{B}} J(\mathcal{B})$
5. use some gradient descent method to find $\mathcal{B}$ and thereby $p_{\mathcal{B}}(g|\boldsymbol{x})$ by minimizing $J(\mathcal{B})$
6. $\hat{g} \approx \arg\max_{g \in R_G} p_{\mathcal{B}}(g|\boldsymbol{x}_i)$
7. return $\hat{g}$

We continue our earlier Example 8.1 now with logistic regression.

**Example 8.6** With the same data as in Example 8.1 we compute a classifier with logistic regression by using stochastic gradient descent with learning rate $\eta = 0.0001$ to minimize the functional $J(\mathcal{B})$. Below, we give both, the classifier obtained by logistic regression and the one obtained by linear discriminant analysis.




This example including the shown figures can be reproduced and modified here: launch binder. △

We conclude by briefly comparing both classifiers in

**Remark** (LDA vs. logistic regression) Even though we do not want to give a detailed analysis of the differences between LDA and logicstic regression, we still want to try to give some general idea. In fact, the major difference between both methods is that LDA makes stronger assumptions on the data, while logistic regression does not make such assumptions. This in some sense allows us to get a more precise prediction of the

model parameters in LDA. However, at the same time, the model-intrinsic stronger bias (i.e. the stronger assumptions) might increase the overall generalization error.

In practice, as often in supervised machine learning, the best way to assess the performance of different models is to indeed try both models and check the training and genearlization error by methods introduced in Chapter 6. △

# 9. Unsupervised learning

We started the part on traditional learning techniques by clarifying that we have two big machine learning task classes: *supervised* and *unsupervised machine learning*. In supervised machine learning, we assume to have inputs $\boldsymbol{X}$ and outputs $\boldsymbol{Y}$ (or $\boldsymbol{G}$) and try to find the hidden relationship between inputs and outputs. In unsupervised learning, we are only given observations $\{\boldsymbol{x}_i\}_{i=1}^N$ from inputs $\boldsymbol{X}$ and try to gather knowledge on the inputs. In that sense, unsupervised machine learning has a strongly different objective and therefore also needs different techniques to achieve these objectives.

In this chapter, we would like to give a brief insight into unsupervised machine learning techniques by picking two examplary unsupervised machine learning tasks that everyone should be familiar with. The first task is *cluster analysis*, in which we try to group or segment samples $\{\boldsymbol{x}_i\}_{i=1}^N$ into different *clusters*, i.e. subsets. In the second task of *principle component analysis*, we derive a model for the input data. As part of this process, we can compute *principle components*, which give insight into the structure of the data and can be used in various applications.

While there exist related techniques in both fields that would allow for the treatment of the data $\{\boldsymbol{x}_i\}_{i=1}^N$ as samples from an input random vector $\boldsymbol{X}$, we, as a further simplification, limit ourselves to only consider techniques that treat the data $\{\boldsymbol{x}_i\}_{i=1}^N$ as deterministically given information, according to the remark on "Deterministic interpretation of inputs/outputs" in Section 2.3. Hence, this chapter will not deal with random variables but only with fixed, given data.

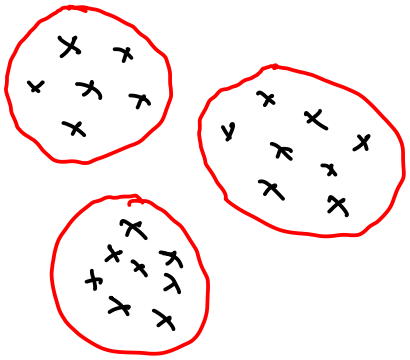
## 9.1 Cluster analysis

The objective of cluster analysis is (vaguely) characterized by

**Definition 9.1** (Cluster analysis) In **cluster analysis** or **clustering**, we group samples $\{\boldsymbol{x}_i\}_{i=1}^N$ into disjoint subsets, called **clusters**, such that all members of one cluster are "more related" each other than to members of other clusters.

We try to give a visual idea of this concept by

**Example 9.1** We consider two-dimensional observations $\{\boldsymbol{x}_i\}_{i=1}^N$, $\boldsymbol{x}_i \in \mathbb{R}^2$ and assume that two observations are "more related" to each other, if they are closer in terms of their Euclidean distance. In that case, a meaningful subdivision into clusters (circled groups) for some exemplary data (black crosses) is given below.



△

A two-dimensional example, like the above one, certainly does not expose the full complexity of the task, yet. However, if the data has thousands or even more dimensions, as we have it in real applications, clustering is certainly not a trivial task.

Our vague definition above contained the even more vague statement of samples that are "more related" to samples in the same cluster. We would like to make this notion more precise by giving the *dissimilarity* as a measure for "less related" in

**Definition 9.2** Consider data $\{\boldsymbol{x}_i\}_{i=1}^N$, $\boldsymbol{x}_i \in \mathbb{R}^D$. A **dissimilarity** for such data is a function

$$d : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}_{\geq 0}$$

that measures the difference / lack of affinity between two data samples $\boldsymbol{x}$, $\boldsymbol{x}'$. For a dissimilarity $d$, it holds at least that

- $d(\boldsymbol{x}, \boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \mathbb{R}^D$,
- $d(\boldsymbol{x}, \boldsymbol{x}') = d(\boldsymbol{x}', \boldsymbol{x})$ for all $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^D$, and
- $d$ is monotonely decreasing for growing "similarity" between two elements $\boldsymbol{x}$ and $\boldsymbol{x}'$.

We give examples of dissimilarity measures in

**Definition 9.3** For $\boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^D$, the **squared Euclidean distance** is given by

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j=1}^{D} (x_j - x'_j)^2 = \|\boldsymbol{x} - \boldsymbol{x}'\|_2^2 .$$

Furthermore, the **element-wise distance** is given by

$$d(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j=1}^{D} |x_j - x'_j| = \|\boldsymbol{x} - \boldsymbol{x}'\|_1 .$$

Finally, the **weighted squared Euclidean distance** is given by

$$d_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{x}') = \sum_{j=1}^{D} w_j (x_j - x'_j)^2 .$$

Noting that there are various ways to carry out a cluster analysis, we focus on *combinatorial clustering*.

**Definition 9.4** Let samples $\{x_i\}_{i=1}^N$ be given. In **combinatorial clustering**, we fix a number of $K < N$ clusters and look for a cluster assignment

$$C : \{1, \ldots, N\} \to \{1, \ldots, K\}$$

that assigns (the index of) each sample $\boldsymbol{x}_i$ a label or cluster

$$i \mapsto C(i)$$

such that the cost functional

$$J(C) = \frac{1}{2} \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(i')=k} d(x_i, x_{i'}) .$$

is minimized. Hence, we look for the optimal cluster assignment $C^*$

$$C^* = \arg\min_{C} J(C) .$$

The name "combinatorial clustering" comes from the fact that the search for the best possible cluster assignment is a combinatorial optimization problem. The given cost functional[1)] implements the idea that all samples within a given cluster should have a minimal mutual dissimilarity. Note further that the "sum over $C(i) = k$" is an abbreviation for a sum over the index set $\{i \in \{1, \ldots, N\} | C(i) = k\}$.

[1)] It is also possible to use a different cost functional in combinatorial clustering. However the given one is the most common one.

As for large number of samples, i.e. large $N$, and even a moderate number of clusters $K$, the search for an exact solution of the above optimization problem becomes intractable, we instead use *iterative greedy descent* to approximatively search for the cluster assignment $C^*$.

**Remark** (Iterative greedy descent) In *iterative greedy descent* for the solution of the minimization problem

$$C^* = \arg\min_C J(C)\,,$$

we start with an initial guess for $C$ and then iteratively try to improve $C$. Finally, we stop, if no further improvement is found.

Algorithms that implement this general idea are usually much faster than the exact combinatorial optimization. However, similar to gradient descent, this iterative greedy descent only allows to find local minima, which might be much worse than the actual global minimum. △

In the following we derive an iterative greedy descent method, namely the well-know *K-means clustering* algorithm, to solve the above optimization problem for the special case of the squared Euclidean distance, i.e. for the optimization problem

$$C^* = \arg\min_C \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\sum_{C(i')=k}\|\boldsymbol{x}_i - \boldsymbol{x}_{i'}\|_2^2\,.$$

An important step towards finding this iterative greedy descent method is

**Lemma 9.1** In the setting of Definition 9.4 and for squared Euclidean distance dissimilarity, i.e. $d(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|^2$, we have

$$J(C) = \sum_{k=1}^{K} N_k \sum_{C(i)=k}\|\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k\|_2^2\,,$$

where $N_k$ is the number of samples in the $k$th cluster and $\bar{\boldsymbol{x}}_k$ is the centroid of the $k$th cluster given by $\bar{\boldsymbol{x}}_k := \frac{1}{N_k}\sum_{C(i)=k}\boldsymbol{x}_i$.

Proof

We start from the functional

$$J(C) = \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\sum_{C(j)=k}\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 \tag{9.1}$$

and apply the addition of a "zero" and basic transformations to the squared norm term

by

$$\begin{aligned}\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 &= \|(\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k) - (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k)\|_2^2 \\ &= ((\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k) - (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k))^\top ((\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k) - (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k)) \\ &= (\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k)^\top (\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k) - 2(\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) + (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) \\ &= \|\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k\|_2^2 - 2(\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) + \|\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k\|_2^2 .\end{aligned}$$

Entering the just received result in (9.1) gives

$$\begin{aligned}J(C) =& \frac{1}{2} \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(j)=k} \left( \|\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k\|_2^2 + \|\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k\|_2^2 \right) \\ & - \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(j)=k} (\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) .\end{aligned}$$

We further work on the bottom term by

$$\begin{aligned}& - \sum_{k=1}^{K} \sum_{C(i)=k} \sum_{C(j)=k} (\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) \\ =& - \sum_{k=1}^{K} \sum_{C(j)=k} \left( \sum\nolimits_{C(i)=k} (\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k) \right)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) \\ =& - \sum_{k=1}^{K} \sum_{C(j)=k} \left( \left( \sum\nolimits_{C(i)=k} \boldsymbol{x}_i \right) - N_k \bar{\boldsymbol{x}}_k \right)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) \\ =& - \sum_{k=1}^{K} \sum_{C(j)=k} (N_k \bar{\boldsymbol{x}}_k - N_k \bar{\boldsymbol{x}}_k)^\top (\boldsymbol{x}_j - \bar{\boldsymbol{x}}_k) \\ =& 0 .\end{aligned}$$

The only non-trivial transformation that we carry out is between line three and four with

$$\sum_{C(i)=k} \boldsymbol{x}_i = N_k \frac{1}{N_k} \sum_{C(i)=k} \boldsymbol{x}_i = N_k \bar{\boldsymbol{x}}_k ,$$

which simply uses the definition of the centroid $\bar{\boldsymbol{x}}_k$. Hence, we continue with

$$\begin{aligned}
J(C) &= \frac{1}{2}\sum_{k=1}^{K}\sum_{C(i)=k}\sum_{C(j)=k}\left(\|\boldsymbol{x}_i-\bar{\boldsymbol{x}}_k\|_2^2+\|\boldsymbol{x}_j-\bar{\boldsymbol{x}}_k\|_2^2\right)\\
&= \frac{1}{2}\sum_{k=1}^{K}\left(\sum_{C(i)=k}\sum_{C(j)=k}\|\boldsymbol{x}_i-\bar{\boldsymbol{x}}_k\|_2^2+\sum_{C(i)=k}\sum_{C(j)=k}\|\boldsymbol{x}_j-\bar{\boldsymbol{x}}_k\|_2^2\right)\\
&= \frac{1}{2}\sum_{k=1}^{K}\left(N_k\sum_{C(i)=k}\|\boldsymbol{x}_i-\bar{\boldsymbol{x}}_k\|_2^2+N_k\sum_{C(j)=k}\|\boldsymbol{x}_j-\bar{\boldsymbol{x}}_k\|_2^2\right)\\
&= \sum_{k=1}^{K}N_k\sum_{C(i)=k}\|\boldsymbol{x}_i-\bar{\boldsymbol{x}}_k\|_2^2\,,
\end{aligned}$$

which establishes the statement of the lemma. □

The important contribution of the lemma is that we can rewrite our minimization problem as

$$C^* = \arg\min_{C}\sum_{k=1}^{K}N_k\sum_{C(i)=k}\|\boldsymbol{x}_i-\bar{\boldsymbol{x}}_k\|_2^2\,,$$

which exposes some more structure in the minimization process. Indeed, we are now able to express the functional in terms of the centroids of each cluster and the distance of the different cluster members to the cluster centroids.

In order to finally derive $K$-means clustering, we use a small trick. Instead of the cost functional $J(C) = \arg\min_C \sum_{k=1}^{K} N_k \sum_{C(i)=k} \|\boldsymbol{x}_i - \bar{\boldsymbol{x}}_k\|_2^2$ we introduce the *relaxed* cost functional

$$J'\left(C, \{\boldsymbol{m}_k\}_{k=1}^{K}\right) = \sum_{k=1}^{K}N_k\sum_{C(i)=k}\|\boldsymbol{x}_i-\boldsymbol{m}_k\|_2^2\,,$$

in which we not only allow for a changing cluster assignment but also for changing locations $\{\boldsymbol{m}\}_{k=1}^{K}$, which replace the original centroids. Then, we seek to minimize the functional both, over the cluster assignment and over these new locations.

Why can we do this relaxation, without sacrificing the correctness of the obtained optimal cluster assignment? The answer lies in

**Lemma 9.2** Let $\{\boldsymbol{x}_i\}_{i=1}^{N}$, $\boldsymbol{x}_i \in \mathbb{R}^D$ the given set of data and $S$ be an arbitrary subset of that data with $\bar{\boldsymbol{x}}_S$ its centroid. It holds

$$\bar{\boldsymbol{x}}_S = \arg\min_{\boldsymbol{m}\in\mathbb{R}^D}\sum_{\boldsymbol{x}_i\in S}\|\boldsymbol{x}_i-\boldsymbol{m}\|^2\,.$$

We will not discuss the proof, here.[2)] However, the lemma basically tells us that an exact, global minimization over the $\{\boldsymbol{m}_k\}_{k=1}^K$ would anyway give as optimal choice $\boldsymbol{m}_k = \bar{\boldsymbol{x}}_k$. Therefore, the relaxation is valid and the global minimum of $J'$ is reached with $J'\left(C^*, \{\boldsymbol{m}_k\}_{k=1}^K\right)$ such that $C^* = \arg\min_C J(C)$ and $\boldsymbol{m}_k = \bar{\boldsymbol{x}}_k$, $k = 1, \ldots, K$.
The main idea of $K$-means clustering is finally summarized in

**Method 13** ($K$-means clustering) Let data $\{\boldsymbol{x}_i\}_{i=1}^N$ and a fixed number of clusters $K$ be given. In *$K$-means clustering*, we solve the optimization problem

$$\begin{aligned}(C^*, \{\boldsymbol{m}_k^*\}_{k=1}^K) &= \arg\min_{C, \{\boldsymbol{m}_k\}} J'\left(C, \{\boldsymbol{m}_k\}_{k=1}^K\right) \\ &= \arg\min_{C, \{\boldsymbol{m}_k\}} \sum_{k=1}^K N_k \sum_{C(i)=k} \|\boldsymbol{x}_i - \boldsymbol{m}_k\|_2^2 \,.\end{aligned}$$

The method becomes an iterative greedy descent, if we iteratively minimize the functional, where, in each iteration, we first minimize $J'$ with respect to $\{\boldsymbol{m}_k\}_{k=1}^K$, while keeping $C$ fixed, and then minimize $J'$ with respect to $C$, while keeping $\{\boldsymbol{m}_k\}_{k=1}^K$ fixed. By alternating the two minimizations, we finally reach a *local* minimum. △

The following algorithm implements this idea:

**Algorithm 14** ($K$-means clustering)
**input:** input data $\{(\boldsymbol{x}_i)\}_{i=1}^N$ ($\boldsymbol{x}_i \in \mathbb{R}^D$), cluster count $K$
**output:** cluster assignment $C^*$, centroids $\{\bar{\boldsymbol{x}}_k^*\}_{k=1}^K$

1. choose initial guess $C^{(0)}$
2. $s \leftarrow 0$
3. repeat:
   a) $s \leftarrow s + 1$
   b) $N_k \leftarrow |\{i | C^{(s-1)}(i) = k\}|, \quad k = 1, \ldots, K$
   c) $\boldsymbol{m}_k^{(s)} \leftarrow \frac{1}{N_k} \sum_{C^{(s-1)}(i)=k} \boldsymbol{x}_i, \quad k = 1, \ldots, K$ *(compute centroids)*
   d) $C^{(s)}(i) \leftarrow \arg\min_{1 \le k \le K} \left\|\boldsymbol{x}_i - \boldsymbol{m}_k^{(s)}\right\|^2, \quad i = 1, \ldots, N$
   *(cluster samples around closest centroid)*
   until $C^{(s)} = C^{(s-1)}$
4. $\bar{\boldsymbol{x}}_k^* \leftarrow \frac{1}{|\{i | C^{(s)}(i)=k\}|} \sum_{C^{(s)}(i)=k} \boldsymbol{x}_i, \quad k = 1, \ldots, K$ *(compute final centroids)*
5. return $C^*$, $\{\bar{\boldsymbol{x}}_k^*\}_{k=1}^K$

The algorithm has, in each iteration, a complexity of $O(K \cdot N \cdot D)$. Let us try to

[2)] The proof would simply would be carried out by computing the gradient of the to-be-minimized term and by setting that one to zero.

understand, what the algorithm does: The algorithms jointly computes the clusters and the centroids of the clusters. It starts with an initial guess for the cluster assignment. Step 3 is the main iteration. In this iteration, we always first compute in c) the centroids for the currently constructed clusters. Then, in step d), we go over all samples and assign them to the cluster with the closest centroid. Hence, consecutively, we find centroids, group around them the points, identify new centroids, etc. This is continued until there is no more change in the cluster assignment. At this point, the algorithm has found a (locally!) optimal cluster assignment.

While the idea of the algorithm should be intuitively clear, its association to Method 13 might not be as immediate. We try to outline that in

**Remark** (Connection between Method 13 and Algorithm 14) In Method 13, we stated that we would find the clustering by alternating a minimization of $J'$ with respect to the centroids $\{\boldsymbol{m}_k\}_{k=1}^K$ and with respect to the cluster assignment $C$. In step 3. c), we implement the minimization with respect to the $\{\boldsymbol{m}_k\}_{k=1}^K$. This becomes clear, if we consult the partial minimization task

$$\{\tilde{\boldsymbol{m}}_k^*\}_{k=1}^K = \arg\min_{\{\boldsymbol{m}_k\}} \sum_{k=1}^K N_k \sum_{C(i)=k} \|\boldsymbol{x}_i - \boldsymbol{m}_k\|_2^2 \,.$$

Following Lemma 9.2, the choice of the individual $\boldsymbol{m}_k$s as the centroid of the clusters minimizes the inner sum and thereby the full functional.

The second part of Method 13, namely the minimization with respect to $C$, is carried out in step 3. d). Here, we have the partial minimization task

$$\tilde{C} = \arg\min_{C} \sum_{k=1}^K N_k \sum_{C(i)=k} \|\boldsymbol{x}_i - \boldsymbol{m}_k\|_2^2 \,.$$

The choice of $\tilde{C}(i) \leftarrow \arg\min_{1\le k\le K} \left\|x_i - m_k^{(s)}\right\|^2$, $i = 1, \dots, N$ in the algorithm might not be the exact minimizer for this minimization task, however, certainly, it will lead to a greedy reduction of the cost functional, as we individually minimize the contributions of the terms $\|\boldsymbol{x}_i - \boldsymbol{m}_k\|_2^2$. Overall, the $K$-means clustering algorithm thus carries out an alternating minimization of $J'$. △

From a practical perspective, it is still not clear how to choose the number of clusters $K$ and the initial guess $C^{(0)}$ for the cluster assignment. We build up some intuition in

**Remark** (How to choose $K$, $C^{(0)}$) The choice of the number of clusters $K$ heavily depends on the use case. In some use cases, it might be pre-determined by the application, thus the cluster count is explicitly given. In other use cases, the choice of $K$ might be part of the task, i.e. $K$ is unknown and needs to be determined. For such cases, there exist a number of heuristics to find $K$, see e.g. [6, Chapter 14].

The choice of the initial guess $C^{(0)}$ for the cluster assignment is rather task-independent but still very crucial. Obviously, we can cast the question of the choice of the cluster

assignment to the question of the choice of the initial centroids. Then a typical heuristics for the choice of the centroids would iteratively add one centroid such that, given the fixed previously added centroids and the new variable center,

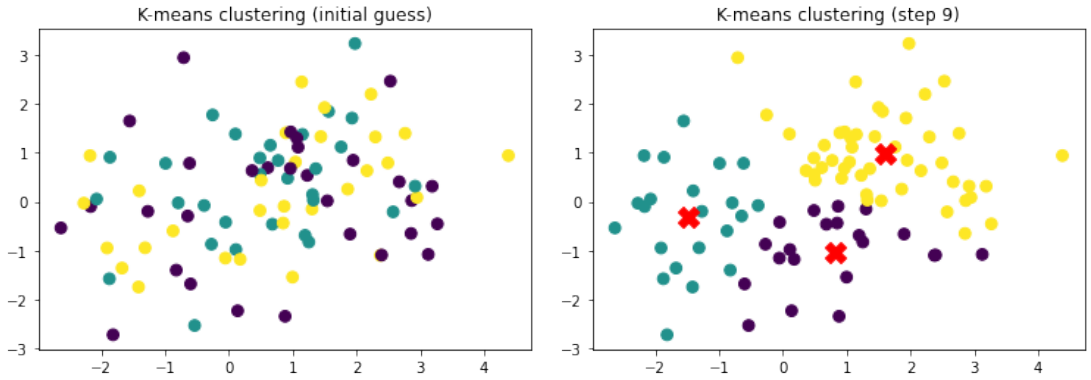
$$\sum_{k=1}^{K} N_k \sum_{C(i)=k} \|\boldsymbol{x}_i - \boldsymbol{m}_k\|_2^2$$

is further minimized. Certainly, many other heuristics exists. Moreover, one should note that $K$-means clustering only finds a *local* minimizer. Therefore, it is advisable to run $K$-means clustering with different, maybe randomized, initial guesses. Hence, one would take the "best" cluster assignment out of these runs. △

We conclude this section with

**Example 9.2** We reconsider the artificial classification training data that we used among others in Example 3.5. Instead of taking the input observations *and* the class labels, we delete the class labels and consider the question of finding a clustering for the input observations with $K = 3$ classes.

Below, we give a random initial guess for the cluster assignment on the left-hand side and the converged cluster assignment (and the centroids shown by red crosses) on the right-hand side.




This example including the shown figures and a visualization of all intermediate steps of the algorithm can be reproduced and modified here: launch binder △

## 9.2 Principal component analysis

In regression from supervised machine learning, we are interested, to find the hidden relationship between inputs $\boldsymbol{X}$ and outputs $\boldsymbol{Y}$. As stated before, in unsupervised machine learning, we have no outputs. Still, *principle component analysis* (PCA) tries to mimic this idea by finding a model for the hidden relationship *within* a given data set $\{\boldsymbol{x}_i\}_{i=1}^N$. As part of this model derivation, most important "directions", which are call *principle components*, are computed. Principal component analysis has many applications in

unsupervised learning and data analysis. Applications include, but are not limited to dimension reduction, lossy compression and feature extraction.

We immediately start with

**Method 14** (Principle component analysis) In *principle component analysis* (PCA), we are given data $\{\boldsymbol{x}_i\}_{i=1}^N$, $\boldsymbol{x}_i \in \mathbb{R}^D$, select a value $d$ with $1 \leq d \leq D$ and try to find a model $f_d$ with

$$f_d(\boldsymbol{\alpha}) = \boldsymbol{\mu} + \sum_{j=1}^{d} \alpha^{(j)} \boldsymbol{v}_j \,,$$

where $\boldsymbol{\alpha} = (\alpha^{(1)}, \ldots, \alpha^{(d)})^\top$, such that

$$\boldsymbol{x}_i \approx f_d(\boldsymbol{\alpha}_i) \qquad\qquad \text{for all } i = 1, \ldots, N$$

and $\{\boldsymbol{\alpha}_i\}_{i=1}^N$, $\boldsymbol{\alpha}_i \in \mathbb{R}^d$. Hence, we want to be able to describe each sample $\boldsymbol{x}_i$ as a linear combination of $d$ vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d \in \mathbb{R}^D$ and some offset $\boldsymbol{\mu} \in \mathbb{R}^D$. The parameter $d$ can somewhat be understood as a "quality parameter" in the sense that one can assume that for a very small $d$, the model will be rather inaccurate, while for a larger $d$ the model will more accurately describe the data.[3)]

An important observation is that, in this model, the $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d, \boldsymbol{\mu}$ parameters and even the coefficient vectors $\boldsymbol{\alpha}_i$ are all unknown and need to be measured, i.e. trained, from the data. To reduce the size of this parameter space, principle component analysis further requires that the vectors $\boldsymbol{v}_j$ are all mutually orthonormal, i.e.

$$\boldsymbol{v}_j^\top \boldsymbol{v}_{j'} = \begin{cases} 1 & \text{if } j = j' \\ 0 & \text{if } j \neq j' \end{cases} \,.$$

To briefly summarize, principal component analysis thus aims at describing the data as a linear combination of orthonormal vectors. The optimal choice for the different free parameters $\{\boldsymbol{v}_j\}_{j=1}^d, \boldsymbol{\mu}, \{\boldsymbol{\alpha}^{(i)}\}_{i=1}^N$ is then computed from the given data by a least squares estimation, hence we get the optimal choices by

$$\left(\{\widehat{\boldsymbol{v}}_j\}_{j=1}^d, \widehat{\boldsymbol{\mu}}, \{\widehat{\boldsymbol{\alpha}}_i\}_{i=1}^N\right) = \underset{\{\boldsymbol{v}_j\}_{j=1}^d, \boldsymbol{\mu}, \{\boldsymbol{\alpha}_i\}_{i=1}^N}{\arg\min} \sum_{i=1}^{N} \|\boldsymbol{x}_i - f_d(\boldsymbol{\alpha}_i)\|_2^2 \,.$$

The resulting vectors $\{\widehat{\boldsymbol{v}}_j\}_{j=1}^d$ are called *principle components*. For notational reasons, we further combine them into a single matrix $\mathcal{V}_d = (\boldsymbol{v}_1 | \cdots | \boldsymbol{v}_d)$ and thus obtain the minimization problem

$$\left(\widehat{\mathcal{V}}_d, \widehat{\boldsymbol{\mu}}, \{\widehat{\boldsymbol{\alpha}}_i\}_{i=1}^N\right) = \underset{\boldsymbol{\mu}, \{\boldsymbol{\alpha}_i\}_{i=1}^N, \mathcal{V}_d}{\arg\min} \sum_{i=1}^{N} \|\boldsymbol{x}_i - \boldsymbol{\mu} - \mathcal{V}_d \boldsymbol{\alpha}_i\|_2^2 \,.$$

△

[3)] Indeed, for $d = D$, the introduced model we exactly represents the given input data.

While the above description fully characterizes the principle component analysis, we should invest a bit more time to simplify the optimization problem. We can achieve a first simplification by

**Lemma 9.3** In the setting of Method 14, let data $\{\boldsymbol{x}_i\}_{i=1}^N$, $\boldsymbol{x}_i \in \mathbb{R}^D$ be given. We consider the minimization problem

$$\left(\widehat{\mathcal{V}}_d, \widehat{\boldsymbol{\mu}}, \{\widehat{\boldsymbol{\alpha}}_i\}_{i=1}^N\right) = \operatorname*{arg\,min}_{\boldsymbol{\mu}, \{\boldsymbol{\alpha}_i\}_{i=1}^N, \mathcal{V}_d} \sum_{i=1}^N \|\boldsymbol{x}_i - \boldsymbol{\mu} - \mathcal{V}_d \boldsymbol{\alpha}_i\|_2^2 ,$$

while assuming for $\mathcal{V}_d \in \mathbb{R}^{D\times d}$ that it holds $\widehat{\mathcal{V}}_d^\top \widehat{\mathcal{V}}_d = \mathcal{I}_d$. By partially optimizing for $\boldsymbol{\mu}$ and $\{\boldsymbol{\alpha}_i\}_{i=1}^N$, we obtain

$$\widehat{\boldsymbol{\mu}} = \frac{1}{N}\sum_{i=1}^N \boldsymbol{x}_i - \frac{1}{N}\sum_{i=1}^N \mathcal{V}_d \boldsymbol{\alpha}_i = \bar{\boldsymbol{x}} - \frac{1}{N}\sum_{i=1}^N \mathcal{V}_d \boldsymbol{\alpha}_i \quad \text{and} \quad \widehat{\boldsymbol{\alpha}}_i = \mathcal{V}_d^\top (\boldsymbol{x}_i - \boldsymbol{\mu}) .$$

To make the solution unique, we further assume $\frac{1}{N}\sum_{i=1}^N \mathcal{V}_d \boldsymbol{\alpha}_i = \boldsymbol{0}$ and obtain

$$\widehat{\boldsymbol{\mu}} = \bar{\boldsymbol{x}} \quad \text{and} \quad \widehat{\boldsymbol{\alpha}}_i = \mathcal{V}_d^\top (\boldsymbol{x}_i - \bar{\boldsymbol{x}}) .$$

We skip the proof. It would just require to set the respective gradients of the cost functional to zero. Given this lemma, we are are able to find the model

$$\hat{f}_d(\boldsymbol{\alpha}) = \bar{\boldsymbol{x}} + \sum_{j=1}^d \alpha^{(j)} \boldsymbol{v}_j = \bar{\boldsymbol{x}} + \widehat{\mathcal{V}}_d \boldsymbol{\alpha}$$

by minimizing

$$\widehat{\mathcal{V}}_d = \operatorname*{arg\,min}_{\mathcal{V}_d \in \mathbb{R}^{D\times d}, \mathcal{V}_d^\top \mathcal{V}_d = \mathcal{I}_d} \sum_{i=1}^N \|(\boldsymbol{x}_i - \bar{\boldsymbol{x}}) - \mathcal{V}_d \mathcal{V}_d^\top (\boldsymbol{x}_i - \bar{\boldsymbol{x}})\|_2^2 .$$

Hence we just "plug" the partial minimizers into the original minimization problem. A typical further simplification is to initially center the data $\{\boldsymbol{x}_i\}_{i=1}^N$. Thereby, the model $\hat{f}_d$ and the minimization problem is further simplified resulting in

**Theorem 9.1** (Principle component analysis) With the setting of Method 14 and with given, centered data $\{\boldsymbol{x}_i\}_{i=1}^N$, i.e. $\frac{1}{N}\sum_{i=1}^N \boldsymbol{x}_i = \boldsymbol{0}$, principle component analysis provides for a choice of $d$ a model

$$\hat{f}_d(\boldsymbol{\alpha}) = \widehat{\mathcal{V}}_d \boldsymbol{\alpha} ,$$

where

$$\widehat{\mathcal{V}}_d = \operatorname*{arg\,min}_{\mathcal{V}_d \in \mathbb{R}^{D\times d}, \mathcal{V}_d^\top \mathcal{V}_d = \mathcal{I}} \sum_{i=1}^N \|\boldsymbol{x}_i - \mathcal{V}_d \mathcal{V}_d^\top \boldsymbol{x}_i\|_2^2 ,$$

such that each individual sample $\boldsymbol{x}_i$ can be approximated by

$$\boldsymbol{x}_i \approx \hat{f}_d(\widehat{\boldsymbol{\alpha}}_i) = \hat{f}_d\left(\widehat{\mathcal{V}}_d^\top \boldsymbol{x}_i\right) = \widehat{\mathcal{V}}_d \widehat{\mathcal{V}}_d^\top \boldsymbol{x}_i\,.$$

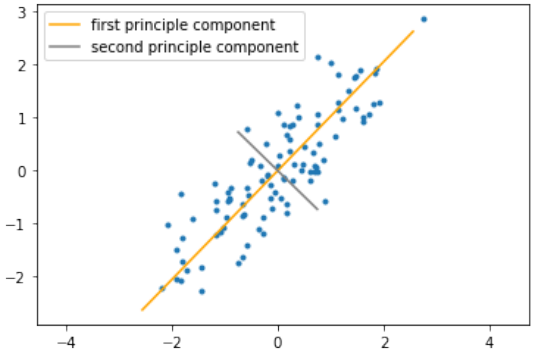
The columns $\boldsymbol{v}_j \in \mathbb{R}^D$ $(j = 1, \ldots, d)$ of $\widehat{\mathcal{V}}_d$ are the principle components.

It is more than time to give an idea of why the vectors $\boldsymbol{v}_j$ are called *principle components*. We do this via

**Example 9.3** (Principle components) Let us consider data $\{\boldsymbol{x}_i\}_{i=1}^N$, $x_i \in \mathbb{R}^2$ that we get as samples from a bivariate normal distribution

$$\mathcal{U}\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 0.8 \\ 0.8 & 1 \end{pmatrix}\right).$$

The below figure visualizes $N = 100$ such samples that are furthermore centered to $(0, 0)^\top$.




If we were to choose $d = 1$ in the principle component analysis, we would obtain the model

$$f_1(\alpha) = \alpha \boldsymbol{v}_1\,.$$

From linear algebra, we immediately see that $f_1$ describes a straight line in the two-dimensional space that goes into the direction of $\boldsymbol{v}_1$, i.e. in the direction of the first principle component. In the above picture, we show it as an orange line. Our intuitive observation is that the line somewhat describes the most important "direction" in that data set, i.e. the direction in which there is the most change in the data. Hence the first principle component describes the "most important direction" in the data. If we were to analyze the construction of the principle component analysis further, we would even be able to show that the given line minimizes the mean squared distance to all points in the data set, where the distance of a point to a line is understood as always via its orthogonal projection.

Extending the PCA to $d = 2$, gives us a model

$$f_2(\boldsymbol{\alpha}) = \alpha^{(1)}\boldsymbol{v}_1 + \alpha^{(2)}\boldsymbol{v}_2\,.$$

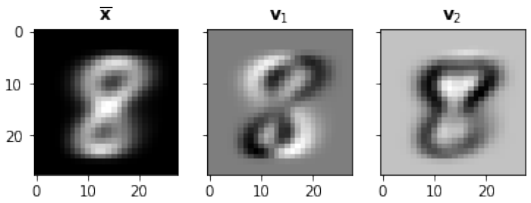
By construction of the PCA, the first principle component will be identical to the first principal component of the case of $d = 1$. The model $f_2$ is a equation describing the full two-dimensional plane, which is in our case the full domain. The second principle component $\boldsymbol{v}_2$ is, by construction, orthogonal to $\boldsymbol{v}_1$. In the above figure, the resulting line is given in grey. While this is not as clearly visible, since we work only with two-dimensional data, the second principle component gives the second most important direction in the data. If we compute, as we do it here, all principle components for a given data set, we can further interpret these principle components as vectors that span up a new coordinate system that somewhat "best fits" the data.

This example including the shown figure can be reproduced and modified here: launch binder △

The above example allows to get a first intuition of the principle components. In the following example, we get an impression, what PCA can do on real data.

**Example 9.4** (Data synthesis) We consider the MNIST data set of hand written digits from Example 2.1. In that data set, we are given greyscale images of hand written digits with $28 \times 28$ pixels, hence we have a data set $\{\boldsymbol{x}_i\}_{i=1}^N$, $\boldsymbol{x}_i \in \mathbb{R}^{784}$. We only pick those images that show the value 8. Below are four example images of this kind.


Then we center each sample by computing the average $\bar{\boldsymbol{x}}$ and subtracting that from all samples. Furthermore, we compute the first two principle components $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. The average and the first two principle components are visualized below.

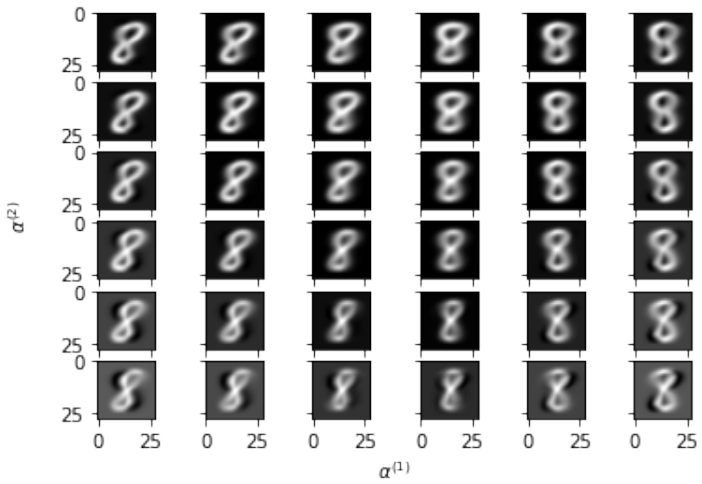



The left picture is simply the "average" hand-written 8. With a little practice, one can see that the first principle component mostly covers the orientation of the hand-written digit. Moreover, the second principle component describes the width of the digit, as we will see in a moment. What we thus learn about our data is that the most dominant differentiation between the digits is their orientation. The second most dominant differentiation is the width.

So how do we see that the two first principle components represent orientation and width? We can see this by generating new, unseen samples by evaluation the model $f_2(\boldsymbol{\alpha})$ that is provided by the PCA. If we want to get "properly looking" digit images, we need to include the mean. Thereby the model becomes

$$f_d(\boldsymbol{\alpha}) = \bar{\boldsymbol{x}} + \alpha^{(1)}\boldsymbol{v}_1 + \alpha^{(2)}\boldsymbol{v}_2\,.$$

Then we sample the model by evaluating it for changing values $\alpha^{(1)} \in [-1000, 1000]$ and $\alpha^{(2)} \in [-1000, 1000]$. The result is shown below.




Along the horizontal axis, we change the coefficient for the first principle component and, indeed, we observe that on the one extreme, the digit is tilted towards the right-hand side, while on the other end, it is slightly tilted towards the left-hand side. At the same time, the vertical axis represents changes in the coefficient for the second principle component. Here, from top to bottom, the digits get slimmer.

This example including the shown figures can be reproduced and modified here: launch binder △

What we have just seen is that PCA not only allows to characterize "main directions" but it also gives a means to generate new, unseen data that follows the characteristics for the originally given data. This is a surprisingly powerful operation.

Now that we have gained a lot of intuition on the method, we should learn how to compute the principle components. The main idea is given by

**Theorem 9.2** Let the setting of Theorem 9.1 be given. The solution $\hat{\mathcal{V}}_d$ of the optimization problem is the matrix composed of the eigenvectors for the $d$ largest eigenvalues of the matrix

$$A = \mathcal{X}^\top \mathcal{X}\,,$$

where $\mathcal{X} \in \mathbb{R}^{N\times D}$ is the matrix created from the data as $\mathcal{X} = (\boldsymbol{x}_1|\cdots|\boldsymbol{x}_N)^\top$.

To be concise, we skip the proof of this theorem and immediately discuss the efficient calculation of the principle components. Following the above theorem, we always would have to compute $\mathcal{X}^\top\mathcal{X}$, which can be extremely expensive, depending on the data set. The common practice is to avoid that by following

**Remark** (Efficient calculation of principle components) For $\mathcal{X} \in \mathbb{R}^{N\times D}$ as in the above theorem, we can compute the *singular value decomposition* (SVD)

$$\mathcal{X} = \mathcal{U}\Sigma\mathcal{V}^\top$$

with $\mathcal{U} \in \mathbb{R}^{N\times D}$, $\Sigma \in \mathbb{R}^{D\times D}$, $\mathcal{V} \in \mathbb{R}^{D\times D}$, $\mathcal{U}^\top\mathcal{U} = \mathcal{I}$, $\mathcal{V}^\top\mathcal{V} = \mathcal{I}$ and $\Sigma = \operatorname{diag}(\sigma_1, \dots, \sigma_D)$. The $\{\sigma_j\}_{j=1}^D$ are called *singular* values and are ordered by decreasing size. The columns of $\mathcal{V} = (\boldsymbol{v}_1|\cdots|\boldsymbol{v}_N)$ are called *right singular vectors*.

The important connection between the SVD and the PCA is that the right singular vectors of $\mathcal{X}$ are the principle components for the data set $\{\boldsymbol{x}_i\}_{i=1}^N$. Hence, we can immediately build $\mathcal{X}$ and compute the principle components via SVD without first building the product $\mathcal{X}^\top\mathcal{X}$ and computing eigenvectors. △

The above remark can immediately be re-cast to an algorithm for the calculation of the principle components of given data.

**Algorithm 15** (Calculation of principle components)
**input:** data set $\{(\boldsymbol{x}_i)\}_{i=1}^N$ ($\boldsymbol{x}_i \in \mathbb{R}^D$), quality parameter $d$
**output:** principle components $\{\boldsymbol{v}_j\}_{j=1}^d$, center of data $\bar{\boldsymbol{x}}$

1. $\bar{\boldsymbol{x}} \leftarrow \frac{1}{N}\sum_{i=1}^N \boldsymbol{x}_i$
2. $\mathcal{X} \leftarrow (\boldsymbol{x}_1 - \bar{\boldsymbol{x}}|\cdots|\boldsymbol{x}_N - \bar{\boldsymbol{x}})^\top$
3. $\mathcal{U}, \Sigma, \mathcal{V} \leftarrow \texttt{svd}(\mathcal{X})$
4. $(\boldsymbol{v}_1|\cdots|\boldsymbol{v}_D) \leftarrow \mathcal{V}$
5. return $\{\boldsymbol{v}_j\}_{j=1}^d$, $\bar{\boldsymbol{x}}$

This concludes our discussion of the PCA. It should, however, be noted that there is (at least) one other very prominent application of the PCA, which is *dimension reduction* or in other words *data compression*. Nevertheless, to be concise, we skip the discussion of this application.

# Part III.

# Modern and advanced learning

# 10. Advanced regression

So far, the linear model has been our only parametric model for regression tasks. In this chapter, we will extend the linear model by so-called *basis expansions* to non-linear models of arbitrary model complexity, while still using the least squares estimator. This gives rise to various new, non-linear, regression models.

The second section of this chapter is concerned with the development of a method that reduces overfitting. The new method is called *ridge regression* and can be applied at least to all parametric models whose parameters are estimated using the least squares predictor.

In this chapter, we will limit ourselves to the case of one-dimensional outputs $Y$, i.e. $K = 1$.

## 10.1 Basis expansion models

In linear regression by least squares, we use a linear model to approximate the regressor as

$$\mathrm{E}\left(Y|\boldsymbol{x}\right) \approx f_{\boldsymbol{\beta}} = \beta_0 + \sum_{j=1}^{D} \beta_j X_j$$

and estimate the optimal coefficient vector by least squares, thus

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{N} L_2(y_i, f_{\boldsymbol{\beta}}(\boldsymbol{x}_i)) \,.$$

If we choose the linear model, we make the very strong assumption that the hidden relationship between the inputs and the outputs is linear. Certainly, for many problems, this assumption is too strong, hence the predictor built by the linear model might have a very strong bias.

To overcome this limitation, we introduce (basis) functions

$$\varphi_m : \mathbb{R}^D \to \mathbb{R}, \quad m = 1, \ldots, M \,,$$

which we understand as (non-linear) transformations of the $N$ inputs $\boldsymbol{X} = (X_1, \ldots, X_N)^\top$ to new $M$ inputs $(\varphi_1(\boldsymbol{X}), \ldots, \varphi_M(\boldsymbol{X}))^\top$. This gives rise to

**Definition 10.1** (Basis expansion model) Let $\boldsymbol{X}$ be a $D$-dimensional input vector and $Y$ be a 1-dimensional quantitative output. Let further a sequence of basis functions $\{\varphi_j\}_{j=1}^M$, $\varphi_m : \mathbb{R}^D \to \mathbb{R}$ be given. The **basis expansion model** is an approximation for the regressor $\mathrm{E}(Y|\boldsymbol{X} = \boldsymbol{x})$ of the form

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{M} \alpha_j \varphi_j(\boldsymbol{X}),$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_N)^\top \in \mathbb{R}^M$ is a vector of unknown parameters or coefficients.

In the basis expansion model, we usually do not explicitly add an intercept, as in the linear model. Instead, we can add it to the expansion by choosing one of the basis functions as $\varphi_j(\boldsymbol{X}) \equiv 1$. Why do we call the $\varphi_j$ *basis* functions?

**Remark** (Basis functions) The notion "basis function" indeed comes from the idea that the $\{\varphi_j\}_{j=1}^M$ form the basis of some vector space over functions. The mathematically interested reader might have seen the concept of a vector space over functions in linear algebra for the example of polynomials. While the terminology is not that important, it is important that the $\{\varphi_j\}_{j=1}^M$ are linear independent, which we need that to make sure that the newly generated inputs are uncorrelated. In other words, we want to make sure that each new input gives unique information. △

Given the new basis expansion model, we can do the training, i.e. the parameter estimation, as before, by the least squares estimator, hence do

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \sum_{i=1}^{N} L_2(y_i, f_{\boldsymbol{\alpha}}(\boldsymbol{x}_i)).$$

We immediately obtain

**Theorem 10.1** (Least squares regression for basis expansion model) Let $\boldsymbol{X}$, $Y$, $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ be given as usual. Moreover, we are given a set of basis functions $\{\varphi_j\}_{j=1}^M$ that lead to the model

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{M} \alpha_j \varphi_j(\boldsymbol{X}).$$

We consider the matrix $\mathcal{X} \in \mathbb{R}^{N \times M}$

$$\mathcal{X} = \begin{pmatrix} \varphi_1(\boldsymbol{x}_1) & \cdots & \varphi_M(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ \varphi_1(\boldsymbol{x}_N) & \cdots & \varphi_M(\boldsymbol{x}_N) \end{pmatrix}$$

Let us further assume that $\mathcal{X}$ has full column rank. Then, the least squares estimator $\hat{\boldsymbol{\alpha}}$ for the parameters $\boldsymbol{\alpha}$ in the model $f_{\boldsymbol{\alpha}}$ is uniquely given by

$$\hat{\boldsymbol{\alpha}} = \left(\mathcal{X}^\top \mathcal{X}\right)^{-1} \mathcal{X}^\top \boldsymbol{y}\,,$$

where $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$.

The proof is almost identical to the proof of Theorem 4.1 and is therefore skipped here. We immediately come to a few examples.

**Example 10.1** (Linear regression via basis expansion) By choosing $M = D + 1$ and

$$\varphi_1(\boldsymbol{X}) = 1, \varphi_2(\boldsymbol{X}) = X_1, \ldots, \varphi_{D+1}(\boldsymbol{X}) = X_D\,,$$

we get the linear model and Theorem 10.1 becomes identical to Theorem 4.1, as we have

$$\mathcal{X} = \begin{pmatrix} \varphi_1(\boldsymbol{x}_1) & \varphi_2(\boldsymbol{x}_2) & \cdots & \varphi_M(\boldsymbol{x}_1) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_1(\boldsymbol{x}_N) & \varphi_2(\boldsymbol{x}_N) & \cdots & \varphi_M(\boldsymbol{x}_N) \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1D} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{ND} \end{pmatrix},$$

where $(1, x_{i1}, \ldots, x_{iD})^\top = \boldsymbol{x}_i$. △

Next, we consider quadratic polynomials as basis expansion.

**Example 10.2** (Quadratic polynomial regression) For simplicity, we stick to the case of $D = 1$, i.e. one-dimensional input, in this example. Instead of using a linear function as model, we use the basis expansion method to construct a quadratic polynomial as model. To this end, we choose the basis functions

$$\varphi_1(X) = 1, \varphi_2(X) = X, \varphi_3(X) = X^2\,,$$

thus we obtain the model

$$f_{\boldsymbol{\alpha}}(X) = \alpha_1 + \alpha_2 X + \alpha_3 X^2\,.$$

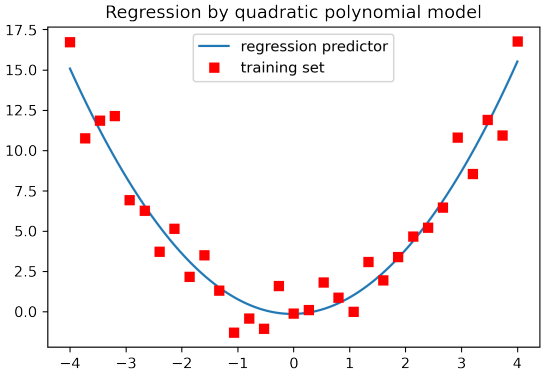
As input observations we use $N = 31$ equidistant samples $x_i \in [-4, 4]$ and build the output observations with noise via

$$y_i = x_i^2 + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2), \quad i = 1, \ldots, N\,,.$$

with $\sigma = 1.5$. To fit the model to data by least squares, we build the matrix

$$\mathcal{X} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{pmatrix}$$

and obtain the coefficient vector as $\hat{\boldsymbol{\alpha}} = (\mathcal{X}^\top \mathcal{X}) \mathcal{X}^\top \boldsymbol{y}$ with $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$. The resulting model is shown below.




It should be clear that, so far, only kNN regression would have allowed us to fit a suitable model into such data. Linear regression, due to its too small model complexity, would have a very large bias.

This example including the shown figure can be reproduced and modified here: launch binder △

Certainly, we are not just limited to a quadratic model with one-dimensional input.

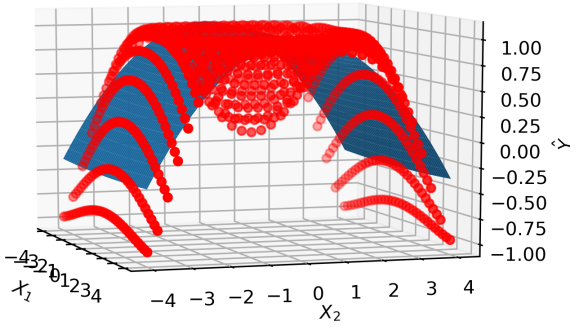
**Example 10.3** (Higher-degree polynomial for higher-dimensional inputs) When extending the quadratic polynomial to $D$-dimensional inputs, obtain $O(D^2)$ basis functions, which are given by

$$\{\varphi_m\}_{m=1}^{M} = \{1, X_1, \ldots, X_D, X_1^2 \ldots, X_D, X_1X_2, \ldots, X_1X_D, X_2X_3, \ldots, X_2X_D, \ldots\}\,.$$

Let us consider training data that we obtain by sampling $N = 31 \cdot 31$ input samples in an equidistant manner on the domain $[-4, 4] \times [-4, 4]$ and generating the outputs via the function

$$f(\boldsymbol{X}) = \sin\left(((0.5X_1)^2 + (0.5X_2)^4)^{1/2}\right)$$

without additional noise. The quadratic model for $D = 2$ has 6 basis functions and the resulting predictor is given below.




It does not very well fit into the data. Specifically the "basin" in the center is not captured. A very natural idea might be to increase the model complexity in order

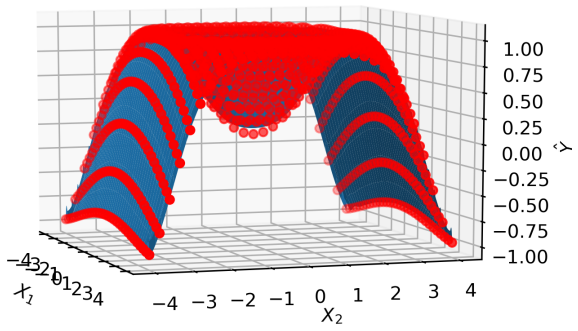
to better fit the data. In case of polynomials, we can do that by choosing a higher polynomial degree.

Let us exemplify that for one-dimensional input and general polynomial degree $p$. In this case we would need to choose the basis functions

$$\varphi_1(X) = 1, \varphi_2(X) = X, \varphi_3(X) = X^2, \ldots, \varphi_{p+1}(X) = X^p \,.$$

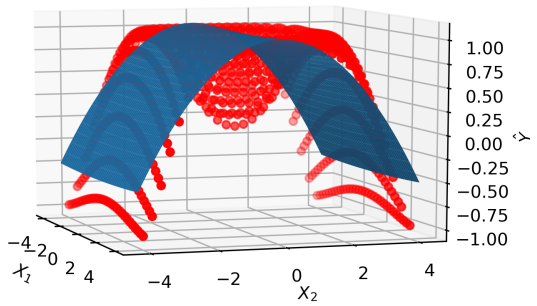
For a general polynomial degree of $p$ and a general dimension $D$, we would finally require $O(D^p)$ basis functions, hence we have an exponential growth in the number of basis functions for growing $p$.

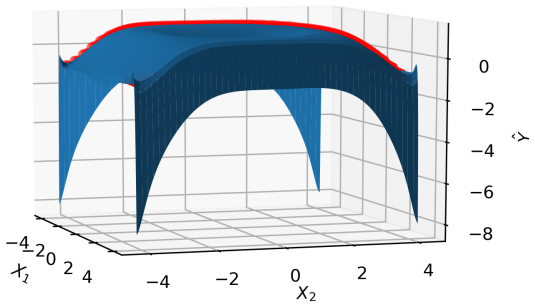
Coming back to our example on concrete data, we now choose $p = 20$ and obtain the below predictor:




Obviously, the higher model complexity reduces the bias in the data and we get a very good fit.

Nevertheless, the high model complexity first comes with a high computational effort, as we require in the two-dimensional case and for $p = 20$ a total of $M = 231$ basis functions. Moreover, we observe bad generalization properties outside the domain of the given training data. To show this, we evaluate both predictors ($p = 2, 20$) again, however this time on the slightly larger domain $[-4.2, 4.2] \times [-4.2, 4.2]$. Below, we see the polynomial of degree $p = 2$ on the left-hand side and the polynomial of degree $p = 20$ on the right-hand side.



Apparently, the quadratic polynomial model has no high variation outside of the domain of the training samples, thus generalizes well. On the other hand, the polynomial of degree $p = 20$ shows a strong drop into the negative direction, noting the scale of the vertical axis. Hence, generalization is very bad outside of the domain covered by the training data.

This example including the shown figures can be reproduced and modified here: launch binder △

The just discussed example is yet another example for the difficulties of the bias-variance trade-off. Moreover, we notice that even if we would accept the issue of the generalization error, still, increasing the model complexity of a polynomial model goes together with an exponential increase of the number of basis functions. While this is still manageable for a two dimensional input and degree $p = 20$ with a total of $M = 231$ basis functions, already an input dimensionality of $D = 8$ and $p = 20$ leads to about 3.1 million(!) basis functions. Recalling that nowaday's machine learning problems can easily have hundreds, thousands or even millions of input dimensions, this clearly becomes intractable.

To overcome that issue, we consider *kernel-based models*. We start with the *Gaussian kernel*.

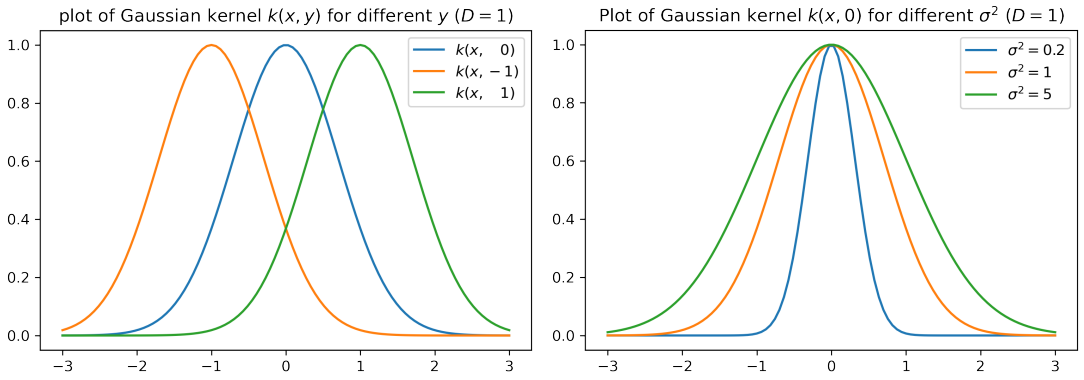
**Definition 10.2** The function $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ with

$$k(\boldsymbol{x}, \boldsymbol{x}') := e^{-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|_2^2}{\sigma^2}}$$

is called **Gaussian kernel** and $\sigma^2$ is a scalar parameter, i.e. the variance or **kernel width**.

A visual interpretation is given in

**Example 10.4** (Gaussian kernel) For $D = 1$, we give below the Gaussian kernel for different parameter settings.




On the left-hand side, we plot the kernel with respect to its first input parameter and

fix three different choices of the second parameter. By changing this second parameter, we shift the function to a different center. On the right-hand side, we fix the center of the kernel while changing the kernel width. As the name suggests, we change the width of the kernel. △

We will immediately come to the idea, how to use a kernel to build a set of basis functions. However, before that, we briefly generalize from the specific choice of the Gaussian kernel to

**Definition 10.3** A continuous bivariate function $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ is called (symmetric) **positive definite kernel** on $\mathbb{R}^D$, if for all $N \in \mathbb{N}$, all finite subsets $\{x_1, \dots, x_N\} \subseteq \mathbb{R}^D$, and all $\boldsymbol{\alpha} \in \mathbb{R}^N \setminus \{\mathbf{0}\}$, we have

$$\sum_{j=1}^{N} \sum_{l=1}^{N} \alpha_j \alpha_l k(\boldsymbol{x}_j, \boldsymbol{x}_l) > 0 \,.$$

It is possible to show

**Lemma 10.1** The Gaussian kernel is positive definite.

Now we combine positive definite kernels and the idea of basis expansion to introduce *kernel-based regression*.

**Method 15** (Kernel-based regression) Let the training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ and a positive definite kernel function $k$ be given. We use the basis expansion model and select as basis functions

$$\varphi_m(\boldsymbol{X}) = k(\boldsymbol{X}, \boldsymbol{x}_j) \qquad\qquad j = 1, \dots, N \,.$$

Hence, we introduce one basis function for each training sample, thus $M = N$. Each of the basis functions is the kernel function centered in another point, namely the training samples. The resulting model is

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}, \boldsymbol{x}_i) \,.$$

△

An interesting observation for the kernel-based model is that it's model complexity, hence the number of basis functions, grows with the number of training samples. Indeed, this is often an approach that much better reflects the nature of the training data than

approches that use a fixed model complexity. Trying to compare the kernel-based model to earlier models, it somewhat combines the properties of locality from kNN regression with the smoothness of polynomial basis expansion models. Moreover, by construction, a growing number of input dimensions does not lead to an increase in the number of basis functions.

Due to the special structure of the kernel-based model we can simplify the usual training by a least squares fit a bit. To this end, we briefly conclude, without proof, from Definition 10.3

**Lemma 10.2** Let $\{\boldsymbol{x}_i\}_{i=1}^N$ be a set of points such that $\boldsymbol{x}_i \in \mathbb{R}^D$ and let $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ be a positive definite kernel. Then, the matrix

$$\mathcal{A} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

is (symmetric) positive definite.

We will colloquially call $\mathcal{A}$ *kernel matrix*. Then, we obtain for the computation of the least squares estimator for the coefficients $\boldsymbol{\alpha}$ by

**Theorem 10.2** (Regression for kernel-based model) Let $\boldsymbol{X}, Y, \mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ be given as usual. Moreover, we are given a positive definite kernel $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ that leads to the kernel-based model

$$f_\alpha(\boldsymbol{X}) = \sum_{j=1}^{N} \alpha_j k(\boldsymbol{X}, \boldsymbol{x}_j) \,.$$

With the kernel matrix $\mathcal{A}$ from Lemma 10.3, the least squares estimator $\hat{\boldsymbol{\alpha}}$ for the parameters $\boldsymbol{\alpha}$ in the model $f_{\boldsymbol{\alpha}}$ is uniquely given by

$$\hat{\boldsymbol{\alpha}} = \mathcal{A}^{-1}\boldsymbol{y} \,,$$

where $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$.

Proof

In Theorem 10.1, we choose for the basis functions $\{\varphi_j\}_{j=1}^M$ $M = N$ and

$$\varphi_j(\boldsymbol{X}) = k(\boldsymbol{X}, \boldsymbol{x}_j) \qquad\qquad j = 1, \ldots, N \,.$$

Then, we immediately observe $\mathcal{X} = \mathcal{A}$. Following Lemma 10.2, $\mathcal{A}$ is further symmetric, positive definite and we thus have that $\mathcal{A} = \mathcal{A}^\top$, $\mathcal{A}$ has full column rank and $\mathcal{A}$ is invertible.

According to Theorem 10.1 we further have

$$\hat{\boldsymbol{\alpha}} = (\mathcal{X}^\top \mathcal{X})^{-1} \mathcal{X}^\top \boldsymbol{y} = (\mathcal{A}^\top \mathcal{A})^{-1} \mathcal{A}^\top \boldsymbol{y} = \mathcal{A}^{-1} (\mathcal{A}^\top)^{-1} \mathcal{A}^\top \boldsymbol{y} = \mathcal{A}^{-1} \boldsymbol{y} \,.$$

□

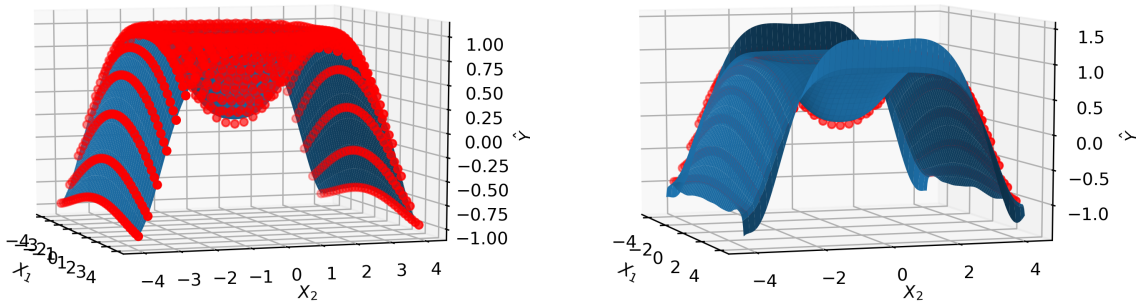
This gives rise to the following algorithm to carry out regression with the kernel-based model.

**Algorithm 16** (Regression for kernel-based model)
**input:** $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ training data, $\boldsymbol{x}$ evaluation point, $k$ positive definite kernel
**output:** kernel-based regression prediction $\hat{Y}(x)$

1. build matrix $\mathcal{A}$
2. solve $\mathcal{A}\hat{\boldsymbol{\alpha}} = \boldsymbol{y}$ using Cholesky factorization
3. $\hat{Y}(\boldsymbol{x}) = \sum_{j=1}^N \hat{\alpha}_j k(\boldsymbol{x}, \boldsymbol{x}_j)$
4. return $\hat{Y}(\boldsymbol{x})$

The complexity of the algorithm is $O(N^3 + D \cdot N^2)$. We conclude by

**Example 10.5** (Kernel-based model) We continue with the same training data as in Example 10.3 and build, using the Gaussian kernel, a kernel-based model. Note that, in principle, one might want to choose the kernel width by cross validation. However, we just select $\sigma^2 = 0.15$ and obtain the following model:




On the left-hand side, we see the model evaluated on the domain $[-4, 4] \times [-4, 4]$. It very nicely fits into the given data. On the right-hand side, we repeat the study of Example 10.3, where we have analyzed the generalization ability of the polynomial model. Again, we evaluate the model outside of the training data on the domain $[-4.2, 4.2] \times [-4.2, 4.2]$. Obviously, still, some overfitting happens. However, it is less strong than in the high-degree polynomial case.

This example including the shown figure can be reproduced and modified here: launch binder △

Note that we did not argue that the kernel-based model would fix the large generalization error of the polynomial model but the problem for high-dimensional inputs. This is still true. Moreover, we will introduce a technique to deal with the overfitting problem in the next section.

## 10.2 Ridge regression

In Theorems 4.1 and 10.1 for the estimation of the coefficients in the linear or basis expansion model, we had the side condition that the matrix $\mathcal{X}$ would have to have full column rank such that the model can be uniquely solved. In the basis expansion model setting, this issue shows up, if we have less training samples than basis functions, thus we are in the *underdetermined* setting (i.e. less training data than coefficients). However, even if we have enough training data, depending on the basis, $\mathcal{X}$ could be numerically closer to a non-full column rank. This is known to happen, e.g., for the kernel-based model and large $N$.

All the above described cases lead to a situation, where, in principle, we can no longer uniquely solve the linear system of equations. However, this issue can be overcome by introducing what is often called a *regularization*, hence an additional condition in the minimization problem that we solve in order estimate the coefficients in the model. A specific form of regularization, the *Tikhonov regularization* leads to what is called *ridge regression*.[1)]

**Definition 10.4** (Ridge regression) Let $\boldsymbol{X}$, $Y$, $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ be given as usual. Moreover, we are given a set of basis functions $\{\varphi_j\}_{j=1}^M$ that lead to the model

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{M} \alpha_j \varphi_j(\boldsymbol{X})\,.$$

In **ridge regression**, we obtain the optimal coefficient vector $\hat{\boldsymbol{\alpha}}$ by solving the optimization problem

$$\hat{\boldsymbol{\alpha}} = \arg\min_{\boldsymbol{\alpha}} \sum_{i=i}^{N} (y_i - f_{\boldsymbol{\alpha}}(\boldsymbol{x}_i))^2 \;+\; \lambda \|\boldsymbol{\alpha}\|_2^2\,,$$

where $\lambda \in \mathbb{R}_{\geq 0}$ is a **regularization parameter**.

Trying to interpret ridge regression, we implicitly limit the size of the coefficient vector

[1)] In context of artificial neural networks, see Chapter **??**, ridge regression and Tikhonov regularization are also called *weight decay*.

$\boldsymbol{\alpha}$.[2)] By imposing this additional condition, we make the solution to the linear system in the least squares estimation unique, as discussed in

**Theorem 10.3** (Ridge regression) We are given the setting of Definition 10.4 and recall that we have the matrix $\mathcal{X} \in \mathbb{R}^{N \times M}$ with

$$\mathcal{X} = \begin{pmatrix} \varphi_1(\boldsymbol{x}_1) & \cdots & \varphi_M(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ \varphi_1(\boldsymbol{x}_N) & \cdots & \varphi_M(\boldsymbol{x}_N) \end{pmatrix}.$$

For $\lambda > 0$, the solution of the minimization problem in ridge regression is always uniquely given by

$$\hat{\boldsymbol{\alpha}} = \left(\mathcal{X}^\top \mathcal{X} + \lambda \mathcal{I}\right)^{-1} \mathcal{X}^\top \boldsymbol{y}\,,$$

where $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$ and $\mathcal{I} \in \mathbb{R}^{M \times M}$ is the identity matrix.

Proof

We start by deriving the solution of the minimization problem as in the proof of Theorem 4.1, however now with the setting of Theorem 10.1 and the above new optimization problem. As we have seen it in the proof of Theorem 4.1, we compute the gradient of the cost functional and set it to zero with

$$\begin{aligned} \mathbf{0} &= \nabla_{\boldsymbol{\alpha}} \sum_{i=i}^{N} (y_i - f_{\boldsymbol{\alpha}}(\boldsymbol{x}_i))^2 \;+\; \lambda \|\boldsymbol{\alpha}\|_2^2 \\ &= -2\mathcal{X}^\top (\boldsymbol{y} - \mathcal{X}\boldsymbol{\alpha}) + \nabla_{\boldsymbol{\alpha}} (\lambda \|\boldsymbol{\alpha}\|_2^2) \\ &= -2\mathcal{X}^\top (\boldsymbol{y} - \mathcal{X}\boldsymbol{\alpha}) + 2\lambda\boldsymbol{\alpha} \\ &= -2\mathcal{X}^\top \boldsymbol{y} + 2\mathcal{X}^\top \mathcal{X}\boldsymbol{\alpha} + 2\lambda\boldsymbol{\alpha} \\ &= -2\mathcal{X}^\top \boldsymbol{y} + 2(\mathcal{X}^\top \mathcal{X} + \lambda\mathcal{I})\boldsymbol{\alpha}\,. \end{aligned}$$

From the first line to the second line, we use Theorem 10.1. Then we simply evaluate the gradient of the squared norm of $\boldsymbol{\alpha}$. The rest are basic transformations. The above statement is equivalent to

$$(\mathcal{X}^\top \mathcal{X} + \lambda\mathcal{I})\boldsymbol{\alpha} = \mathcal{X}^\top \boldsymbol{y}\,.$$

To make sure that we can uniquely solve the linear system, or equivalently that $(\mathcal{X}^\top \mathcal{X} + \lambda\mathcal{I})$ is invertible, we first observe that, by construction, $\mathcal{X}^\top \mathcal{X}$ is at least positive semi-definite, thus all eigenvalues of $\mathcal{X}^\top \mathcal{X}$ are non-negative. Let $\sigma \geq 0$ be arbitrary but fixed eigenvalue of $\mathcal{X}^\top \mathcal{X}$ and $\boldsymbol{x}$ be an arbitrary vector that is in the space spanned up by the

[2)] This is also why ridge regression is one of the so-called *shrinkage methods*.

eigenvector that is associated to $\sigma$. Then, he have

$$\begin{aligned} & \sigma \boldsymbol{x} = \mathcal{X}^\top \mathcal{X} \boldsymbol{x} \\ \Leftrightarrow\ & \sigma \boldsymbol{x} + \lambda \boldsymbol{x} = \mathcal{X}^\top \mathcal{X} \boldsymbol{x} + \lambda \boldsymbol{x} \\ \Leftrightarrow\ & (\sigma + \lambda) \boldsymbol{x} = (\mathcal{X}^\top \mathcal{X} + \lambda \mathcal{I}) \boldsymbol{x} \,. \end{aligned}$$

In the second line, we simply add the term $\lambda \boldsymbol{x}$ to both sides, while in the last line, we just combine the terms. This short derivation shows that, if $\sigma$ is an eigenvalue of $\mathcal{X}^\top \mathcal{X}$, then the matrix $(\mathcal{X}^\top \mathcal{X} + \lambda \mathcal{I})$ has the eigenvalue $\sigma + \lambda$. Hence all eigenvalues are shifted by $\lambda$.

As we know that the smallest eigenvalue of $\mathcal{X}^\top \mathcal{X}$ is in worst case zero, however the smallest eigenvalue of $(\mathcal{X}^\top \mathcal{X} + \lambda \mathcal{I})$ is shifted away from zero by $\lambda$, the smallest possible eigenvalue of $(\mathcal{X}^\top \mathcal{X} + \lambda \mathcal{I})$ is $\lambda$, thus $(\mathcal{X}^\top \mathcal{X} + \lambda \mathcal{I})$ is invertible (irrespective of the training data). □

We should give a small technical remark for those, who might want to use ridge regression for the linear model.

**Remark** (Ridge regression for linear model) In the linear model, i.e. when choosing the basis $\varphi_1(\boldsymbol{X}) = 1, \varphi_2(\boldsymbol{X}) = X_1, \ldots, \varphi_{D+1}(\boldsymbol{X}) = X_D$, the coefficient $\alpha_1$, which is associated to the intercept, always depends on the centroid of the training data. Hence, when applying ridge regression, i.e. when further minimizing with respect to $\lambda \|\boldsymbol{\alpha}\|_2^2$, we suddenly couple the center of the training data with the regularization.

This coupling should be avoided by first centering the data and then fitting $\alpha_1$ independently of the remaining parameters, where we apply the regularization only to the remaining parameters. More details on this are given in [6]. △

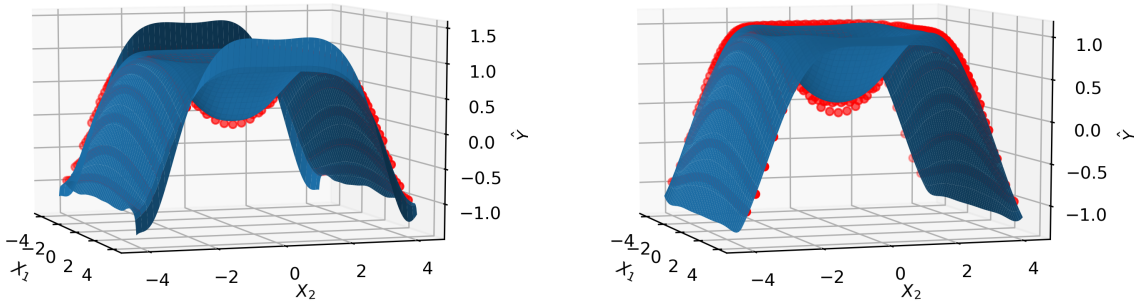
So far, we mostly motivated the use of ridge regression or Tikhonov regularization via the unique "solvability" of the least squares estimation. However, indeed, the main use of ridge regression is connected to *overfitting*.

**Remark** (Overfitting) As discussed earlier, in overfitting, we face the situation of having a model with a high complexity that has a too low bias but a rather high variance. The result of overfitting is that we observe a rather large generalization error, even if we have a (very) low training error. As one can see in practice, regularization / ridge regression is one of several tools to reduce overfitting. Indeed, by restricting the coefficient vector, we reduce the model complexity, hence slightly increase the bias while (often) more strongly reducing the variance. As observable outcome, we reduce the generalization error, while only slightly increasing the training error. △

Let us apply this intuition to Example 10.5 in which we computed a predictor with the kernel-base model.

**Example 10.6** (Ridge regression for kernel-based model) In Example 10.5, we have seen

that even when using the kernel-based model (compared to the polynomial model), we still observe a clear generalization error, when evaluating the predictor outside of the domain covered by the training set. Therefore, we now repeat the same experiment, however, we use this time *kernel ridge regression* (according to the remark below) with $\lambda = 0.001$ to reduce the overfitting effect. The result can be seen in the below figures.




On the left-hand side, we have the original situation, i.e. the original kernel model evaluated on $[-4.2, 4.2] \times [-4.2, 4.2]$. The right-hand side shows the new kernel model estimated via kernel ridge regression. It is apparent that the predictor "nicely" extends outside the training data.

This example including the shown figures can be reproduced and modified here: launch binder △

In some fields of application, the combination of ridge regression and the kernel-based model is rather popular. There, it is introduced as

**Method 16** (Kernel ridge regression) In *kernel ridge regression* we are given training data and a positive definite kernel. Using this, we build the kernel-based model and estimate its coefficients from the data via ridge regression. △

Combining Theorem 10.2 and Theorem 10.3, we obtain for kernel ridge regression

**Theorem 10.4** (Ridge regression for the kernel-based model) Let $\boldsymbol{X}, Y, \mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ be given as usual. Moreover, we are given a positive definite kernel $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ that leads to the kernel-based model

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{N} \alpha_j k(\boldsymbol{X}, \boldsymbol{x}_j) \,.$$

We consider the matrix $\mathcal{A} \in \mathbb{R}^{N \times N}$

$$\mathcal{A} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \ddots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix} .$$

Then the ridge regression estimator $\hat{\boldsymbol{\alpha}}$ for the parameters $\boldsymbol{\alpha}$ in the model $f_{\boldsymbol{\alpha}}$ is uniquely given by

$$\hat{\boldsymbol{\alpha}} = (\mathcal{A}^T\mathcal{A} + \lambda\mathcal{I})^{-1}\mathcal{A}^\top \boldsymbol{y}\,,$$

where $\boldsymbol{y} = (y_1, \ldots, y_N)^\top$.

**Remark** (Kernel ridge regression) In literature on kernel-based models, we mostly find under the notion of *kernel ridge regression* a model of the form

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{N} \alpha_j k(\boldsymbol{X}, \boldsymbol{x}_j)\,,$$

where the optimal set of coefficients is given by

$$\hat{\boldsymbol{\alpha}} = (\mathcal{A} + \lambda\mathcal{I})^{-1}\boldsymbol{y}\,.$$

Indeed, this result cannot be directly derived from Theorem 10.3. Instead, the coefficients $\hat{\boldsymbol{\alpha}}$ are found by minimizing a *dual* minimization problem and by applying the so-called *kernel trick* [].

Alternatively, we can immeditately derive the *kernel ridge regression* model by minimizing the slightly modified cost functional

$$J(\boldsymbol{\alpha}) = \sum_{i=1}^{N} \left\| y_i - \sum_{j=1}^{N} \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) \right\|_2^2 + \lambda \|\boldsymbol{\alpha}\|_{\mathcal{A}}^2\,,$$

with $\|\boldsymbol{\alpha}\|_{\mathcal{A}}^2 = \boldsymbol{\alpha}^T\mathcal{A}\boldsymbol{\alpha}$ using its gradient being set to zero, as usual. In the following, we will use the just discussed *kernel ridge regression* model and not the one that is derived in Theorem 10.4. △

We can derive the algorithm to carry out kernel ridge regression as

**Algorithm 17** (Kernel ridge regression)

**input:** $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ training data, $\boldsymbol{x}$ evaluation point, $\lambda$ regularization parameter, $k$ positive definite kernel

**output:** kernel ridge regression prediction $\hat{Y}(\boldsymbol{x})$

1. build matrix $\mathcal{A}$
2. solve $(\mathcal{A} + \lambda\mathcal{I})\hat{\boldsymbol{\alpha}} = \boldsymbol{y}$ using Cholesky factorization
3. $\hat{Y}(\boldsymbol{x}) = \sum_{j=1}^{N} \hat{\boldsymbol{\alpha}}_j k(\boldsymbol{x}, \boldsymbol{x}_j)$
4. return $\hat{Y}(\boldsymbol{x})$

It has a complexity of $O(N^3 + D \cdot N^2)$. We conclude by discussing, how to choose the regularization parameter $\lambda$.

**Remark** (Choice of $\lambda$) In practical applications, the choice of the regularization parameter $\lambda$ is an additional (hyper-)parameter in the training. Qualitatively speaking, a too large $\lambda$ might introduce a too large bias, while a too small $\lambda$ might not reduce the issue of overfitting enought. Therefore, this regularization parameter is often estimated data-dependent using cross validatiion. $\triangle$

# 11. Artificial Neural Networks

In this chapter, we introduce *artificial neural networks* (ANN) as a non-linear parametric model for regression and classification tasks. Artificial neural networks were originally introduced as an artificial model for the brain. While this type of model was initially not very successful, some modifications in the training process and the existence of powerful compute hardware (graphics processing units, GPUs), recently leads to its very successful use for many applications.

In the following, we will start by introducing the *multilayer perceptron*, which is the simplest version of an artificial neural network, more specifically of a *feed-forward network*. After describing the general setup of this model in the first section, we will introduce the standard *backpropagation* algorithm for the training of the multilayer perceptron in the second section.

## 11.1 Multilayer perceptron

With the multilayer perceptron, we introduce a non-linear model. Let us briefly review two ideas on how to build a non-linear model in

**Remark** (Methods to build a non-linear model) In Section 10.1, we already introduced the *basis expansion model*, which is a means to build a non-linear model. As a reminder, for a given $D$-dimensional input $\boldsymbol{X}$, the basis expansion model is given by

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{M} \alpha_j \varphi_j(\boldsymbol{X}) \,.$$

The non-linearity is introduced by the transformation of the inputs via the basis functions $\varphi_j$. While the basis functions might have one or two hyperparameters, the model has generally speaking rather few, namely $M$ parameters. For reasons that will become clear soon, we briefly rewrite the basis expansion model as

$$f_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{j=1}^{M} \alpha_j \varphi_j(\boldsymbol{X}) = \sum_{j=1}^{M} \alpha_j Z_j \,,$$

where the $Z_j$s are new variables with $Z_j = \varphi_j(\boldsymbol{X})$.

Another approach to obtain a non-linear model is the *projection pursuit regression model*. It is given by

$$f_{\boldsymbol{\omega}}(\boldsymbol{X}) = \sum_{m=1}^{M} g_m(\boldsymbol{\omega}_m^\top \boldsymbol{X}), \qquad \text{with } \boldsymbol{\omega}_m \in \mathbb{R}^D .$$

In this model, the $g_m$ are $M$ fixed non-linear functions that are applied each to a different linear model. Note that we do not have a liner combination, however just a sum over the non-linearly transformed linear models. Obviously, the resulting model has many parameters, namely $D \cdot M$. △

The *multilayer perceptron* combines these ideas in a single approach, which we motivate in

**Remark** (Multilayer perceptron motivation) In the *multilayer perceptron* (with one hidden layer), we combine the basis expansion model with the projection pursuit regression model as follows. As always, we start with an input (vector) $\boldsymbol{X} = (X_1, \dots, X_D)^\top$. Then, similar to the projection pursuit model, we build $M$ linear models and apply some non-linear *activation function* $\sigma$ to them, thus

$$\sigma(\omega_{m0} + \boldsymbol{\omega}_m^\top \boldsymbol{X}), \qquad m = 1, \dots, M .$$

However, instead of just adding them up, we consider them as new basis functions $\varphi_m(\boldsymbol{X})$ or transformed variables $\boldsymbol{Z}_m$, obtaining

$$Z_m = \varphi_m(\boldsymbol{X}) = \sigma(\omega_{m0} + \boldsymbol{\omega}_m^\top \boldsymbol{X}) .$$

These basis functions or transformed variables are finally recursively used to build up a new model in the style of the basis expansion model, thus

$$f(\boldsymbol{X}) = \alpha_0 + \sum_{m=1}^{M} \alpha_m \varphi_m(\boldsymbol{X}) = \alpha_0 + \sum_{m=1}^{M} \alpha_m Z_m .$$

△

We immediately generalize this idea from the one-dimensional output case to the $K$-dimensional output case and obtain

**Definition 11.1** (Multilayer perceptron) Let $\boldsymbol{X}$ be a $D$-dimensional input and $\boldsymbol{Y}$ be a $K$-dimensional quantitative output. The **multilayer perceptron (with one hidden layer)** (MLP) is the non-linear model $f = (f_1, \dots, f_K)^\top$ with

$$\begin{aligned} Z_m &= \sigma\left(\omega_{m0} + \boldsymbol{\omega}_m^\top \boldsymbol{X}\right), && \text{for } m = 1, \dots, M , \\ T_k &= \alpha_{k0} + \sum_{m=1}^{M} \alpha_{km} Z_m, && \text{for } k = 1, \dots, K , \\ f_k(\boldsymbol{X}) = Y_k &= g_k\left(T_1, \dots, T_K\right), && \text{for } k = 1, \dots, K , \end{aligned}$$

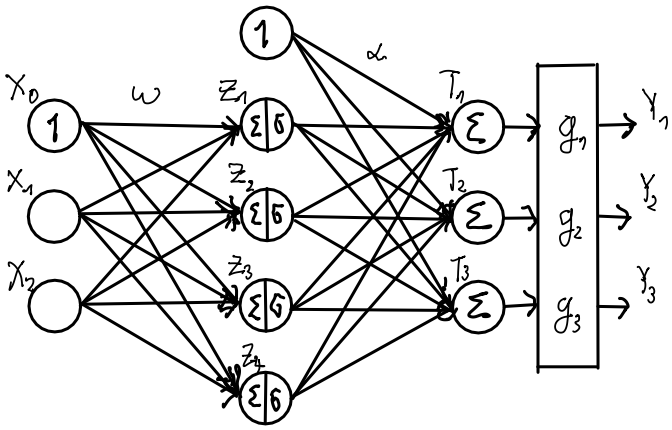
where

- the $X_j$ are called **input units** and form the **input layer**,
- the $Z_m$ are called **hidden units** and form the **hidden layer**,
- the $Y_k$ are called **output units** and form the **output layer**,
- the $\boldsymbol{\omega}$, $\boldsymbol{\alpha}$ are called the **weights**,
- $\sigma : \mathbb{R} \to \mathbb{R}$ is a non-linear **activation (function)**, and
- $g_k : \mathbb{R}^K \to \mathbb{R}$ is the **output activation (function)**.

The *units* are also called **neurons** or **variables**.

As the definition is a bit unwieldy, we will carry its out the discussion in immediate connection to an example.

**Example 11.1** (A simple multilayer perceptron) It is much more handy to discuss the MLP via the visual representation that is typically given for it. To this end, we limit ourselves to $D = 2$ input dimensions and $K = 3$ output dimensions and the hidden layer with $M = 4$ hidden units:




In the above visualization, each unit/neuron/variable is depicted by a node that is connected by edges. Each edge corresponds to a weight. Thereby, we form a *network*, which is a special type of graph. The input units $X_1$ and $X_2$ in the *input layer* simply represent the input variables. We also add a node with the label 1, to indicate that we have an intercept, which we could also give by $X_0 \equiv 1$. From the *input layer*, we use the equation

$$Z_m = \sigma\left(\omega_{m0} + \boldsymbol{\omega}_m^\top \boldsymbol{X}\right)$$

to obtain the values that are represented by the units / variables / neurons $Z_m$ in the *hidden layer*. Each unit $Z_m$, hence this newly computed variable, takes a weighted linear combination of the inputs and the intercept (as indicated by the $\Sigma$ symbol) and applies

the *activation* function $\sigma$ to the sum. Each edge between an input $X_j$ unit and a hidden unit $Z_m$ corresponds to a weight $\omega_{mj}$ in the linear combination.

In a next step, the model takes the $Z_m$ variables from the hidden layer (plus an intercept), and uses the formula $T_k = \alpha_{k0} + \sum_{m=1}^{M} \alpha_{km} Z_m$ to compute an intermediate output as weighted sum. Here, an edge between the unit $Z_m$ in the hidden layer and the drawn node $T_k$ represents the weight $\alpha_{km}$. The variables $T_k$ are part of the output layer. Depending the application we need to apply an additional *output activation* to the variables $T_k$, which might also couple the values from all variables $T_1, \ldots, T_k$. The three outputs $Y_1, \ldots, Y_3$ are thus formed by

$$Y_k = g_k(T_1, \ldots, T_3)\,.$$

The multilayer perceptron as given by Definition 11.1 deviates from the one described in the earlier motivating remark only by a higher-dimensional output, i.e. here $K = 3$ instead of only one output unit and the choice of $g$, which was simply the identity (i.e. non-existent) in that remark. △

Given the above re-interpretation as a graph we can also better understand the notion of a *feed-forward neural network*. It is simply an artificial neural network, in which the corresponding graph has no circles.

Even though Example 11.1 might have already given some more insight into Definition 11.1, there are still some parts, which we only skipped over. One of them are the *activations*.

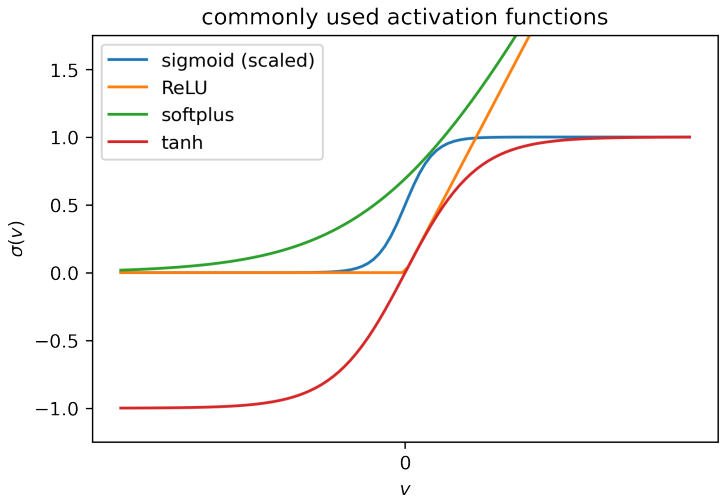
Ignoring for now the output activation, the activation function $\sigma$ is the only component in the model that makes it non-linear. Hence, if we were to set $\sigma$ as the identity or some linear function, the multilayer perceptron would only be a very complicated way to write down a linear model. So how to choose $\sigma$?

Starting with a historical remark, the notion of an "activation" stems from the early interpretation of the ANN as a model for the brain. There, a neuron would send out a signal, if gets from other neurons an input that goes beyond some threshold. Many of the commonly used non-linear activation functions try to mimic this thresholding behavior, as we see in

**Example 11.2** (Commonly used activation functions) We list a few commonly used activation functions with

- the *sigmoid function* $\sigma(v) = \frac{1}{1+\exp(-v)}$,
- the *ReLU function* $\sigma(v) = \begin{cases} 1 & v \geq 0 \\ 0 & v < 0 \end{cases}$,
- the *softplus function* (as smoothed ReLU) $\sigma(v) = log(1 + \exp(v))$, and
- the *tanh function* $\sigma(v) = \tanh(v)$.

The below visualization shows (scaled versions) of all of these functions. It becomes becomes evident that they all somewhat implement a thresholding around the value 0.




△

As we have discussed before, the MLP would be a liner model without the activation $\sigma$. Even though all proposed activations are non-linear functions, it can happen that they behave as linear functions, as discussed in

**Remark** (Linear behavior of $\sigma$) For some of the usually used activations, like the sigmoid function the numerical behavior for small input values $v$, i.e. for values $v$ that are in the range of almost 0, can be the one of a linear function, as we can see it in the visualization given in Example 11.1. Hence, it can happen that, in case of too small weights, the MLP starts to behave like a linear model. This is an issue that we have to keep in mind. A proposal to deal with this problem will be given in Section **??**. △

As just seen, the choice of the activation is an important degree of freedom in the construction of multilayer perceptron. We also did not discuss, yet, the choice of the *output activation*. It will become clear, if we actually study the two use cases of the MLP, namely regression and classification. Let us start by regression in

**Method 17** (Regression using the multilayer perceptron) In regression using the multilayer perceptron, we are as usual given training data $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, $\boldsymbol{x}_i \in \mathbb{R}^D$, $\boldsymbol{y}_i \in \mathbb{R}^K$. The input and output dimensionality of that data already implies the size, i.e. the number of units, of the input and output layer. As degrees of freedom, we can choose the number of hidden units $M$ and the activation function $\sigma$ in the model. In regression, it is moreover usual to set

$$g_k(T_1, \dots, T_k) = T_k$$

for all $k = 1, \dots, K$, thus the output activation is simply the identity and is thus dropped.[1)] Having made the selection of number of hidden units and activation, we fit

[1)] Other choices are possible. To give an example, ReLU or softplus is sometimes used to enforce non-negative outputs.

the weights $\boldsymbol{\omega} = (\omega_{mj})_{m=1,\dots,M,\, j=0,\dots,D}$, $\boldsymbol{\alpha} = (\alpha_{km})_{k=1,\dots,K,\, m=0,\dots,M}$[2)] by solving

$$(\hat{\boldsymbol{\omega}}, \hat{\boldsymbol{\alpha}}) = \arg\min_{(\boldsymbol{\omega},\boldsymbol{\alpha})} \sum_{i=1}^{N} L_2(\boldsymbol{y}_i, f(\boldsymbol{x}_i)) = \arg\min_{(\boldsymbol{\omega},\boldsymbol{\alpha})} \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik}, f_k(\boldsymbol{x}_i))^2 \,,$$

where $L_2$ is (here) the $K$-dimensional squared loss. The solution of the above mimimization problem, i.e. the training, is carried out by some gradient descent type algorithm. Finally, the constructed model can be evaluated for a given evaluation point. △

Next, we discuss classification. The attentive reader will see very strong similarities to *logistic regression*.

**Method 18** (Classification using the multilayer perceptron) We assume that we are given training data of the form $\mathcal{T} = \{(\boldsymbol{x}_i, g_i)\}_{i=1}^N$, where the input is $D$-dimensional, i.e. $\boldsymbol{x}_i \in \mathbb{R}^D$, however we have a one-dimensional $r$-class output, thus $g_i \in \{1, \dots, r\}$. As we have seen it in classification for the indicator matrix, we then convert the output measurements to a one-hot encoding, thus generate for $i = 1, \dots, N$

$$\boldsymbol{y}_i \in \mathbb{R}^r = \begin{pmatrix} y_{i1} \\ \vdots \\ y_{ir} \end{pmatrix}, \qquad \text{with } y_{il} = \begin{cases} 1 & g_i = l \\ 0 & g_i \neq l \end{cases} .$$

Hence, we have constructed a regression problem with $D$ dimensional input and $r$-dimensional output, thus have an input layer size of $D$ (not counting the intercept) and an output layer size of $K = r$. Similar to the regression case, we are thus left with choosing the number of hidden units $M$ and the activation $\sigma$. In contrast, to the regression case, we however have to make sure that the $r$ outputs behave like the class posteriors $p(g|\boldsymbol{x}_i)$. Therefore, we choose as output activation the softmax function

$$g_k(T_1, \dots, T_K) = \frac{\exp(T_k)}{\sum_{l=1}^K \exp(T_l)} \,,$$

which we have already used in logistic regression to enforce the necessary property.

Having selected all parameters above, the training in classification using the MLP is usually carried out by minimizing the cross entropy loss as in the logistic regression case

$$(\hat{\boldsymbol{\omega}}, \hat{\boldsymbol{\alpha}}) = \arg\min_{(\boldsymbol{\omega},\boldsymbol{\alpha})} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(\boldsymbol{x}_i)) \,.$$

Finally, the predicted class is obtained for an evaluation point $\boldsymbol{x}$ by solving

$$\arg\max_k f_k(\boldsymbol{x}) \,.$$

△

[2)] To keep the notation readable, we – for now – express the weights just as two very long vectors, even though a treatment as a matrix will turn out to be more meaningful.

To summarize, classification using the MLP is the combination of regression using the MLP (on the indicator matrix) and logistic regression. Certainly, there are also cases, in which other combinations of output activations and encodings of the qualitative output might be used. We here however just stick to the most typical case.

The two method descriptions above outline that we will always have to make choices on quite some parameters in the MLP and more generally in artificial neural networks. We fix some notion for these choices in a rather informal

**Definition 11.2** (Network architecture) The collection of all choices of number of neurons, activations, output activation, etc. is called *network architecture*.

Now we would like to go "deep", hence move from a single hidden layer to multiple hidden layers in the MLP. However, before that, we make a small notational change as outlined in

**Remark** (Notational change for the weights) Due to the derivation of the MLP from the basis expansion model and the projection pursuit regression model, we have given the weights between the input and the hidden layer the name $\boldsymbol{\omega}$ and the weights between the hidden layer and the output layer the name $\boldsymbol{\alpha}$. However, besides of their location in the network, there is no clear reason to distinguish them by a different name. Therefore, we now switch to a notation, in which we simply rename the weights $\boldsymbol{\omega}$ as $\boldsymbol{\omega}^{(1)}$ and the weights $\boldsymbol{\alpha}$ as $\boldsymbol{\omega}^{(2)}$. Thereby the three defining equations of the MLP become

$$\begin{aligned} Z_m &= \sigma\left(\omega_{m0}^{(1)} + {\boldsymbol{\omega}_m^{(1)}}^\top \boldsymbol{X}\right), \quad \text{for } m = 1, \dots, M, \\ T_k &= \omega_{k0}^{(2)} + \sum_{m=1}^{M} \omega_{km}^{(2)} Z_m, \quad \text{for } k = 1, \dots, K, \\ f_k(\boldsymbol{X}) &= Y_k = g_k\left(T_1, \dots, T_K\right), \quad \text{for } k = 1, \dots, K. \end{aligned}$$

△

This facilitates the move to the MLP with $L$ hidden layers in

**Definition 11.3** (Multilayer perceptron (multiple hidden layers)) Let $\boldsymbol{X}$ be a $D$-dimensional input and $\boldsymbol{Y}$ be a $K$-dimensional quantitative output. The **multilayer**

**perceptron** with $L$ hidden layers is the non-linear model $f_{\boldsymbol{\omega}} = (f_{\boldsymbol{\omega},1}, \ldots, f_{\boldsymbol{\omega},K})^\top$

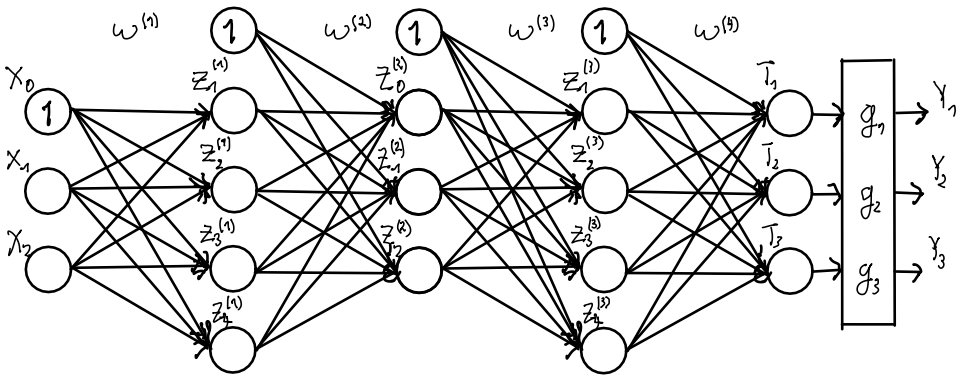
$$
\begin{aligned}
Z_j^{(0)} &= X_j, && \text{for } j = 1, \ldots, D\,, \\
Z_m^{(l)} &= \sigma_l\left(\omega_{m0}^{(l)} + \boldsymbol{\omega}_m^{(l)\top} \boldsymbol{Z}^{(l-1)}\right), && \text{for } m = 1, \ldots, M_l,\ l = 1, \ldots, L\,, \\
T_k &= \omega_{k0}^{(L+1)} + \sum_{m=1}^{M_L} \omega_{km}^{(L+1)} Z_m^{(L)}, && \text{for } k = 1, \ldots, K\,, \\
f_{\boldsymbol{\omega},k}(X) = Y_k &= g_k\left(T_1, \ldots, T_K\right), && \text{for } k = 1, \ldots, K\,,
\end{aligned}
$$

where, in difference to Definition 6.1

- the $Z_m^{(l)}$ are the hidden units of the $l$th hidden layer,
- $\sigma_l : \mathbb{R} \to \mathbb{R}$ is the activation (function) of the $l$th layer.

Here, we simply recursively repeat the idea of the construction of the hidden units / variables form the single hidden layer case. Moreover, we treat the input layer as 0th hidden layer. Also, we allow for layer-dependent activations $\sigma_l$. The idea becomes much more clear, if we investigate a visualization of a concrete example in

**Example 11.3** (MLP with multiple hidden layers) The following visualization shows a multilayer perceptron with $L = 3$ hidden layers, an input layer size of $D = 2$, an output layer size of $K = 3$ and hidden layer sizes $M_1 = 4$, $M_2 = 3$ and $M_3 = 4$.

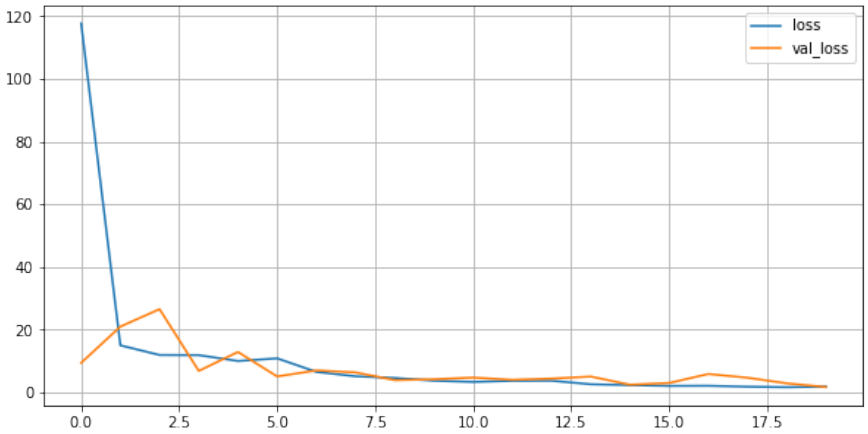


△

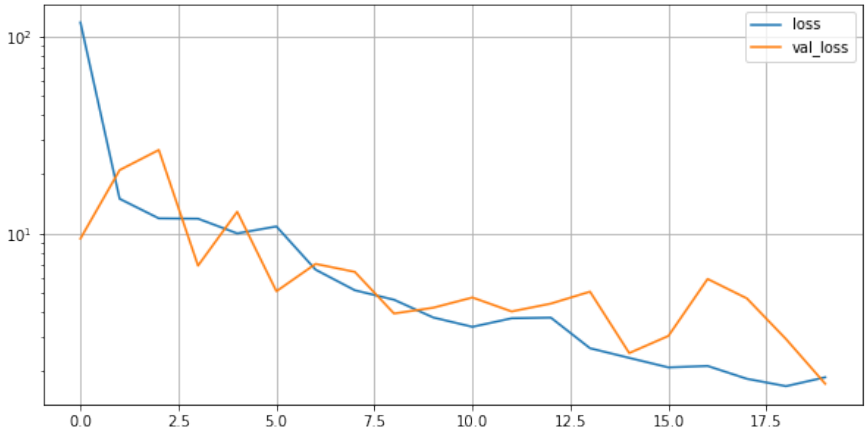
The above example gives our first "deep" neural network, noting that in practice various numbers of layers are used.

Even though, we don't know yet exactly how to train an MLP, we want to take the opportunity to give a first example of regression using an MLP model for given training data.

**Example 11.4** (Regression using an MLP for the energy efficiency data set) In this example, we use the energy efficiency data set discussed in Example **??** and used in Example **??**. As a reminder, in this data set, we have a $D = 8$-dimensional input and a one-dimensional quantitative output. Hence, we want to solve a regression task. We use a simple MLP to cover this regression task. In the MLP, we have $L = 4$ hidden layers, each with $M_l = 30$ units. The hidden layers are further equipped with the ReLu activation. As discussed in Method 17, we use the $L_2$ loss in the minimization, i.e. do a least squares fit. The training is carried out with stochastic gradient descent with a learning rate of $\eta = 10^{-3}$. Overall, the training is run over 30 epochs. Below, we report on the development of the training and generalization error, where the latter one is computed using the validation set approach.




Here, "loss" stands for the training error and "val_loss" stands for the validation error. As, for a training / validation error closer to zero, it becomes hard to seed the further progress in the training, it is useful to do a semi-logarithmic plot with the error given on a logarithmic vertical axis, as below.




Here, we can see that the training error consistently goes down, while the validation error fluctuates a bit. Towards the end of the training process, the model has a slight tendency towards overfitting.

This example including the shown figures can be reproduced and modified here: launch binder △

At this point, still, some questions are open on the multilayer perceptron. First, we might ask ourselves how to exactly train a multilayer perceptron. Then, we might ask ourselves what are good architectures to solve some problems and even more, how things work in practice. Finally, we might want to understand the difference between MLPs and recent *deep neural networks*.

The first question on training an MLP is answered in the next section. All other questions will be answered in the last chapter.

## 11.2 Backpropagation

In this section, we introduce the *backpropagation* algorithm, which is the standard approach for training an artificial neural network. As part of the backpropagation algorithm, we apply stochastic or mini-batch gradient descent to minimize the cost functional, which is given by

$$J(\boldsymbol{\omega}) = \sum_{i=1}^{N} L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i)) \,,$$

with $L$ some loss. Then, one step of, e.g., mini-batch gradient descent becomes

$$\boldsymbol{\omega}' = \boldsymbol{\omega} + \eta \sum_{i=1}^{N_b} \nabla_{\boldsymbol{\omega}} L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i)) \,.$$

Hence in each step of the gradient descent, we need to evaluate $\nabla_{\boldsymbol{\omega}} L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i))$ for the MLP model $f_{\boldsymbol{\omega}}$.

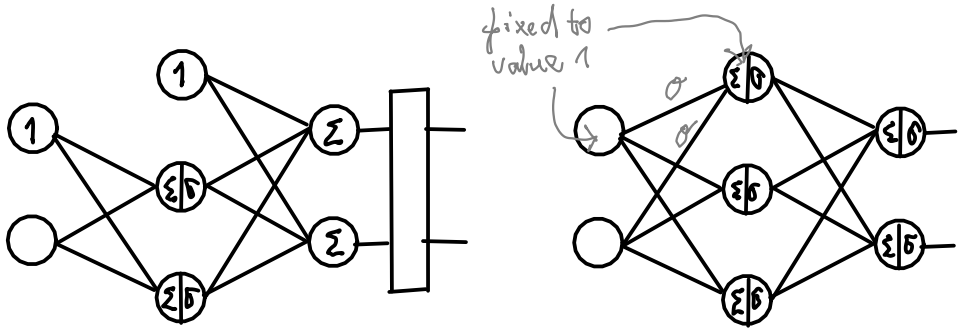
We will show that the evaluation of the gradient of the loss for the MLP model is a two-step process. In the first step, the *forward pass* or *forward propagation*, we take the input $\boldsymbol{x}_i$ and propagate it through the neural network. Then, in a second step, the *backward pass* or *backward propagation*, a value similar to $L(\boldsymbol{y}_i, f_{\boldsymbol{\omega}}(\boldsymbol{x}_i))$ is evaluated and its gradient with respect to $\boldsymbol{\omega}$ is derived by propagating this loss evaluation backwards through the network. The combination of gradient descent and forward and backward passes is then the *backpropagation* algorithm.

To enter the derivation of the backpropagation algorithm and more explicitly the forward and backward pass, we slightly generalize and simplify the description of the underlying multilayer perceptron.

**Remark** (Simplified description of MLP) As the derivation of the backpropagation is anyway quite technical, we make a few simplifications. The first simplification is that we treat all activations equally, i.e. we no longer distinguish between *activations* and *output activations*. Thereby, we exclude the case, where an activation might couple across the output units. This is e.g. the case for the softmax activation. The second

simplification is that we treat all units equally, hence explicitly removing intercept units. The third simplification is that we assume that there exist edges, thus weights, beween all nodes of consecutive layers, while we accept that there might be edges with a fixed weight of zero. This last modification indeed allows us to still introduce an intercept, by fixing the value of a unit to 1.

Let us give a very short example to visualize the changes:




On the left-hand side, we have the original MLP with one input, a hidden layer with two units and an output layer that would also support the softmax activation as for classification. On the right-hand side, we treat all units, including the output units identical. Moreover, we added missing edges. To still implement the intercept logic, we set the weights of the two added edges to constant zero and make sure that somewhere in the code, that executes the MLP, the output of the intercept nodes are always overwritten by ones. △

Note that the artificial limitation of having no coupling in the output units, thus excluding classification, can be fixed by some special treatment of the output layer. We, however, will not discuss this further.

The above simplifications lead to a slightly changed multilayer perceptron model given by

**Definition 11.4** (Multilayer perceptron (simplified)) Let $\boldsymbol{X}$ be a $D$-dimensional input and $\boldsymbol{Y}$ be a $K$-dimensional quantitative output. The **multilayer perceptron** with $L$

hidden layers is the non-linear model $f_{\boldsymbol{\omega}} = (f_{\boldsymbol{\omega},1}, \ldots, f_{\boldsymbol{\omega},K})^\top$

$$
\begin{aligned}
Z^{(0)} &= X \\
Z_m^{(l)} &= \sigma_l\left(A_m^{(l)}\right), \quad A_m^{(l)} = \sum_{v=1}^{M_{l-1}} \omega_{mv}^{(l)} Z_v^{(l-1)}, \\
&\qquad \text{for } m = 1, \ldots, M_l,\ l = 1, \ldots, L+1,\ M_{L+1} = K\,, \\
f_{\boldsymbol{\omega},k}(X) &= Y_k = Z_k^{(L+1)},\ \text{for } k = 1, \ldots, K\,,
\end{aligned}
$$

where, we have

- $Z_m^{(l)}$: $m$th unit of the $l$th layer,
- $A_m^{(l)}$: input to the activation in the $m$th unit of the $l$th layer,
- $\omega_{mv}^{(l)}$: weight between units $Z_v^{(l-1)}$ and $Z_m^{(l)}$, and
- $\sigma_l$: activation (function) of the $l$th layer.

In comparison to the original model, the output layer is now just the $(L+1)$th layer and all layers follow the same formula

$$
Z_m^{(l)} = \sigma_l\left(\sum_{v=1}^{M_{l-1}} \omega_{mv}^{(l)} Z_v^{(l-1)}\right)\,.
$$

As a further slight change, we split the computation of the $l$th layer in first the application of the linear combination of the previous layers with

$$
A_m^{(l)} = \sum_{v=1}^{M_{l-1}} \omega_{mv}^{(l)} Z_v^{(l-1)}
$$

and second the application of the activation with

$$
Z_m^{(l)} = \sigma_l\left(A_m^{(l)}\right)\,.
$$

This helps us to describe the forward and backward pass in the backpropagation algorithm.

In a next step, we introduce the forward pass, which is simply the evaluation of the MLP model for a given input $\boldsymbol{x}$. In principle, we could just apply Definition 11.4 to do the forward pass. However, from an implementation perspective, it is more convenient to reformulate the model evaluation in terms of linear algebra operations. If we do that, we obtain

**Lemma 11.1** (Forward pass) Let a multi-layer perceptron as in Definition 11.4 be given. It holds for $l = 1, \dots, L+1$

$$\begin{aligned} \boldsymbol{A}^{(l)} &= \mathcal{W}^{(l)} \boldsymbol{Z}^{(l-1)}, \\ \boldsymbol{Z}^{(l)} &= \boldsymbol{\sigma}_l(\boldsymbol{A}^{(l)}). \end{aligned}$$

with $\mathcal{W}^{(l)} \in \mathbb{R}^{M_l \times M_{l-1}}$, with entries $\omega_{mv}^{(l)}$, $m = 1, \dots, M_l$, $v = 1, \dots, M_{l-1}$ and $\boldsymbol{\sigma}_l : \mathbb{R}^{M_l} \to \mathbb{R}^{M_l}$ the function that applies element-wise the scalar function $\sigma_l$. Furthermore, we have $Z^{(l)} = \left(Z_1^{(l)}, \dots, Z_{M_l}^{(l)}\right)^\top$ and $\boldsymbol{A}^{(l)} = \left(A_1^{(l)}, \dots, A_{M_l}^{(l)}\right)^\top$

There is no need for a proof, as this lemma just expresses the original model as a consecutive application of matrix-vector products and the application of the activation. This result gives rise to

**Algorithm 18** (MLP evaluation, a.k.a. forward pass / propagation)
**input:** MLP given by $\left(L, \{M_l\}_{l=0}^{L+1}, \{\mathcal{W}^{(l)}\}_{l=1}^{L+1}, \{\boldsymbol{\sigma}_l\}_{l=1}^{L+1}\right)$, $\boldsymbol{x}$ input
**output:** $\boldsymbol{f}_{\boldsymbol{\omega}}(\boldsymbol{x}) = (f_{\boldsymbol{\omega},1}(\boldsymbol{x}), \dots, f_{\boldsymbol{\omega},K}(\boldsymbol{x}))^\top$

1. $\boldsymbol{z}^{(0)} \leftarrow \boldsymbol{x}$
2. for $l = 1, \dots, L+1$:
   a) $\boldsymbol{a}^{(l)} = \mathcal{W}^{(l)} \boldsymbol{z}^{(l-1)}$
   b) $\boldsymbol{z}^{(l)} = \boldsymbol{\sigma}_l\left(\boldsymbol{a}^{(l)}\right)$
3. $\boldsymbol{f}_{\boldsymbol{\omega}}(\boldsymbol{x}) \leftarrow \boldsymbol{z}^{(L+1)}$
4. return $\boldsymbol{f}_{\boldsymbol{\omega}}(\boldsymbol{x})$

If we the size of the input and output layers are as usual $D$ and $K$ and if we further choose a fixed size $M$ for all $L$ hidden layers, then we obtain a complexity of $O(M \cdot D + M \cdot K + L \cdot M^2)$ for the above algorithm.

Let us briefly recall that our objective is to compute the gradient of the loss $L(\boldsymbol{f}_{\boldsymbol{\omega}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$ with respect to the weights in the MLP, hence

$$\nabla_{\boldsymbol{\omega}} L(\boldsymbol{f}(\boldsymbol{x}_i), \boldsymbol{y}_i) = \left(\frac{\partial L}{\partial \omega_{mv}^{(l)}}\right)_{\substack{l=1,\dots,L+1 \\ m=1,\dots,M_l \\ v=1,\dots,M_{l-1}}}.$$

We need a small

**Remark** (Notation to express weights) Note that the notation for the weights, here, gets a bit weird. Indeed $\boldsymbol{\omega}$ is espressed as a very long vector that runs over three different indices. If it were to run over two indices, we would usually express it as a matrix. The

mathematical equivalent for a three-dimensional object of numbers is an *order* 3 *tensor*. However, we want to avoid this new terminology. Therefore, we confront the reader with the idea of a vector that runs over three indices. △

We now want to derive an important formula that will allow us to compute the gradient from above. To this end, we first need to get a bit of knowledge from calculus in higher dimensions.

**Knowledge 11.1** (Chain rule in higher dimensions) Let $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ be a function that is differentiable in a location $\boldsymbol{p} \in \mathbb{R}^n$ and $\boldsymbol{g} : \mathbb{R}^m \to \mathbb{R}^l$ be a function that is differentiable in a location $\boldsymbol{q} = \boldsymbol{f}(\boldsymbol{p})$. Then, its concatenation $\boldsymbol{h} = \boldsymbol{g} \circ \boldsymbol{f}$ is differentiable in location $\boldsymbol{p}$ and it holds the **chain rule**

$$d\boldsymbol{h}(\boldsymbol{p}) = d(\boldsymbol{g} \circ \boldsymbol{f})(\boldsymbol{p}) = d\boldsymbol{g}(\boldsymbol{q}) \circ d\boldsymbol{f}(\boldsymbol{p})\,.$$

Since we are rather used to express derivatives in terms of partial derivatives, we further introduce explicit statements of what input vectors are mapped to what output vectors. Hence, we introduce

$$\begin{aligned} \boldsymbol{f} &: (x_1, \ldots, x_n)^\top \mapsto (y_1, \ldots, y_m)^\top \\ \boldsymbol{g} &: (y_1, \ldots, y_m)^\top \mapsto (z_1, \ldots, z_l)^\top \\ \boldsymbol{h} &: (x_1, \ldots, x_n)^\top \mapsto (z_1, \ldots, z_l)^\top \end{aligned}$$

Then, we obtain the chain rule with partial derivatives as

$$\frac{\partial z_j}{\partial x_k} = \sum_{i=1}^{m} \frac{\partial z_j}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_k}\,,$$

for all $j = 1, \ldots, l$ and $k = 1, \ldots, n$.

The chain rule in higher dimensions allows us to introduce

**Lemma 11.2** Let a multi-layer perceptron as in Definition 11.4 be given. Moreover, the activations $\sigma_l$ are required to be appropriately differentiable. We want to compute the gradient $\nabla_{\boldsymbol{\omega}} L(\boldsymbol{f}_{\boldsymbol{\omega}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$, where we treat $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ as constant. Thus the function $L$ only depends on the weights vector $\boldsymbol{\omega}$.
Then, it holds for $l = 1, \ldots, L+1$, $m = 1, \ldots, M_l$, $v = 1, \ldots, M_{l-1}$:

$$\frac{\partial L}{\partial \omega_{mv}^{(l)}} = \delta_m^{(l)}\, Z_v^{(l-1)}\,,$$

where we define $\delta_m^{(l)}$ by the partial derivative

$$\frac{\partial L}{\partial A_m^{(l)}}\,.$$

Proof

The proof requires to have the visual representation of the MLP in the back of the mind. Assuming some fixed training sample $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, the computation of the loss can be understood as mapping

$$\boldsymbol{\omega} = \left(\omega_{11}^{(1)}, \ldots, \omega_{M_{L+1}M_L}^{(L+1)}\right) \quad \mapsto \quad L\,,$$

thus given the set of weights, we obtain the value $L$ that is the loss. Note that we slightly abuse notation here, since we identify the function name $L$ with the value $L(\boldsymbol{f}_{\boldsymbol{\omega}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$ that is computed.

The value for the loss is obtained by the forward pass. Thus in fact, $L$ is computed by propagating the information from the left to the right through the MLP. This propagation can be understood as a sequence of concatenations of functions that are recursivly applyed to each other.

Let us now mentally cut the network within one layer $l$ into two pieces, such that we obtain the value for the vector of variables $\boldsymbol{A}^{(l)}$ by a mapping $\boldsymbol{s}$

$$\boldsymbol{s} : \boldsymbol{\omega} = \left(\omega_{11}^{(1)}, \ldots, \omega_{M_{L+1}M_L}^{(L+1)}\right) \quad \mapsto \quad \boldsymbol{A}^{(l)} = (A_1^{(l)}, \ldots, A_{m_l}^{(l)})\,.$$

Moreover, starting from the variables $\boldsymbol{A}^{(l)}$, we have the mapping $\boldsymbol{t}$ that represents the remainder of the network including the loss evaluation as

$$\boldsymbol{t} : \boldsymbol{A}^{(l)} = (A_1^{(l)}, \ldots, A_{M_l}^{(l)}) \quad \mapsto \quad L\,.$$

Hence, we can evaluate the loss by evaluating the concatenated functions $\boldsymbol{t} \circ \boldsymbol{s}$. This is the scenario in which we can apply the chain rule from Knowledge 11.1 and obtain

$$\frac{\partial L}{\partial \omega_{mv}^{(l)}} = \sum_{i=1}^{M_l} \frac{\partial L}{\partial A_i^{(l)}} \cdot \frac{\partial A_i^{(l)}}{\partial \omega_{mv}^{(l)}}\,.$$

By further investigating the partial derivative $\frac{\partial A_i^{(l)}}{\partial \omega_{mv}^{(l)}}$, we observe

$$\frac{\partial A_i^{(l)}}{\partial \omega_{mv}^{(l)}} = 0 \qquad\qquad \text{if } i \neq m\,,$$

since for $i \neq m$ the weight $\omega_{mv}^{(l)}$ does not show up in the chain of formulas that leads to the computation of $A_i^l$. Thus, we obtain

$$\frac{\partial L}{\partial \omega_{mv}^{(l)}} = \sum_{i=1}^{M_l} \frac{\partial L}{\partial A_i^{(l)}} \cdot \frac{\partial A_i^{(l)}}{\partial \omega_{mv}^{(l)}} = \frac{\partial L}{\partial A_m^{(l)}} \cdot \frac{\partial A_m^{(l)}}{\partial \omega_{mv}^{(l)}}\,.$$

Having a further look at $\frac{\partial A_m^{(l)}}{\partial \omega_{mv}^{(l)}}$, we compute

$$\frac{\partial A_m^{(l)}}{\partial \omega_{mv}^{(l)}} = \frac{\partial}{\partial \omega_{mv}^{(l)}} \left( \sum_{u=1}^{M_{l-1}} \omega_{mu}^{(l)} Z_u^{(l-1)} \right) = Z_v^{(l-1)}\,,$$

where the first step is just the definition of $A_m^{(l)}$ and the second step incorporates that the $Z_u^{(l-1)}$ do not depend on $\omega_{mv}^{(l)}$ and that obviously

$$\frac{\partial \omega_{mu}^{(l)}}{\partial \omega_{mv}^{(l)}} = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{if } u \neq v \end{cases} .$$

As a conclusion, we obtain

$$\frac{\partial L}{\partial \omega_{mv}^{(l)}} = \frac{\partial L}{\partial A_m^{(l)}} \cdot \frac{\partial A_m^{(l)}}{\partial \omega_{mv}^{(l)}} = \frac{\partial L}{\partial A_m^{(l)}} \cdot Z_v^{(l-1)} = \delta_m^{(l)} Z_v^{(l-1)}$$

with the definition of $\delta_m^{(l)}$ given in the statement of the lemma. □

Let us inspect, what the lemma provides us. If we were able to compute $\delta_m^{(l)}$ and $Z_v^{(l)}$ for all $l = 1, \dots, L+1$, $m = 1, \dots, M_l$ and $v = 1, \dots, M_{l-1}$, then we would be able to evaluate the full gradient of the loss. Indeed, we already know how to compute the $Z_v^{(l)}$. This is simply done by picking a training sample $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ and applying the forward pass. Hence, what remains is to find a way to compute the $\delta_m^{(l)}$. This is done using

**Theorem 11.1** (Backward pass / propagagtion formula) Let the setting of Lemma 11.2 be given. Furthermore, we consider the $L_2$ loss. The $\delta_m^{(l)}$ are recursively given starting with

$$\delta_m^{(L+1)} = 2 \left( Z_m^{(L+1)} - y_{im} \right) \sigma'_{L+1} \left( A_m^{(L+1)} \right), \quad m = 1, \dots, M_{L+1}$$

and continuing with $l = L+1, \dots, 2$ as

$$\delta_v^{(l-1)} = \sigma'_{l-1} \left( A_v^{(l-1)} \right) \sum_{m=1}^{M_l} \delta_m^{(l)} \omega_{mv}^{(l)}, \quad v = 1, \dots, M_{l-1},$$

where $\sigma'$ is the first derivative of the activation.

How to interpret this statement? Indeed, we start at the right-hand side of out network in with the data $Z_m^{(l+1)}$ in the output layer, which is just the evaluated MLP for input $\boldsymbol{x}_i$. From this value, we subtract the given training information $y_{im}$ (i.e. the $m$th component of $\boldsymbol{y}_i$) and multiply $2\sigma'_{L+1}\left(A_m^{(L+1)}\right)$, which we also already computed in the forward pass. Then, in the subsequent steps, we take the information that we just computed, $\delta_m^{(l)}$, and transport it one layer to the left to $\delta_m^{(l-1)}$ by only using information from the immediately neighboring layers.

Over time, hence by propagating from right to left in the network, we collect all necessary $\delta_v^{(l)}$s and have thereby computed all required inputs for the gradient evaluation.

Let us prove this.

Proof

Taking the $L_2$ loss, we have using the known definitions

$$L_2\left(\boldsymbol{Z}^{(L+1)}, \boldsymbol{y}_i\right) = \sum_{m=1}^{M_{L+1}} \left(Z_m^{(L+1)} - y_{im}\right)^2 = \sum_{m=1}^{M_{L+1}} \left(\sigma_{L+1}\left(A_m^{(L+1)}\right) - y_{im}\right)^2 .$$

First, we evaluate $\delta_m^{(L+1)}$ by

$$\begin{aligned}
\delta_m^{(L+1)} = \frac{\partial L_2}{\partial A_m^{(L+1)}} &= \frac{\partial}{\partial A_m^{(L+1)}} \left( \sum_{n=1}^{M_{L+1}} \left(\sigma_{L+1}\left(A_n^{(L+1)}\right) - y_{in}\right)^2 \right) \\
&= \frac{\partial}{\partial A_m^{(L+1)}} \left( \left(\sigma_{L+1}\left(A_m^{(L+1)}\right) - y_{im}\right)^2 \right) \\
&= 2\left( \left(\sigma_{L+1}\left(A_m^{(L+1)}\right) - y_{im}\right) \right) \left( \sigma'_{L+1}\left(A_m^{(L+1)}\right) \right) \\
&= 2\left(Z_m^{(L+1)} - y_{im}\right) \left( \sigma'_{L+1}\left(A_m^{(L+1)}\right) \right) .
\end{aligned}$$

From the first to the second line, we observe that the derivative with respect to $A_m^{(L+1)}$ cancels away all terms that depend on $A_n^{(L+1)}$ with $n \neq m$. The remainder is just the chain rule and the use of the definition of $Z_m^{(L+1)}$.

Now, we prove the second part of the statement for $l = L+1, \ldots, 2$. By definition, we have

$$\delta_v^{(l-1)} = \frac{\partial L_2}{\partial A_v^{(l-1)}} .$$

Similar to the proof of Lemma 11.2, we now understand $L_2$ as a function in the variables $\boldsymbol{A}^{(l-1)}$. Then, we split this function up and consider the evaluation of the loss as a concatenation of two functions, where the first function computes the variables $\boldsymbol{A}^{(l)}$ from the $\boldsymbol{A}^{(l-1)}$ and the second one computes $L_2$ from $\boldsymbol{A}^{(l)}$. This enables us to express the partial derivative $\frac{\partial L_2}{\partial A_v^{(l-1)}}$ via the chain rule as

$$\delta_v^{(l-1)} = \frac{\partial L_2}{\partial A_v^{(l-1)}} = \sum_{m=1}^{M_l} \frac{\partial L_2}{\partial A_m^{(l)}} \cdot \frac{\partial A_m^{(l)}}{\partial A_v^{(l-1)}} = \sum_{m=1}^{M_l} \delta_m^{(l)} \cdot \frac{\partial A_m^{(l)}}{\partial A_v^{(l-1)}} ,$$

which already gives us a recursive definition for the $\delta_v^{(l-1)}$. However, we still need to evaluate the last factor further. This is done by

$$\frac{\partial A_m^{(l)}}{\partial A_v^{(l-1)}} = \frac{\partial}{\partial A_v^{(l-1)}} \sum_{u=1}^{M_{l-1}} \omega_{mu}^{(l)} Z_u^{(l-1)} = \frac{\partial}{\partial A_v^{(l-1)}} \sum_{u=1}^{M_{l-1}} \omega_{mu}^{(l)} \sigma_{l-1}\left(A_u^{(l-1)}\right) = \omega_{mv}^{(l)} \sigma'_{l-1}\left(A_v^{(l-1)}\right) .$$

Combining this last equation with the previous equation gives the statement. □

Looking carefully into the above proof, one sees that indeed the choice of the loss function only affects the evaluation of $\delta_m^{(L+1)}$. Hence, we just need to adapt that part for other losses.

Even though the above result looks rather technical, it can rather easily be implemented, as we can express the equations in Theorem 11.1 by simple linear algebra operations, as given in

**Corollary 11.1** Let the setting of Theorem 11.1 be given. The vectors $\boldsymbol{\delta}^{(l)} := \left(\delta_1^{(l)}, \dots, \delta_{M_l}^{(l)}\right)^\top$, can be recursively computed via

$$\boldsymbol{\delta}^{(l-1)} = \mathcal{S}^{(l-1)} {\mathcal{W}^{(l)}}^\top \boldsymbol{\delta}^{(l)} ,$$

with $\mathcal{S}^{(l)} = \operatorname{diag}\left(\sigma_l'\left(A_1^{(l)}\right), \dots, \sigma_l'\left(A_{M_l}^{(l)}\right)\right)$ for $l = 1, \dots, L+1$, and the starting value

$$\boldsymbol{\delta}^{(L+1)} = 2\mathcal{S}^{(L+1)}(\boldsymbol{Z}^{(L+1)} - \boldsymbol{y}_i) .$$

There is no need to prove this, as it is just rewriting the equations in vector format. Combining all the above results, we obtain the part of the backpropagation algorithm that computes the necessary gradient of the loss in

**Algorithm 19** ($L_2$ loss gradient evaluation by backpropagation)
**input:** MLP given by $\left(L_2, \{M_l\}_{l=0}^{L+1}, \{\mathcal{W}^{(l)}\}_{l=1}^{L+1}, \{\boldsymbol{\sigma}_l\}_{l=1}^{L+1}\right)$, $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ training sample
**output:** gradient $\left(\frac{\partial L_2}{\partial \omega_{mv}^{(l)}}\right)_{l=1,\dots,L+1, m=1,\dots,M_l, v=1,\dots,M_{l-1}}$

1. $\boldsymbol{z}^{(0)} \leftarrow \boldsymbol{x}_i$
2. for $l = 1, \dots, L+1$:
   a) $\boldsymbol{a}^{(l)} \leftarrow \mathcal{W}^{(l)} \boldsymbol{z}^{(l-1)}$
   b) $\boldsymbol{z}^{(l)} \leftarrow \boldsymbol{\sigma}_l(\boldsymbol{a}^{(l)})$
   c) build $\mathcal{S}^{(l)}$ according to Corollary 11.1
3. $\boldsymbol{\delta}^{(L+1)} \leftarrow 2\mathcal{S}^{(L+1)}(\boldsymbol{z}^{(L+1)} - \boldsymbol{y}_i)$.
4. for $l = L+1, \dots, 2$:
   a) $\boldsymbol{\delta}^{(l-1)} \leftarrow \mathcal{S}^{(l-1)} {\mathcal{W}^{(l)}}^\top \boldsymbol{\delta}^{(l)}$
5. for $l = 1, \dots, L+1, m = 1, \dots, M_l, v = 1, \dots, M_{l-1}$:
   a) $\frac{\partial L}{\partial \omega_{mv}^{(l)}} = \delta_m^{(l)} Z_v^{(l-1)}$
6. return $\left(\frac{\partial L}{\partial \omega_{mv}^{(l)}}\right)_{l=1,\dots,L+1, m=1,\dots,M_l, v=1,\dots,M_{l-1}}$

Together with (stochastic) gradient descent, this establishes the *backpropagation algorithm* that is used to train the MLP.

# 12. Deep learning

In this final chapter, we bring the rather theoretical observations on artificial neural networks and their training into actual applications. At the same time, the titel of this chapter intentionally reflects the wording that is recently used for machine learning based on artificial neural networks. However, what is *deep learning*?

Before we attempt to give an answer, we need to try to structure the notions of *artificial neural network*, *feed-forward neural network*, and *multilayer perceptron*. Indeed the artificial neural network is the most general concept. It, roughly speaking, describes all parametric models that can be expressed in a network representation. The feed-forward neural network is a specific artificial neural network that, as discussed before, does not allow to have circles in the network. Finally, the multilayer perceptron, is a special type of FFNN.

While all FFNNs have an input and an output layer, the MLP has hidden layers with units / neurons / variables that are specifically connected to *all* units of the previous layer. Therefore, in context of more general FFNNs, the hidden layers of MLPs are called *fully connected layers*. Hence, a FFNN with only fully connected layers is an MLP.

Coming back to the notion of *deep learning*, this notion is mostly connected to the development of FFNNs with several or even many hidden layers that are of various types. Some of these layers are fully connected layers. However, depending on the application, there are also other types of layers.

This chapter starts by discussing the training of feed-forward neural networks from a practical perspective. We will not only give examples with data, but will have a look a the most common issues and practical solutions to them, which include the introduction of further neural network layer types. The second section of this chapter is dedicated to FFNN models, often called *convolutional neural networks*, that contain convolutional and other layer types. This flavor of networks has been initially mainly used for inputs that are images, however is now used in wide spread applications.

## 12.1 Practical training of neural networks

In this section, we will be discussing practical aspects of the training of a feed-forward neural network. We start with

**Remark** (Training of FFNNs in a nutshell) We try to outline the different ingredients that we need for the training of a FFNN.

**Network architecture** We start with some given FFNN, which is expressed in terms of its *architecture*, i.e. the number of layers, the type of layers, activations, etc. Once, this is all fixed, we run the backpropagation algorithm, discussed in Section 11.2 for a loss that we have chosen appropriately.

**Minimizer** In the backpropagation algorithm we have to choose a minimizer for the cost functional $J(\boldsymbol{\omega})$. A classical choice for the minimizer is the mini-batch gradient descent, which we know already. However, there are also other methods that, depending on the network, might have a higher learning speed. An example for this is the *ADAM minimizer* []. Most of these minimizers can also run in "mini-batch mode".

**Mini-batch size** We need to choose the mini-batch size. The choice of this size is sensible as it has to incorporate the (parallelization) abilities of the underlying hardware,[1)] and the compromise between learning speed and the advantages of stochastic choices of new descent steps.

**Learning rate** Similarly, as discussed in Chapter **??**, the learning rate is an important parameter to choose. Implementations of neural network training often provide several different semi-automatic strategies to choose the learning rate.

**Initialization of weights** Only if the initial setting for the weights in the backpropagation algorithm is properly chosen, the minimizers will be able to easily find a good (local) optimum for the loss.

If all these parameters are properly chosen, then we usually use some implementation (e.g. *Tensorflow / Keras*, *PyTorch*) to carry out the training of the given network. △

Note that the selection of the above parameters is no exact science. Instead, they heavily depend on the underlying model, which again strongly depends on the given data and the given supervised learning task. Hence, it is common practice to start into the development of new FFNN architectures and the selection of parameters in the training by starting from existing, documented, networks and parameter choices. Hence, there is no "best choice" for these settings, but rather some common sense. Moreover, there are some "tricks" for common issues in training in order to get good training performance. We will try to discuss some of these "tricks" using Example 11.4 from the previous chapter.

Let us now to give one of the common issues in FFNN training by

**Remark** (Vanishing or exploding gradients problem) A typical issue in training of artificial neural networks is the *vanishing / exploding gradients problem*. On the one side, it can happen that gradients can get smaller and smaller when being backpropagated through the layers. In worst case, then the first layers do not receive any update during the backpropagation algorithm, which is why training stagnates. On the other side, gradients

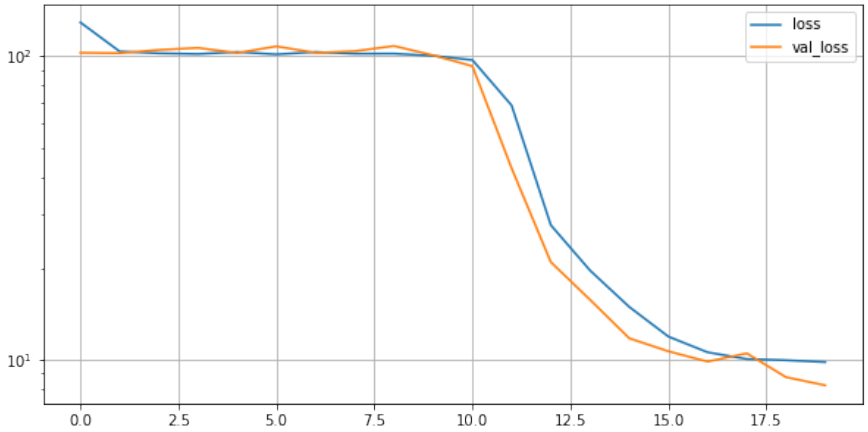
[1)] Usually, the data of one batch is parallely processed by the hardware.

can “explode” during training, i.e. quickly become extremely large, e.g. due to a too large learning rate.

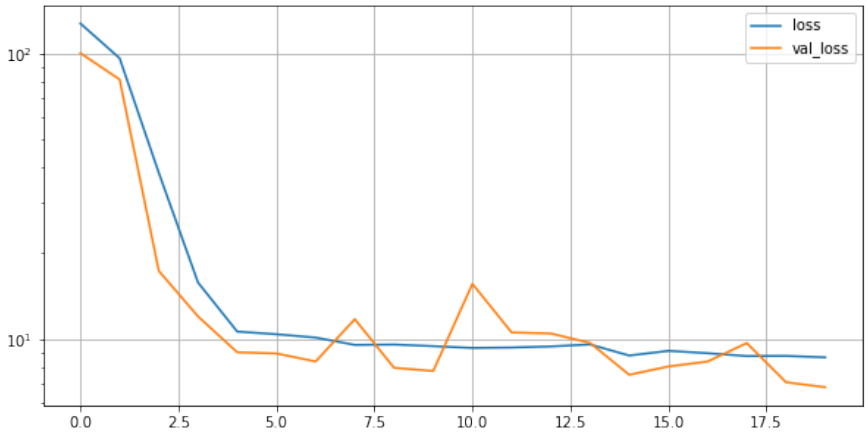
An exemplary reason for the vanishing issue is, e.g., present for the sigmoid activation. That activation saturates for large input values, such that indeed the derivative, thus the gradient, becomes almost or even identical to zero. △

One typical solution to the issue of vanishing gradients due to, e.g., the sigmoid activation is to make sure that an adapted initialization for the weights is used, such that the inputs to the activation do not get too large, leading to a vanishing gradient. We discuss this in

**Example 12.1** (Choice of weight initialization) We repeat the training experiment from Example 11.4. However, this time, we choose the sigmoid activation instead of the ReLu activation and initialize all weights using a normal distribution. The training result of this architecture choice and initialization is given below.




Compared to the previous example, we obtain a much worse training result. Then, we replace the choice of the normally distributed weights, by an optimized weights choice following the approach of Glorot et. al []. The results are given below.



Compared to training result for the normally distributed weights, the now achieve training result is better, while still not as good as the one for the ReLu activation.

This example including the shown figures can be reproduced and modified here: launch binder △

Often, an appropriate *initialization* of the weights might not be enough to solve the vanishing / exploding gradients problem. Therefore, another approach is to "learn" the optimal scale and offset of the inputs to e.g. a fully connected layer. This is done using a *batch normalization layer*.

**Definition 12.1** Given the terminology from Definition 11.4, the **batch normalization layer** considers training data given in a batch $\mathcal{T}_b = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{N_b}$ of size $N_b$ and computes from a previous layer of size $M_{l-1}$ with variables $Z_m^{(l-1)}$ new $M_l = M_{l-1}$ zero-centered and normalized variables $Z_m^{(l)}$ such that

$$Z_m^{(l)} = \gamma_m^{(l)} \cdot A_m^{(l)} + \beta_m^{(l)}, \qquad \text{with } m = 1, \dots, M_l\,.$$

Here, the $\gamma_m^{(l)}$ and $\beta_m^{(l)}$ are trainable parameters for a scaling and a shift of the data coming from the previous layer. Moreover, $A_m^{(l)}$ is the zero-centered and normalized input $Z_m^{(l-1)}$, which is computed by

$$A_m^{(l)} = \frac{Z_m^{(l-1)} - \mu_m}{\sqrt{\sigma_m^2 + \varepsilon}}, \qquad \text{with } m = 1, \dots, M_l\,.$$

To obtain the necessary mean $\mu_m$ and standard deviation $\sigma_m^2$, the artificial neural network model is evaluated on the training set batch $\mathcal{T}_b$, such that we obtain for each training input sample $\boldsymbol{x}_i$ in $\mathcal{T}_b$ the variables $Z_m^{(l),i}$ from the $(l-1)$th layer and compute

$$\mu_m = \frac{1}{N_b} \sum_{i=1}^{N_b} Z_m^{(l-1),i}, \qquad \text{with } m = 1, \dots, M_l\,,$$

$$\sigma_m^2 = \frac{1}{N_b} \sum_{i=1}^{N_b} \left(Z_m^{(l-1),i} - \mu_m\right)^2, \qquad \text{with } m = 1, \dots, M_l\,,$$

In the definition of $A_m^{(l)}$, $\varepsilon$ is a small number, usually $\varepsilon \approx 10^{-5}$ to assure that no division by zero can happen.

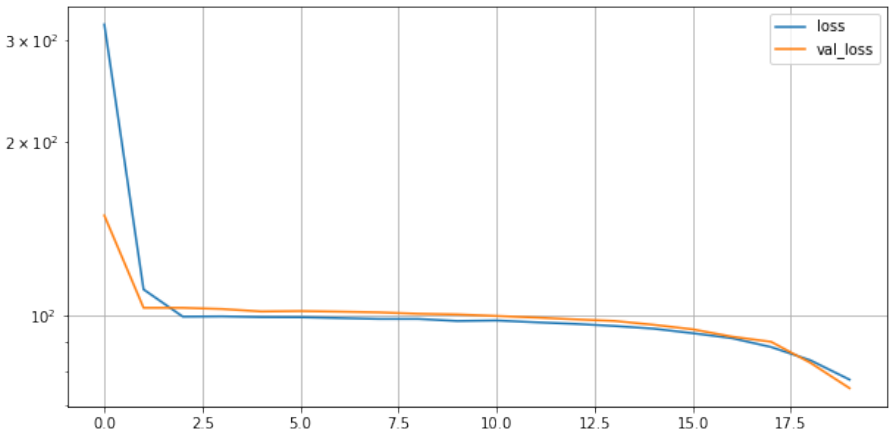
To summarize, the batch normalization layer first zero-centers and normalizes the output of the previous layer and then applies a trainable scaling and shift to it. The trainable parameters $\gamma_m^{(l)}, \beta_m^{(l)}$ with $m = 1, \dots, M_l$ are obtained by incorporating them as further free parameters in the backpropagation algorithm. In order to evaluate a FFNN with a batch normalization layer, the normalizations $\sigma_m^2$ and shifts $\mu_m$ are stored during the training.

While the consideration of the normalization over a given batch of inputs seems to be a surprising coupling with the training method, typical implementations of the backpropagation algorithm always compute all variables / units for a batch of training data, such that this information is typically present in a given backpropagation code.

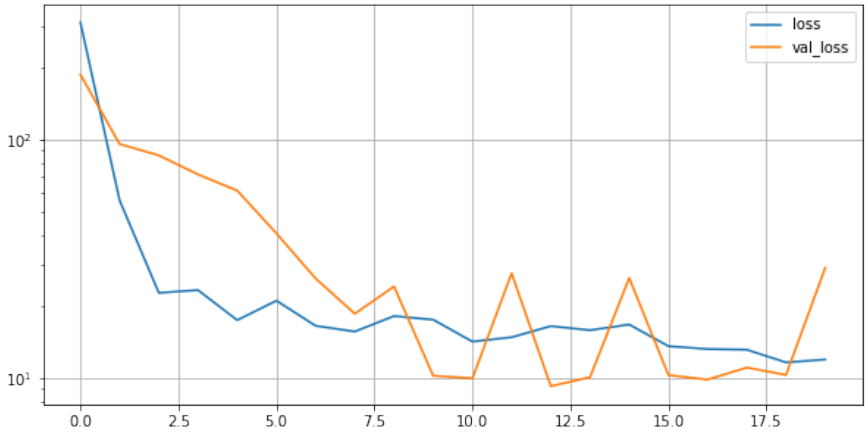
It has been shown in the literature [] that this approach can stabilize the training process specifically for "deeper" FFNNs. Due to the stronger stabilization, i.e. avoidance of gradient explosion, one can choose larger batch sizes or larger learning rates. This tends to speed up the training process in terms of the training error decrease with respect to the number of epochs. However, at the same time, adding batch normalization layers also makes the training, per epoch, more expensive.

We give an example for the use of batch normalization layers in

**Example 12.2** (Use of batch normalization layer) This example starts form the previous Example 12.1, with sigmoid activation and the weight initialization following []. However, instead of using stochastic gradient descent, we use mini-batch gradient descent with a mini-batch size of $N_b = 10$. If we follow this approach, we get obtain the following training result.




Quite obviously, after 20 epochs of the training process, the training and generalization error are still quite high. In a modified version of the ANN model, we add a batch normalization layer before each hidden (fully connected) layer. The training result is given below.




We clearly observe that the training is much faster and reaches a much better error level after 20 epochs.

This example including the shown figures can be reproduced and modified here: launch binder △

While the success of feed-forward neural network models also goes back to their very high model complexity with often tens of thousands to millions of parameters, the very high model complexity and thereby overfitting is one of the big issues of FFNN models. The following remark summarizes a few approaches to overcome overfitting for FFNNs.

**Remark** (How to overcome overfitting in FFNN models?) We briefly discuss three solutions to deal with overfitting of FFNN models: The first one is *early stopping* in training. It goes back to the mostly empirical observation that overfitting starts to establish for higher number of epochs in the application of the backpropagation process. Therefore, one might stop the training of a FFNN model at some earlier point to avoid running into the overfitting issue.

The second measure against overfitting is already known from Section 10.2. By using Tikhonov / $l_2$ regularization or *weight decay*, as it is called in context of FFNN models, we can reduce overfitting.

The third, very heavily used approach against overfitting is the addition of *dropout layers*. A dropout layer is simply given by

$$Z_m^{(l)} = \begin{cases} 0 & \text{with probability } p, \\ \frac{Z_m^{(l-1)}}{1-p} & \text{with probability } (1-p) \end{cases}, \qquad \text{with } m = 1, \dots, M_l\,,$$

where $p$ is a user-provided parameter of the layer. Essentially, a dropout layer drops in each training step with probability $p$ some of the units and rescales the remaining ones appropriately. In the model evaluation, dropout layers are ignored.

It has been shown in practice [] that this approach leads to an extreme stabilization of the learning process and sometimes shows very strong generalization error improvements. △

Certainly, there would be way more to say about practical considerations in FFNN model training. However, as this field is quickly evolving, the reader is referred to recent publications in the field to get the latest "best practice" considerations.

## 12.2 Convolutional Neural Networks

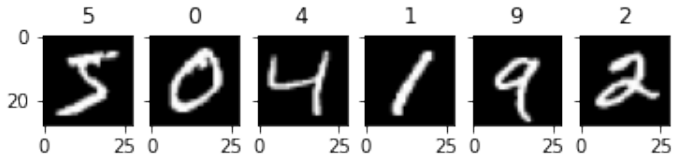
*Convolutional neural networks* (CNNs) are feed-forward neural networks with convolutional (and pooling) layers. This type of neural network has been initially introduced in context of image classification [], however is now used in wide spread applications. In the following, we will stick to the image classification use case, and will first introduce a naive approach to carry out image classification. Afterwards, we discuss the idea and concept of convolutional layers and typical architectures for image classification, which are then applied again to the initial example to show the improvement obtained by CNNs.

Before we give an example for image classification using an MLP, we need to make a

**Remark** (FFNNs for image data) Until now, we mostly considered a *layer* in a feed-forward neural network as a one-dimensional object. Hence, the units of a layer $l$ just form a *vector* $\boldsymbol{Z}^{(l)}$ of $M_l$ variables. In context of images, we will see that it is meaningful to extend this approach to a view, in which a layer is a two-dimensional object, where the individual units obtain two-dimensional indices, like $Z^{(l)}_{(m_1,m_2)}$. This allows to have a "neighborhood" concept for the different units, thus we can have units that are on the top of or to the left of other units, just as we have it for pixels in an image.[2)] △

Given this knowledge it is easier to understand

**Example 12.3** (Image classification by an MLP for the MNIST data set) We consider the MNIST data set for the classification of hand-written digits, as it has been discussed before in Examples 2.1 and 9.4. To apply an MLP for image classification to the data set, we first need to scale the grayscale values $[0, 255]$ to the range $[0, 1]$. A few samples of the training data are given below.

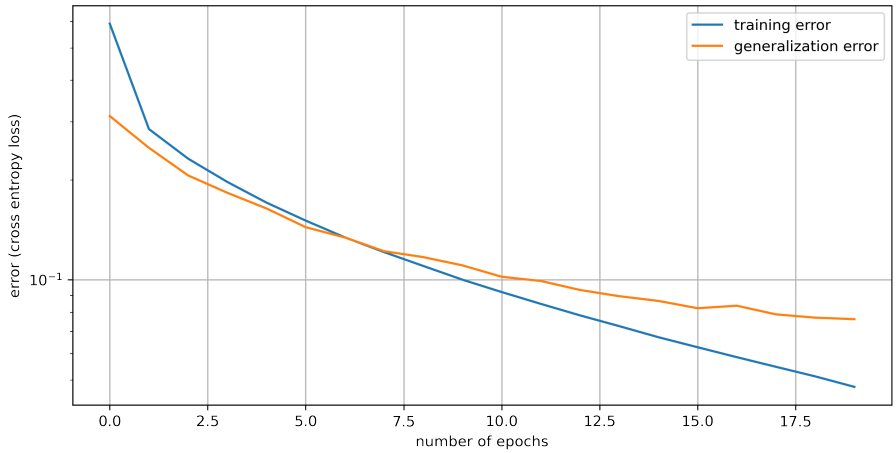



For the MNIST data set, we then consider the input layer of size $28 \times 28$. Since the fully connected layers, thus the hidden layers of the MLP, are still one-dimensional layers, we first need a *flatten layer* that just takes the two-dimensional input layer and "flattens"

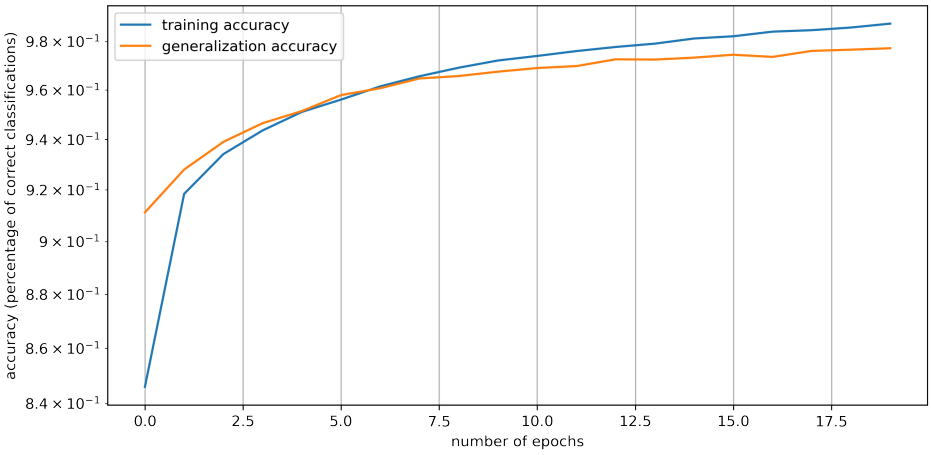
[2)] In practice, as we see soon, we often go for 3-dimensional or even $n$-dimensional layers, i.e. *order $n$ tensors*, depending on the data and layer type.

it out to a one-dimensional layer. Then, we apply a fully connected layer with 300 units and the ReLu activation and afterwards a fully connected layer with 100 units and again the ReLu activation. The output layer is then again a fully connected layer, however, with 10 units (for ten different classes $\{0, 1, \ldots, 9\}$) and the softmax activation.

For the training, we use the cross entropy loss and mini-batch gradient descent with a batch size of $N_b = 32$ and a learning rate of $10^{-3}$. The optimizer is run over 20 epochs. The generalization error, in addition to the training error, is monitored using the validation set approach. The result of the training process is given below.




Qualitatively speaking, we observe a decay in both the training and generalization error, while, over time, the model starts to overfit. As the cross entropy loss is not an easy to interpret measure, we also measure "accuracy" which is the error expressed as the ratio of correctly classified samples versus total number of samples. Ideally, we would want to have an accuracy of 1.0, which would correspond to 100% correct classifications. Below we give training and generalization accuracy from the same training run.




Looking into the details of the constructed model, we get a maximum of about 97.7 percent of correctly classified samples as generalization accuracy.

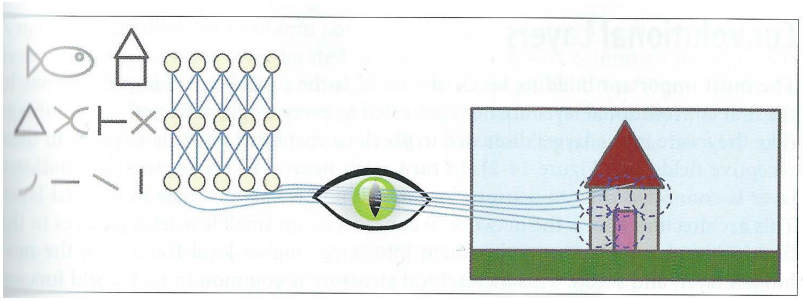
This example including the shown figures can be reproduced and modified here: launch binder △

On first sight, the result of 97.7% percent correctly classified hand-written digits sounds like a very good result. However, if one wants to apply such a classification model on, e.g., millions of ZIP codes on letters, the 2.3% badly classified digits can suddenly become a big issue.

Looking a bit into the history of artificial neural networks following [], indeed the search for a better classifier for the MNIST data set, above, has pushed the further development of FFNNs. Specifically, it has been observed that MLPs remain limited in their classification performance for image data. Finally, with the introduction of *convolutional neural networks*, the classification accuracy for the MNIST data set could be significantly improved.

CNNs go back to an observation about the way, humans "process" image information.

**Remark** (Human visual cortex) In the human visual cortex, i.e. that region of the brain that processes images, the neurons start by detecting patterns like horizontal lines, vertical lines, diagonal lines, lines crossing each other, etc. in small image patches. The availability or non-availability of a given pattern gives rise to features. Then, these features are, in several levels of re-combination combined into more and more complex features that finally represent complex objects, as shown below



[Géron]

△

*Convolutional layers* in a convolutional neural network try to mimic this feature extraction process. The idea, how this feature extraction process can be carried out goes back to the *discrete filtering* of images as it is done in image processing.

**Knowledge 12.1** (Convolutional filters) In image processing, a discrete *convolutional filter kernel* can be given by an $(2n+1)\times(2m+1)$ matrix $\mathcal{K} = (k_{ij})_{i=-n,\dots,n,j=-m,\dots m}$. For an image, that is understood as a matrix $\mathcal{D} \in \mathbb{R}^{N_1\times N_2}$, the filtered image $\mathcal{F} \in \mathbb{R}^{N_1\times N_2}$ is obtained by the discrete convolution

$$\mathcal{F} = \mathcal{K} * \mathcal{D}$$

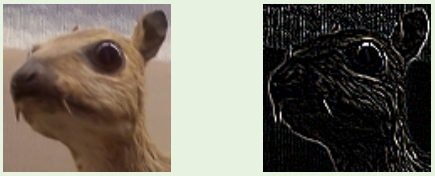
with

$$f_{n_1 n_2} = \sum_{i=-n}^{n} \sum_{j=-m}^{m} k_{ij} d_{n_1+i, n_2+j} \, .$$

To give an example, the filter kernel

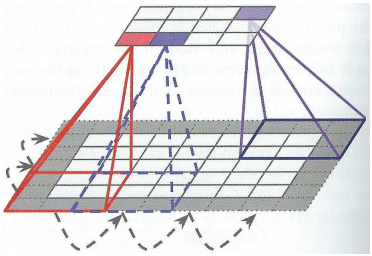
$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

applied to the below image on the left-hand side, can extract the edges given below on the right-hand side.



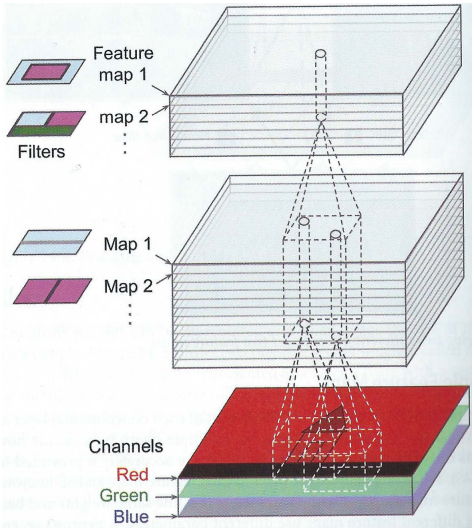
[Wikipedia]

Instead of manually constructing hundreds of filters that extract features from a given image, convolutional layers provide *trainable* filters. To realize this idea, a convolutional layer, in contrast to a fully connected layer, connects a given unit in the new layer only to a few neighboring units of the previous layer, thereby forming something like a *receptive field*, as can be seen below.



To actually get a filter, the convolutional layer moreover uses the identical set of weight for each of these receptive fields. Hence, the same weights are applied to small patches of the previous layer and form one unit in the new layer. These shared weights then take the role of the entries of the filter kernel in Knowledge 12.1. However, in contrast to the filter kernel there, these weights are trained form the data.

The just discussed idea leads to *one* filter. However, we are interested in many filters that are applied to one two-dimensional layer. Therefore, we "stack" many filters following the above construction on top of each other, obtaining *filter maps*, as shown below.




This leads to the following definition of a convolutional layer.

**Definition 12.2** (Convolutional layer) With the terminology from Definition 11.4, we consider units $Z^{(l)}_{m_1,m_2,k}$ in two dimensional layers at coordinates $(m_1, m_2)$. The third coordinate $k = 1, \ldots, N_f^{(l)}$ gives the $k$th feature map, hence the $k$th level in the stack. Then, the **convolutional layer** is given by

$$Z^{(l)}_{m_1,m_2,k} = b^{(l),k} + \sum_{i=0}^{r_1^{(l),k}-1} \sum_{j=0}^{r_2^{(l),k}-1} \sum_{k'=1}^{N_f^{(l-1)}} w_{i,j,k',k} Z^{(l-1)}_{v_1,v_2,k'}$$

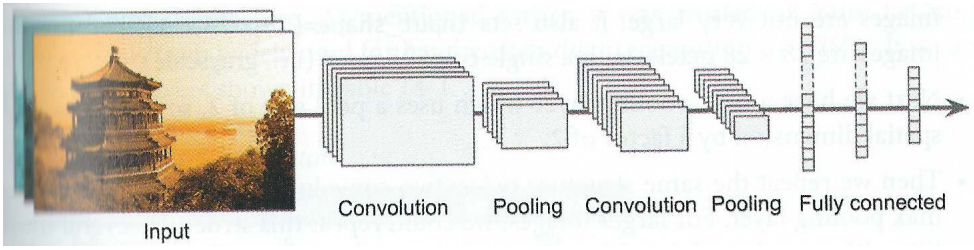
with $v_1 = m_1 s_1^{(l),k} + i$ and $v_2 = m_2 s_2^{(l),k} + j$. Here

- $N_f^{(l)}$ is the number of feature maps / filters in the $l$th layer,
- $r_1^{(l),k}, r_2^{(l),k}$ are the receptive field sizes in layer $l$ and map $k$,
- $s_1^{(k),k}, s_2^{(k),k}$ are the strides in layer $l$ and map $k$, and
- $b^{(l),k}$ is the intercept / bias of layer $l$ and map $k$.

In addition to convolutional layers, a typical CNN also has *pooling layers*. These layers are used to reduce the amount of units per layer, as the convolutional layers can introduce quite many units in a layer. A pooling layer follows the same ideas as a convolutional layer, however, it does not have weights but applies a reduction operation like maximum, minimum or average to the receptive field.

A typical CNN architecture is summarized in

**Remark** (Typical CNN architecture) A schematic view of a typical architecture of a convolutional neural network, mostly for image classification, is given in the below figure.




[Géron]

Hence, we always have a sequence of a convolutional layer and a pooling layer, which is maybe repeated several times. The convolutional layers are usually concluded by a ReLu activation. The pooling layers reduce the size in order to concentrate on larger features. These sequences of convolutional layers and pooling layers are followed by several fully connected layers.

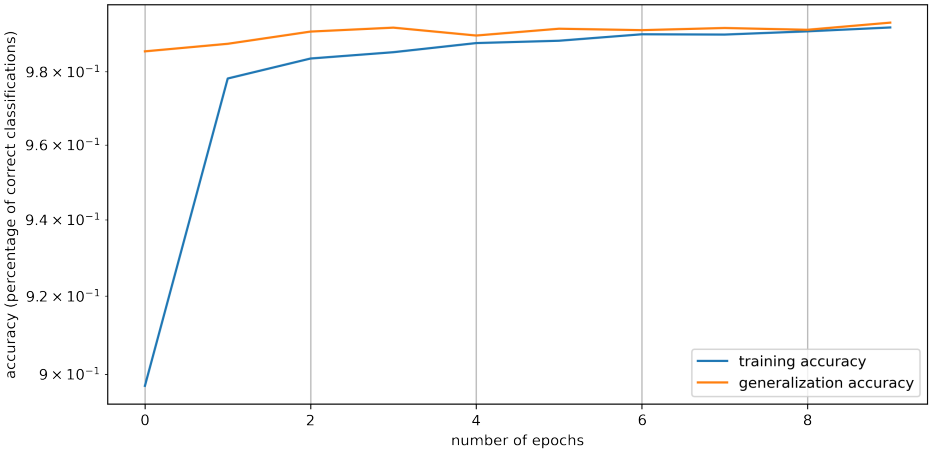
Trying to match this with our initial motivation from the human visual cortex, the convolutional and pooling layers provide a trainable feature extractor while the fully connected layers provide the classical machine learning model. △

Let us now apply a CNN to carry out image classification.

**Example 12.4** (Image classification by a CNN) We start from the same data as in Example 12.3. However, this time, we use a CNN model. The model follows the architecture that sequentially applies the following layers to the input layer:

- convolutional layer (receptive field size $7 \times 7$, ReLu activation, $N_f = 64$ filters
- max pooling layer (reducing layer size $28 \times 28 \times 64$ to $14 \times 14 \times 64$)
- convolutional layer (receptive field size $3 \times 3$, ReLu activation, $N_f = 128$ filters)
- convolutional layer (receptive field size $3 \times 3$, ReLu activation, $N_f = 128$ filters)
- max pooling layer (reducing layer size $14 \times 14 \times 128$ to $7 \times 7 \times 128$)
- convolutional layer (receptive field size $3 \times 3$, ReLu activation, $N_f = 256$ filters)
- convolutional layer (receptive field size $3 \times 3$, ReLu activation, $N_f = 256$ filters)
- max pooling layer (reducing layer size $7 \times 7 \times 256$ to $3 \times 3 \times 256$)
- flatten layer (giving a one-dimensional layer with 2304 units)
- fully connected layer (128 units, ReLu activation)
- dropout layer ($p = 0.5$)
- fully connected layer (64 units, ReLu activation)
- dropout layer ($p = 0.5$)
- fully connected layer (10 units, softmax activation)

Then, we train the model with a special form of the ADAM optimizer [], batch size $N_b = 32$ and let the training process run over 10 epochs. The training and generalization *accuracy* for this image classification model are reported below.




Looking into the details of the obtained training result, we achieve about 99.4% accuracy for the hand-written digit classification data.

This example including the shown figure can be reproduced and modified here: launch binder △

The above example is certainly only a very simple CNN for image classification. Current state of the art image classifiers can no longer be easily trained without a special hardware setup and look much more complex than the above example. In some cases, even further types of layers are introduced. Nevertheless, the general idea remains the same. With this, we would also like to conclude the discussion of "deep learning". Of course, there are many more applications and a continuously exploding amount of new contributions to this field. It is now up to the reader to use the knowledge gathered here, and start diving deeper into the field by reading state-of-the art research papers.

# Bibliography


[1] N. Balakrishnan, M. Koutras, and K. Politis. *Introduction to Probability: Models and Applications*. Wiley, 2019.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[3] A. DasGupta. *Fundamentals of Probability: A First Course*. Springer Texts in Statistics. Springer New York, 2010.

[4] A. DasGupta. *Probability for Statistics and Machine Learning: Fundamentals and Advanced Topics*. Springer Texts in Statistics. Springer New York, 2011.

[5] A. Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, CA, 2017.

[6] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[7] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space, 2013. arXiv preprint 1301.3781.

[8] M. Rupp, A. Tkatchenko, K.-R. Müller, and O. A. von Lilienfeld. Fast and accurate modeling of molecular atomization energies with machine learning. *Phys. Rev. Lett.*, 108:058301, Jan 2012.

[9] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press, 2014.

[10] A. Tsanas and A. Xifara. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and Buildings*, 49:560–567, 2012.