MARC DE KAMPS

MACHINE LEARNING (v1.0)

UNIVERSITY OF LEEDS

Contents

List of Figures

List of Tables

1 Introduction

1.1 Entry Level Requirements

In this module we assume that you are familiar with:

- *Linear Algebra.* You should be familiar with matrix vector manipulations. You should know what eigenvalues and eigenvectors are.
- *Basics of Statistics.* You should be aware of the concepts of probability, expectation, variance and the Gaussian distribution. Most of these concepts will be reviewed briefly.
- *Python Programming.* You should be familiar with Python data structures, functions and classes. You should know what numpy arrays are and be able to slice them.

1.2 Warning

The material presented here is intended to provide the theoretical framework for the later material. It is not suitable for the analysis of real world data without further reading. We have not discussed the need for more robust inference when data contains *outliers*. At the very least, you should consult section 7.4 of [1]. In Unit 4 you may find useful techniques to model outliers as a mixture of different processes.

[1] K. P. Murphy (2012). *Machine learning: a probabilistic perspective*. MIT press

1.3 Learning Outcomes for Unit 1

In this unit we will revise some of the basic notions of statistics. In particular you should be familiar with:

- Fundamental concepts of statistics: stochastic variables, probability distributions, probability densities.
- Joint and marginal distributions, Bayes' rule.
- Likelihood.

- Maximum likelihood estimation.

We will then go on to study a Bayesian interpretation of coin throws. You should be able to:

- Use the Bernoulli process as a model for coin or dice experiments.
- Explain the relationship between the binomial and the Bernoulli distribution.
- Give a Bayesian interpretation of a Bernoulli process using the Beta distribution.

2 Revision of Probability Concepts

2.1 Basic Concepts

2.1.1 Probability

Obligatory examples of stochastic processes are coin flipping and dice throwing. A coin lands on head or tails, these are the possible *outcomes*. The set of all outcomes is the *state space*. The state space formed by the outcome of a dice throw is $\{'1', '2', '3', '4', '5', '6'\}$, where $'1'$ stands for: 'the dice landed such that the side which has one marker is on top'. $'1'$ is shorter. Sometimes, the numerical value is relevant, e.g. in many board games, and the state space is considered to be the numbers $\{1, 2, 3, 4, 5, 6\}$ when there is no risk of confusing the numerical value with its actual realisation. The state space of the outcome of throwing two dice will be $\{1, \cdots, 36\}$, etc. State spaces can be *discrete* or *continuous*: the time of arrival for the next train at a platform can take any value between 1 ns to two years, at least in the UK.

Stochastic variables are variables representing potential outcomes of a probabilistic event, an event to which we attribute a certain amount of *uncertainty*. The process of realising this event is called a *stochastic process*. Upon the realisation of the event, the stochastic variable associated with the event has assumed as a value one of the possible elements of the state space associated with the stochastic process at the exclusion of all other possible elements. Stochastic variables are denoted by capital letters. In the context of a coin flip, the statement $P(X =' T') = 0.5$ usually means that the outcome of the next coin flip results in 'tails' is 50 %. Here X is shorthand for 'the outcome of a coin flip'. The context where a stochastic variable acquires its meaning must be specified very carefully. The outcome of a coin throw is called an *event*. Events are indicated by capital letters. So A can stand for: '6 came up when the dice was thrown', which is an event. The value of the outcome is a *realisation*, so in the previous example '6' is the realisation of the stochastic process of dice throwing for that particular event. A set of outcomes is called a *sample*, the set size is

called the *sample size*.

A *probability distribution* is a set of probabilities defined over a state space such that each element of state space has one probability associated with it. The sum of all probabilities must add to one. For a discrete distribution we can label the state space with an index i, and define the probability p_i associated with outcome i. We have:

1. $0 \leq p_i \leq 1$

2. $\sum_i p_i = 1$

If we generate or collect new events that are distributed according to some probability distribution, we are said to *to sample* the distribution.

The amount of uncertainty by which a particular outcome is realised is quantified by a numerical value, a *probability*. The probability of a particular outcome is a real number in the interval $[0, 1]$, where 0 represents absolute certainty that the outcome will not be realised and 1 represents absolute certainty of its realisation. The intrinsic meaning of such probabilities is subject of an ongoing debate.

The *frequentist* interpretation considers probability as the ratio of two numbers observed in a large number of repeated experiments. For example for a particular dice, one can throw a large number of times, say N times. The probability of the even '1' is approximately given by the number of times that '1' came up, divided by N. The frequentist imagines that this experiment in principle could be performed an arbitrary large number of times and that the observation of the ratio of outcomes would converge to a real number that would then constitute *the* probability. Although intuitively appealing, there are problems with this definition. A real dice might wear in the process of throwing, thereby slowly changing the propensity for '1' to come up, thereby creating tension between the mathematical process of taking the limit and the real world implementation of it. It also assumes that the experiment can be prepared identically without physical attrition of the dice. But a human thrower would tire, potentially subtly altering the probability of realising '1'. Maybe the human does not want to throw '1' and is somehow capable of influencing the odds slightly. Mechanical processes for throwing dice would suffer from wear and tear again. It turns out to be surprisingly hard to create a rigorous definition of the concept of probability. The frequentist approach dominates in high school teaching, possibly indicating that despite the difficulty in establishing a rigorous footing, the concept has intuitive appeal.

The *Bayesian* interpretation considers probability a quantitative measure of subjective believe. If I believe a coin is fair - and this

believe is subjective, because I may distrust the person who provided me with it -, then the probability for an outcome '1' is $\frac{1}{6}$. In the Bayesian view this *prior* belief can be modified in the face of experimental outcomes in a very specific way that we will discuss in detail in Sec. 3.0.6. Advocates of the Bayesian approach argue that some situations where we routinely assign probabilities to potential outcomes are not repeatable on principle. If I already have taken an exam, but am uncertain about its outcome, I may assign a probability that I passed it, say 30 %. This indicates a degree of pessimism, but the outcome is determined and the situation is clearly non repeatable because I now know the questions for this particular exam. I would probably do better on it next time, or if I were not allowed to prepare for it in the interest of frequentist purity I might have forgotten the subject matter. Probability here, a Bayesian would argue, is more the quantification of a subjectively held belief, rather than a number to determined by repeating the exam a large number of times, something that is not possible, even in principle. Moreover, the outcome is already determined. It is just that I don't know the outcome. Does it make sense to assign a number at all? Clearly, if I start to compare notes with other students I may realise that I actually performed better than I believed initially and my probability of 30 % is unreasonably low. So apparently, in the face of new information I can lower or raise this probability. Jaynes [1] gives a fascinating discussion of the Bayesian view point which argues that not only probabilities can be considered to be quantitative measures of subjective believe, but that by assumption of a number of reasonable, intuitively obvious rules for combining these believes one can arrive at the sum and product rule to be discussed below [2]. We will rely on these rules extensively and frequentists and Bayesians do not disagree on them so we will not pursue the Bayesian interpretation of these rules. However, anyone who wants to make a career in machine learning is recommended to get acquainted with these ideas at some point.

[1] E. T. Jaynes (2003). *Probability theory: The logic of science*. Cambridge University Press

[2] The first three chapters of Jaynes' book are available online at https://bayes.wustl.edu/etj/prob/book.pdf

2.1.2 *Expected Value and Variance for Discrete Variables*

If a function f(x) is defined over a discrete states space, i.e. for each x_i in the state space we know $f(x_i)$, then the *expectation value* is:

$$\mathbb{E}[f] = \sum_i p(x_i) f(x_i). \tag{2.1}$$

The *variance* is :

$$\text{var}[f] = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2], \tag{2.2}$$

which can also be written as:

$$\text{var}[f] = \mathbb{E}[f(x)^2] - \mathbb{E}^2[f(x)]$$

(you should be able to prove this).

The expected value indicates the centre of gravity of the probability distribution, the variance is a measure for how spread out the probability is. In general, the larger the variance, the less representative the expected value is for a typical event.

The n-th *moment* of a discrete distribution is given by:

$$\mu_n = \sum_i p(x_i) f(x_i)^n,$$

so

$$\mathbb{E}[f] = \mu_1$$

and

$$\text{var}[f] = \mu_2 - \mu_1^2$$

Higher moments give information about whether a distribution is skewed, and in general the more moments are known, the more accurate a distribution can be characterised. Some distributions, as we will see are only characterised by a few moments.

2.1.3 *Examples of Discrete Distributions*

We have already seen examples of distributions of discrete variables: dice throwing and coin tossing. The Bernoulli process can be considered as a model of throwing a coin. Its state space is the two-element set $\{0, 1\}$. The process is defined by:

$$\text{Ber}(x \mid \mu) = \mu^x (1-\mu)^{1-x}, \tag{2.3}$$

for $0 \leq \mu \leq 1$.

In one neat formula this expresses the fact that the probability for obtaining $x = 1$ is given by μ, and that the probability for $x = 0$ is given by $1 - \mu$. This process can be used to model a fair coin for $\mu = 0.5$ and one with bias for other values. You should be able to establish that:

$$\begin{aligned} \mathbb{E}[x] &= \mu \\ \text{var}[x] &= \mu(1-\mu) \end{aligned} \tag{2.4}$$

Another important distribution, closely related to the Bernoulli process, is the binomial distribution. It can emerge, for example, in

the context of a random walk. Assume that we start at position 0 and can move one step to the right with probability μ or stay in place with probability $1 - \mu$. Each individual step can be modelled by a Bernoulli process. A natural question to ask is where do we end after N steps. It is clear that the position must be somewhere between $-N$ and N, assuming that each step increments (decrements) our position coordinate by 1. What would the distribution look like? This position is modeled by a binomial process:

$$\text{Bin}(m, N \mid \mu) = \left(\begin{array}{c} N \\ m \end{array} \right) \mu^m (1-\mu)^{N-m}, \tag{2.5}$$

where

$$\left(\begin{array}{c} N \\ m \end{array} \right) = \frac{N!}{(N-m)!m!} \tag{2.6}$$

In order to end up at position m we need m moves to the right. Since we assume that moves are independent the probability of any sequence containing m moves to the right and $N - m$ non moves is simply $\mu^m(1-\mu)^{N-m}$. There are $\left(\begin{array}{c} N \\ m \end{array} \right)$ such moves as this is the number of ways that we can select m objects out of a set of N total objects (with replacement).

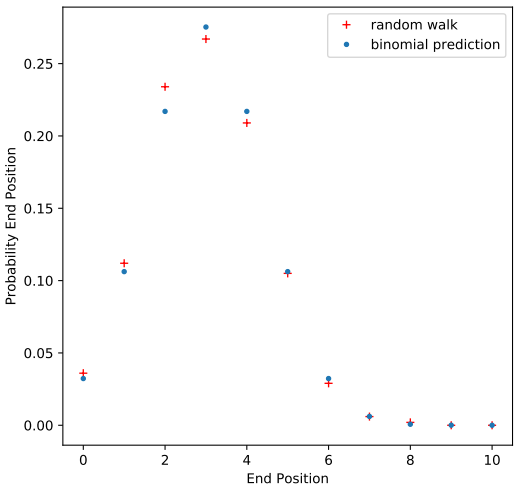


Figure 2.1: Example of a random walk as a binomial process. In total 10 steps are considered. For each step, there is a probability μ to move to the right and a probability 1 - μ to stay in place. This 10 step process is simulated with a Bernoulli process and the end position is recorded 1000 times. A histogram of the end positions is normalised so as to produce an observed probability density function ('random walk'). The probability density can also be calculated as a binomial process $\text{Bin}(10, 0.3)$. This distribution is also shown ('binomial prediction').

Loosely speaking, the binomial distribution emerges if we are interested in the distribution of the sum of a fixed number of outcomes of Bernoulli distribution events. Other examples are the sum of the outcome of two dice throws, which is essential in many board games.

2.2 Continuous Probability Distributions

The outcome of some stochastic processes can be in a continuous state space. The probability of a patient dying in the next 5 years is not restricted to discrete values, and a probabilistic statement about the adult height a child is likely to grow to obviously requires a continuous interval, something like $[0.5, 2.3]$. Probability density functions can be defined on complex sample spaces. For simplicity, here we will just consider a single connected interval in one dimension, which may be infinite, closed, open or half open. More complex situations will be discussed when needed. On this interval we will consider a *probability density function*. These functions are not necessarily smooth or even continuous. Their most important properties are:

1. $\int_a^b f(x)dx = 1$ for interval I, where $I = (a, b), [a, b), (a, b], or [a, b]$.
2. For every sub interval of I, I_0, $0 \leq \int_{I_0} f(x)dx \leq 1$.
3. Summed over all sub intervals, the probability must add to 1. We will remain deliberately vague about the term 'all sub intervals'. It can be well defined, but is highly technical.

The second condition implies that for all $x \in I, f(x) \geq 0$. It does *not* imply $f(x) < 1$. A probability density can attain arbitrarily large values, as long as its integral on any subinterval of I is less or equal to one.

2.2.1 Expectation and Variance of a Continuous Distribution

For a continuous probability distribution $p(x)$ defined on interval $I = [a, b]$, the expectation of an arbitrary function is:

$$\mathbb{E}[f(x)] = \int_a^b f(x)p(x)dx \tag{2.7}$$

The variance is defined as:

$$\text{var}[f(x)] = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]. \tag{2.8}$$

If the probability density function has a single peak, the value for which it occurs is the *mode* of the distribution. A population density function with a single peak is called *unimodal*. If more than one maximum occurs, the function is called *multimodal*.

The expectation value gives an indication where the bulk of the probability mass is residing. The variance gives an indication as to how widely the probability mass is distributed. For a distribution function with a single narrow peak, the variance will be small, for

a broader peak, which must be flatter as the total area under the curve must be equal to 1, the variance will be larger. also multimodal distributions tend to have larger variance than unimodal once for peaks of comparable width.

The expectation value and variance are simple expressions in the *moments* of the distribution function. The n-th moment, m_n is defined as:

$$m_n = \int (f(x))^n p(x) dx$$

2.2.2 *Example: Uniform distribution*

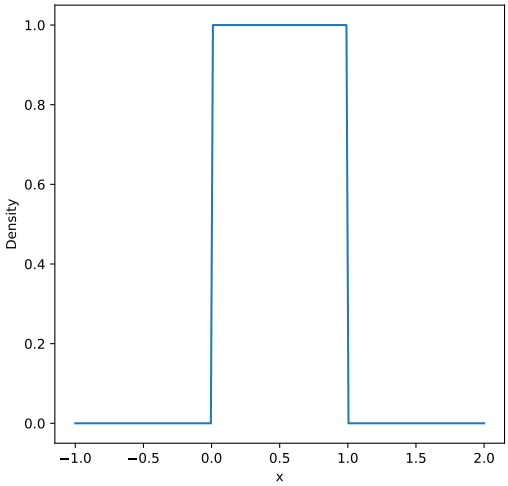


Figure 2.2: Top: the uniform distribution.

The function $f(x) = 1$ on $[0, 1]$ (Fig. 2.2 is a probability density function, expression the fact every number is equally likely to be sampled (technically with probability 0). It easy to check that is satisfies the two requirements given above. Some care has to be taken in interpreting probabilities compared to discrete probabilities. It makes sense to specify an interval and ask what the probability is that the next sample value will be in that interval. E.g. the probability to sample a number from the uniform distribution in the interval $[0.1, 0.2]$ is equal to 0.1. The probability to sample the number π exactly is 0.

It is not possible to write computer algorithms that generate truly random numbers, but a number of pseudo algorithms is known that generate sequences of numbers that *appear* random enough to evade tests that try to establish whether there is structure in the data that was generated. These algorithms simulate the uniform

distribution, so it plays a central role in simulation random processes. In the activity *Simulating Stochastic Processes*, you will experiment with how you can simulate arbitrary distributions based on *random number generators* that sample the uniform distribution.

2.2.3 *Example: Gaussian or Normal Distribution*

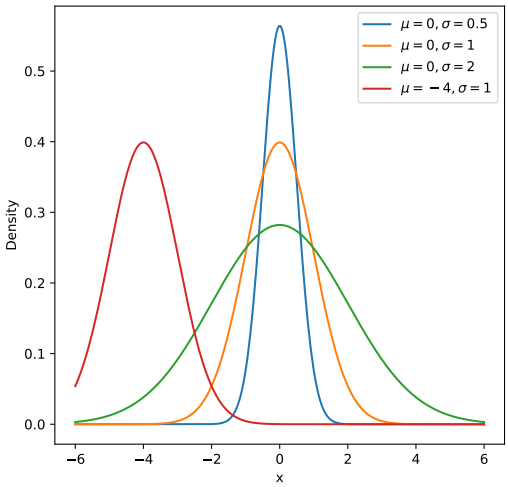


Figure 2.3: The Gaussian or normal distribution for several values of μ and σ.

Examples of the Gaussian distribution are shown in Fig 2.3. A closed formula exists:

$$\mathcal{N}(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \tag{2.9}$$

The distribution has a bell shape, which is wider or narrower dependent on parameter σ, the variance. Its mode (peak) is at $x = \mu$, the function is symmetric around this value. One can show that:

$$\begin{aligned} \mathbb{E}[\mathcal{N}(x \mid \mu, \sigma^2)] &= \mu \\ \text{var}[\mathcal{N}(x \mid \mu, \sigma^2)] &= \sigma^2 \end{aligned} \tag{2.10}$$

2.2.4 *Likelihood*

Many models of stochastic processes are *parameterised*. We have already seen several examples of parameterised distributions. For example, the Bernoulli process, which is governed by a parameter μ.

When $P(X \mid \mu)$ is considered to be a function of X we refer to it as a probability distribution. Probabilities sum to one:

$$\sum_i P(X = x_i \mid \mu) = 1,$$

where in this particular example $x_0 =' 0'$ and $x_1 =' 1'$. Probability distributions are *normalised*.

We will see that it is sometimes useful to consider this object to be a function of μ. It is then called a *likelihood*. An importance difference between probability and likelihood is that normalising the distribution with respect to the likelihood parameter usually does not make sense. In general, we would consider μ a continuous variable: $\mu \in [0, 1]$. There is no sensible way to define the normalisation of a likelihood and no need to do so.

An interesting question is now: 'Given a number of observed coin tosses, can I *infer* the parameter μ?' A direct question might be: 'is this coin fair'? If we can infer that the $\mu = 0.5$, the answer would be yes. Someone inclined to gamble and intent on maximising the profit might even go further and ask: 'can I predict future series of coin throws and thereby place my bets optimally?'. Such a person would be intent on *prediction* and might not even care about the particular value of μ or the question whether the coin is fair, but only about predictions that are as accurate as possible.

The question of how to infer parameters from data is central in machine learning and will be discussed throughout the module.

2.2.5 *Maximum Likelihood Estimation*

Maximum Likelihood Estimation (MLE) is the most widely used method to infer the parameters of a distribution when given a certain dataset. We will illustrate the technique with a number of examples.

Example: Bernoulli Process.

Imagine that we have observed a sequence of 0s and 1s, e.g. 000101010001, which we call the data set $\mathcal{D}$.

- We assume that these events have been independently generated by the same Bernoulli process (independently identically distributed; *iid*).

The probability for creating this particular sequence can be expressed in the unknown parameter, since the probability for each event is known and the events are independent. We find:

$$P(\mathcal{D} \mid \mu) = (1-\mu)^3\mu(1-\mu)\mu(1-\mu)\mu(1-\mu)^3\mu = (1-\mu)^8\mu^4$$

In MLE we consider the left hand side as a function of μ, i.e. as a likelihood and will look for the value of μ that maximises the likelihood associated with the observed data. To solve this problem, we

first note that the general case of a sequence of N data points with N_1 observations is hardly more complicated:

$$P(\mathcal{D} \mid \mu) = \mu^{N_1}(1-\mu)^{N-N_1} \tag{2.11}$$

We want to maximise this quantity. In order to do this we will take the logarithm, using the monotony of the logarithm: the value for μ that maximises the likelihood also maximises the log likelihood, which is a simpler expression.

$$\ln P(\mathcal{D} \mid \mu) = N_1 \ln \mu + (N - N_1) \ln(1-\mu)$$

We find the value of μ that maximises the likelihood, μ_{ML} by:

$$\frac{d \ln P(\mathcal{D} \mid \mu)}{d\mu} \mid_{\mu=\mu_{ML}} = 0$$

this works out as:

$$\frac{N_1}{\mu_{ML}} - \frac{N - N_1}{1 - \mu_{ML}} = 0,$$

solving for μ:

$$\mu_{ML} = \frac{N_1}{N} \tag{2.12}$$

The second derivative is:

$$-\frac{N_1}{\mu^2} - \frac{(N - N_1)}{(1-\mu)^2}$$

Since $N - N_1 > 0$ and $N > 0$, this term is negative. Therefore $\mu = \frac{N_1}{N}$ is a maximum of the log likelihood, moreover it is the only mnaximum. We conclude that this value of μ maximises the likelihood.

Intuitively, this estimate for μ makes sense. When the estimate is made on a large amount of data, it will approach the true value of μ, but the MLE approach is a point estimate for μ without any quantification of uncertainty. This is a substantial drawback when inference is done on small datasets. We will address this issue when we confront the MLE approach with Bayesian inference in Sec. 3.1.2.

3 Multivariate Distributions and Bayes' Theorem

'

3.0.1 Joint Probabilities

If we throw not one but two dice, the size of our state space increases from 6 to 36, as there are 36 different combinations which the pair of dice can realise. Note that here a distinction is being maintained between the combined events where '1' is the result of the first die and '2' the result of the second die, as opposed to a '2' for the first die and a '1' for the second die.

It then makes sense to take the Cartesian product of the two state spaces resulting in a new one that represents unique combinations of two events and consider a probability distribution over the combinations. When a probability distribution is defined over the potential outcome of a pair of events, rather than a single one, the probability distribution is called *bivariate*, whereas as probability distribution over potential outcomes of single events is called *univariate*.

A bivariate distribution is often represented as a matrix. A bivariate distribution is defined over a pair of events and the first dimension of the matrix represents the potential outcomes of the first event in the pair, the second dimension the potential outcomes of the second event in the pair.

Consider the Table 3.1, which lists the probabilities for every possible outcome when a pair of dice is thrown. The following can be observed:

- The entries of the table add up to 1, as they should because they represent probabilities.
- The probabilities are close but not quite equal to 0.02778, which is the decimal representation of $\frac{1}{36}$
- The table is not symmetric.

Dice 1 / Dice 2	'1'	'2'	'3'	'4'	'5'	'6'
'1'	0.0282	0.0282	0.0282	0.0286	0.0286	0.0282
'2'	0.0299	0.0299	0.0299	0.0302	0.0302	0.0299
'3'	0.0249	0.0249	0.0249	0.0252	0.0252	0.0249
'4'	0.0232	0.0232	0.0232	0.0235	0.0235	0.0232
'5'	0.0266	0.0266	0.0266	0.0269	0.0269	0.0266
'6'	0.0332	0.0332	0.0332	0.0336	0.0336	0.0332

Table 3.1: A summary of joint probabilities for the occurrence of pairs of outcomes when two dice are thrown.

Assume that:

$$\mathbf{p_1} = (0.17, 0.18, 0.15, 0.14, 0.16, 0.2)^T$$

and

$$\mathbf{p_2} = (0.166, 0.166, 0.166, 0.168, 0.168, 0.166)^T,$$

where $\mathbf{p_1}_1$, the first component of vector $\mathbf{p_1}$ is $P(X_1 =' 1')$, the probability of rolling dice 1 yielding a '1', etc. We can verify that the occurrence of both $P(X_1 =' i')$ and $P(X_2 =' j')$ is simply given by the product of these probabilities. We write:

$$P(X_1 =' i', X_2 =' j') = P(X_1 =' i')P(X_2 =' j')$$

.

Here $P(X_1 =' i', X_2 =' j')$ is called the *joint probability* of outcome $'i'$ for dice 1 and $'j'$ for dice 2. Intuitively, the fact that the joint probability is the product of the probabilities of the individual outcomes reflects the expectation that the outcome of throwing dice 1 is independent of that of dice 2. In general two events, say a with probability $P(a)$ and b with probability $P(b)$ are *independent* if the probability for their joint occurrence $P(a, b)$ is given by:

$$P(a, b) = P(a)P(b) \tag{3.1}$$

Two stochastic variables are independent if Eq. 3.1 holds for all possible values of their sample space. Note that although the table indeed reflects that rolling dice 1 does not influence the outcome of dice 2 - Eq. 3.1 is satisfied for this example - that the table is not symmetric: in general it is not true that $P(X_1 =' i', X_2 =' j')$ is equal to $P(X_1 =' j', X_2 =' i')$.

Not all bivariate probability distributions are composed from independent probabilities. Let X, Y be a stochastic processes, each with sample space $\{-1, 0, 1\}$. Let $P(X, Y)$ be the joint probability

distribution that assumes $p = \frac{1}{4}$ on four points $(-1,0),(1,0),(0,1)$ and $(0,-1)$. The stochastic variables X and Y are not independent, because, for example when we know that $X = -1$ that $Y = 0$. Independence would require that upon the realisation of X we still have no information about the realisation of Y.

Nor do in a bivariate process the sample spaces of X and Y have to be the same. They can contain different objects and be of different dimension.

Height \ Nationality	Zobany	Grabandan	Σ
$L < 150$	0.0068	0.0573	0.0641
$150 \leq L < 160$	0.0158	0.2074	0.2231
$160 \leq < 170$	0.0300	0.3285	0.3585
$170 \leq L < 180$	0.0371	0.2074	0.2445
$180 \leq L < 190$	0.0300	0.0520	0.0820
$190 \leq L < 200$	0.0158	0.0051	0.0209
$L > 200$	0.0068	0.0002	0.0070
Σ	0.1422	0.8578	1.0000

Table 3.2: The joint probability for encountering an individual of certain nationality and height.

3.0.2 *Marginal Distributions*

A joint probability that is specified over a number of stochastic variables represents all available information on their joint occurrence. Often we are asking questions about a subset of those variables. We may be interested in the distribution of heights, irrespective of nationality. Or we may want to infer the probability that we will encounter a Zobany national. These are questions about the *marginal distributions*

Given a joint distribution $P(X,Y)$, the marginal distribution over X is defined as:

$$P(X) = \sum_{Y} P(X,Y) \tag{3.2}$$

Here the sum is over all elements of the sample space of stochastic variable Y, so Eq. 3.2 is shorthand for:

$$P(X = a) = \sum_{b \in S_y} P(X = a, Y = b),$$

where S_y is the sample space of stochastic variable Y.

If we sample the distribution above, i.e. if we randomly select an individual of Zobany or Grabandan nationality, we may categorise the height of this individual and associate this with stochastic variable Y, which is defined over the height categories given in Tab. 3.2. Similarly we may associate the nationality with stochastic variable X which is defined over the two element set $\{Z, G\}$, where for convenience we have identified nationality with the first letter of its string representation.

In Tab. 3.2 the last row already gives the probability distribution over X as the summation over individual columns precisely corresponds to the definition Eq. 3.2.

Similarly, the distribution of height, irrespective of nationality, can be obtained by summing over the columns of a given height category, and the marginal distribution over Y is represented as the last column of Tab. 3.2.

Two important remarks about marginal distributions:

1. In general, It is not possible to reconstruct the joint probability distribution from the marginal distributions. Only when two stochastic processes are independent can one establish the joint distribution by multiplication. Marginalisation generally constitutes a significant loss of information.

2. The process of marginalisation as described here is unambiguous and simple to implement. Yet, one of the fundamental problems of machine learning is that marginalisation is hard. We will return to this issue when discussing continuous probability distributions, where it amounts to integration.

3.0.3 *Conditional Probabilities*

Another natural question to ask is: 'given that a person is a Zobany national, what is the probability distribution of their height?'. Or: 'given that someone is more than 2m tall, what is the probability that they are a Grabandan national?'. These are examples of conditional probabilities. Let us focus on the first question. Again identifying stochastic variable X with nationality and Y with height category, the question after the probability distribution of Y *given* that $X =' Z'$ (the individual is a Zobany national, which is written as:

$$P(Y = y_i | X =' Z'),$$

i.e. the probability that the person is in height category y_i, i.e. one of the height categories listed in Tab. 3.2.

This probability is clearly proportional to the numbers in the 'Zobany' column, but the numbers in this column do not specify a

probability distribution, as they do not add up to one. This gives a clue as to how to solve this problem. After all, the probability of being a Zobany national is smaller than one and we need to compensate for this. It is clear that

$$\sum_i P(X =' Z', Y = y_i) = P(X =' Z')$$

If we add all possibilities for someone to be of a certain height, whilst knowing that someone is Zobanian, we must have the probability that someone is Zobanian. A Zobanian must fall in *some* height category. So

$$\sum_i \frac{P(X =' Z', Y = y_i)}{P(X =' Z')} = 1$$

It is clear that the numbers:

$$P(Y = y_i \mid X =' Z') \equiv \frac{P(X =' Z', Y = y_i)}{P(X =' Z')} \tag{3.3}$$

satisfy all requirements for a proper probability distribution. The resulting distribution is the *probability distribution over the heights, conditioned on nationality.*

Similarly we can condition nationality on height, e.g ask the question 'What is the probability for an individual to be Zobanian when over 2m tall?'.

From Tab. 3.2 we see that the probability to be Zobanian and over 2m tall is 0.0068. The probability that someone is over 2m, given that we already know is the individual is Zobanian is 0.0068/0.1422 = 0.0478. Similarly, we can ask what the probability is for an individual known to be Grabandan to be over 2m, which is 0.00023. Conditioning gives us the possibility to compare like with like: we have compensated for the fact that the probability in general of meeting a Grabandan is much higher than meeting a Zobanian and there is a difference in height that is larger - for 2m tall individuals - than is apparent from a casual glance at Tab. 3.2.

3.0.4 *Joint vs Marginal and Condition Probabilities*

It is important to realise that conditional and marginal probabilities in general convey substantially less information than the joint probability. Figure 3.4 shows a joint distribution with a clear structure, but the marginal distributions broadly convey the extent of the data but not much more. It is certainly not possible to infer the structure of the joint distribution from the marginal ones.

Conditioning the joint probability on one or more variables is similarly reductive. Figure 3.5 shows the same joint probability distribution, alongside with two one dimensional distributions which are

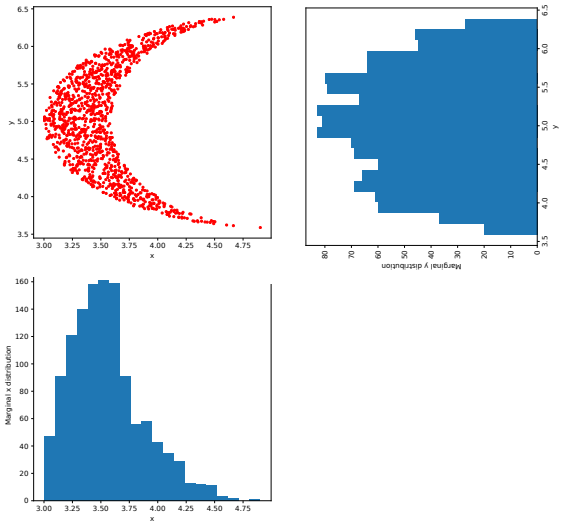


Figure 3.4: Joint probability distribution function in two variables x and y. The marginal distributions are shown as a histogram in the margins of the joint distribution.

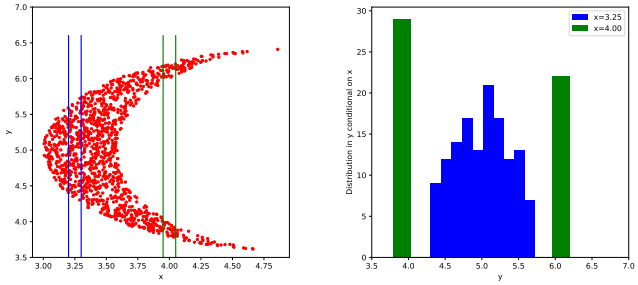


Figure 3.5: The same joint probability distribution conditioned on two different values of x. Each value of x results in a one dimensional conditional distribution which can be unimodal or bimodal.

obtained by conditioning the joint distribution on two different values of x. The shape and properties of these distributions are clearly strongly dependent on which value of x one chooses to condition on. It is also clear that is impossible to reconstruct the original distribution. For that one would need the conditional distributions for all values of x.

3.0.5 *The Sum Rule and Product Rule*

The concepts introduced in the previous section are key to statistics and therefore machine learning. They amount to two rules: The *sum rule*:

$$P(X) = \sum_{Y} p(X, Y) \tag{3.4}$$

The *product rule*

$$P(X,Y) = p(Y \mid X)P(X) \tag{3.5}$$

3.0.6 *Bayes' Rule*

Bayes' rule (sometimes called Bayes' law) relates conditional probabilities. Consider the product rule and observe that it may be written in two ways:

$$P(X,Y) = P(Y \mid X)P(X), \tag{3.6}$$

or:

$$P(X,Y) = P(X \mid Y)P(Y) \tag{3.7}$$

Inspecting Tab. 3.2 makes plausible that these decompositions are equally valid. It follows that:

$$P(X \mid Y) = \frac{P(Y \mid X)}{P(Y)}P(X) \tag{3.8}$$

or using the sum rule:

$$P(X \mid Y) = \frac{P(Y \mid X)}{\sum_X P(Y \mid X)P(X)}P(X) \tag{3.9}$$

Both Eqs. 3.8 and 3.9 are called Bayes' Rule, Bayes' Law (but not Baye's Rule).

On the one hand they reflect relatively trivial relationships between conditional probabilities. On the other hand they offer a tool that has had profound impact on the development of statistics in the late 20th and 21st century. In particular, a consistent and systematic use of this rule avoids pitfalls in the analysis of data. There are indications that we handle conditional probability well in everyday reasoning if we use them to express *causal* relationships, but seriously mishandle them if conditional probabilities do not express a causal relationship [1]. Now please consider *Unit1_Activity_Bayes* to see a real world example of the relevance of Bayes' rule.

In the next section, we will provide a very simple example of Bayesian analysis of a stochastic process.

[1] J. Pearl, M. Glymour, and N. P. Jewell (2016). *Causal inference in statistics: A primer*. John Wiley & Sons; and J. Pearl and D. Mackenzie (2018). *The book of why: the new science of cause and effect*. Basic books

3.1 *Bayesian Inference: an Unfair Coin*

3.1.1 *Introduction*

Imagine that we are predicting a series of coin tosses. Again, we model this with a Bernoulli process. A stochastic variable governed by a Bernoulli process has two possible outcomes: $\{0,1\}$ (we may associate 0 with 'head' and 1 with 'tails'). If $\mu = 0.5$ the coin is fair

and if we throw $N = 100$ times we may get a sequence of heads and tails. The individual elements of the throw may be unpredictable, but we expect approximately 50 heads and 50 tails. If we were to see 25 heads and 75 tails we would have a hard time believing $\mu = 0.5$ and might guess it is closer to $\mu = 0.25$, which is the MLE.

3.1.2 *Bayesian Approach*

The Bayesian approach requires us to make assumptions about the prior values of μ. In the Bayesian view, probability corresponds to a subjective degree of believe. In principle a coin can be weighted, and μ can take on any value. To simplify the situation, we assume that μ is discretised and can take on any of five values.

Since we know that μ can only assume five values, we must define a probability distribution over these values, the *prior* distribution. We could, for example, assume that our opponent is most likely to be fair, but allow for some probability that the coin is unfair in their advantage:

μ	$P_{prior}(\mu)$
0.	0.05
0.25	0.05
0.50	0.7
0.75	0.15
1.0	0.05

Table 3.3: The prior probabilities of a coin with 5 possible weighting values. It is most likely to be fair, but has small probabilities of being loaded.

Our objective is to reassess these probabilities in light of a sequence of coin tosses that we have observed. Bayes' rule gives us:

$$P(\mu = \mu_i \mid D) = \frac{P(D \mid \mu_i)}{P(D)} P_{prior}(\mu = \mu_i) \tag{3.10}$$

Here $P(D \mid \mu_i)$ is simply the likelihood of the data as defined above:

$$P(D \mid \mu_i) = \mu^{N_1}(1 - \mu_i)^{N - N_1},$$

given μ it gives the probability of the observation of a particular sequence, which can be summarised in the only two numbers that are relevant: the number of 0 outcomes, N_0, and the number of 1 outcomes N_1, from which we can compute $N = N_0 + N_1$. Given a sequence of observations, we have the numbers to compute the posterior probabilities $P(\mu \mid D)$.

Now assume that we have observed 67 'tails' out of a 100 throws. This gives us all the numbers we need: $N_0 = 23, N_1 = 67, N = 100$

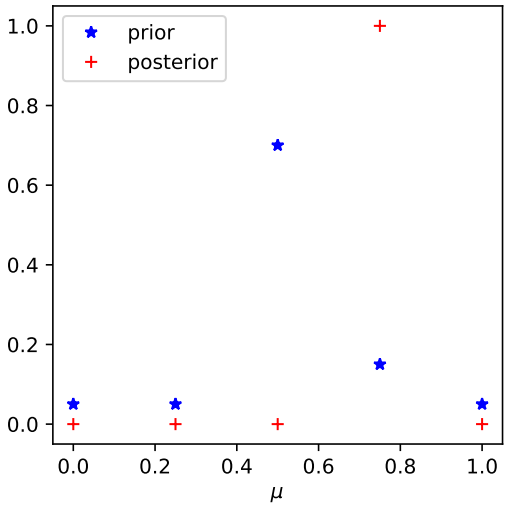


Figure 3.6: The prior and posterior over a discrete set of μ values. The prior distribution was by our own choosing, reflecting the fact that we believed that the coin is fair, but allowing for uncertainty. The posterior distribution results from a consequent application of Bayes rule. It has shifted markedly to the right and now peaks nearly at the true value. The peak is relatively narrow, reflecting that we are now relatively confident.

Figure 3.6 shows both the prior and posterior probability. In the generation of the data sequence, we actually used $\mu = 0.75$, i.e. the outcome of 75 tails would have been the most likely outcome. The posterior distribution peaks at closely this value and reflects that our estimate of μ has improved considerably. The peak is also narrower, reflecting that we are more confident in our outcome.

The calculation has been performed in a Jupyter notebook called *Unfair coins - Maximum Likelihood and Bayes*. You can experiment with different values of prior, number of throws and μ and observe the effect of these changes by running the notebook.

3.1.3 *The Complication of Normalisation*

Please note the following: it is relatively straightforward to calculate a given posterior probability for a single μ-value *up to a normalisation factor*:

$$P_{posterior}(\mu = 0.5) \sim P(D \mid \mu) P_{prior}(\mu = 0.5)$$

This is a lightweight calculation, given the simple form of the likelihood and that we know the prior. It is not apparent from this simple example that computationally the biggest problem is the normalisation: we have to calculate $P(D)$ in Eq. 3.10, and we can only do that by calculating $P(D \mid \mu_i)$ *for all* μ_i. Alternatively, we can forego normalisation at first, ignoring the normalisation factor $P(D)$, but then we have to normalise the resulting outcomes. Either way, we

have to go through the process of calculating *all* posterior outcomes, even if we're not interested in some of them! From this simple table example it will not be clear why this is a problem. We will address it again when we have discussed conditional probability distributions, but this problem is a major obstacle in the general application of Bayesian statistics!.

3.1.4 *Continuous Prior distributions*

In the example above, to illustrate the application of Bayes' rule to a table of conditional probabilities, we restricted the value of the μ variable to a number of discrete values. This demonstrates how simple the application of Bayes' rule really is, but a purist might complain that the maximum likelihood can take on any value, whereas the prior and posterior distribution are only defined on a small number of values, and therefore we are not comparing like with like.

The continuous version is not fundamentally different, although it requires continuous prior and posterior distributions over the interval $[0, 1]$, thereby covering all values take μ could take potentially. For continuous variables Bayes' rule is as follows:

$$p(\mu \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \mu)}{p(\mathcal{D})},$$

where

$$p(\mathcal{D}) = \int_0^1 p(\mathcal{D} \mid \mu) p(\mu) d\mu$$

The main difference is that we now have to pick a continuous prior distribution. We could pick many, bearing in mind that it is supposed to express a prior belief about the value of the coin. If we firmly believe that $\mu = 0.5$ we should pick a prior that is sharply peaked around that value. If we are not firm in our belief, we could take a broader peak. If we believe the coin if false, we can take a value closer to 0 or 1.

As a functional form for the prior we use Beta function:

$$\text{Beta}(\mu \mid a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \mu^{a-1} (1-\mu)^{b-1}$$

The Gamma functions are a normalisation factor, which we will not discuss in detail here, but can easily be calculated using numpy. For positive integer values of n, $\Gamma(n) = (n-1)!$.

The functional dependence is reminiscent of the likelihood. Remember that for N observations of which N_1 were '1' the likelihood is given by:

$$p(\mathcal{D} \mid \mu) = \mu^{N_1} (1-\mu)^{N-N_1}$$

Note the following:

1. The likelihood and the prior have a very similar form. This is deliberate: the computation of the posterior is very simple.

2. By picking variables a and b appropriately, we have great freedom in picking the shape of the prior. Helpful in this regard are the following relations:

$$\mathbb{E}(\mu) = \frac{a}{a+b} \tag{3.11}$$

$$\text{var}(\mu) = \frac{ab}{(a+b)^2(a+b+1)} \tag{3.12}$$

For example, the formula for the expectation shows that if you want your distribution to peak at $\mu = 0.5$, you have to pick $a = b$.

We show the great variety of shapes that can be picked by selecting a and b appropriately:

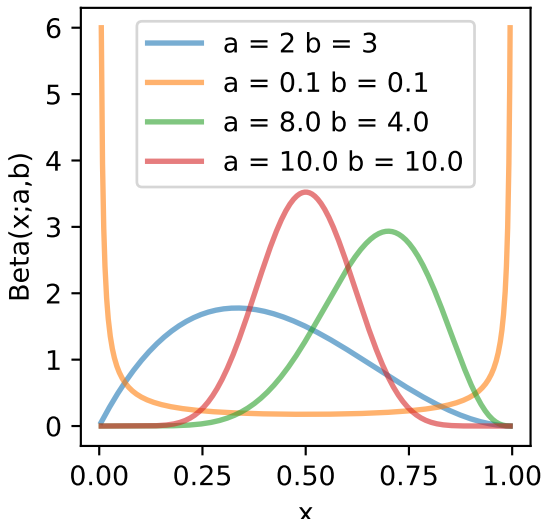


Figure 3.7: The Beta distribution for several values of a and b.

Applying Bayes' Theorem is remarkably straightforward: Multiplying likelihood and prior, we obtain the posterior:

$$p(\mu| \mid \mathcal{D}) \sim \mu^{N_1+a-1}(1-\mu)^{N-N_1+b-1}$$

Note:

1. The posterior is again a Beta function. The calculations shown here are greatly facilitated by a prior that plays nicely with the likelihood.

2. The normalisation factor that isn't shown here can easily be calculated:
$$\frac{\Gamma(N+a+b)}{\Gamma(N_1+a)\Gamma(N-N_1+b)}$$

3. Using the formula for the expectation we can calculate that:
$$\mathbb{E}[\mu]_{posterior} = \frac{N_1+a}{a+b+N}$$

4. In the limit of an infinite number of observations this is equal to $\frac{N_1}{N}$. In this limit the posterior will peak sharply around the maximum likelihood estimate.

5. For a finite amount of data, the expectation value of the posterior represents a compromise between the (expectation of) the prior distribution and the maximum likelihood estimate.

6. Since the parameters a and b essentially reflect the counts of the number of outcomes for $x = 1$, N_1 and the total number of observations, N, the choice of prior can here be seen to be equivalent to artificially picking a number of prior observations by the modeller.

7. In Unit 2 we will see that whilst the prior will generally be chosen such that prior times likelihood will result in the posterior having the same functional form as the prior, the prior usually does not have the same functional form as the likelihood.

We finish with a number of examples: We use a highly loaded coin $\mu = 0.75$ $N = 3$; prior $a = 10, b = 10$, which corresponds to a prior that peaks around $\mu = 0.5$, i.e. initially we believed that our coin was fair. The result is shown in Fig. 3.8 (top). We start with a fairly broad prior, centred around $\mu = 0.5$ reflecting a prior believe that the coin is balanced. After three observations, we see that this prior has shifted slightly towards higher values, but has not changed substantially. We simply have not made a sufficient number of observations. After 100 observations(bottom), the picture has changed markedly: the posterior distribution peaks close to its true mean and is sharp, reflecting a relatively high confidence in this value.

3.1.5 *Discussion*

It is important to compare the Bayesian analysis to maximum likelihood estimation. Consider that we have a true, but unknown value of $\mu = 0.5$ and that we are allowed to observe the result of three throws only. This could easily be three tails, forcing us to conclude, on the basis of MLE that $\mu = 1$. If we were to predict future events, we would have to conclude that they only can be tails. No human

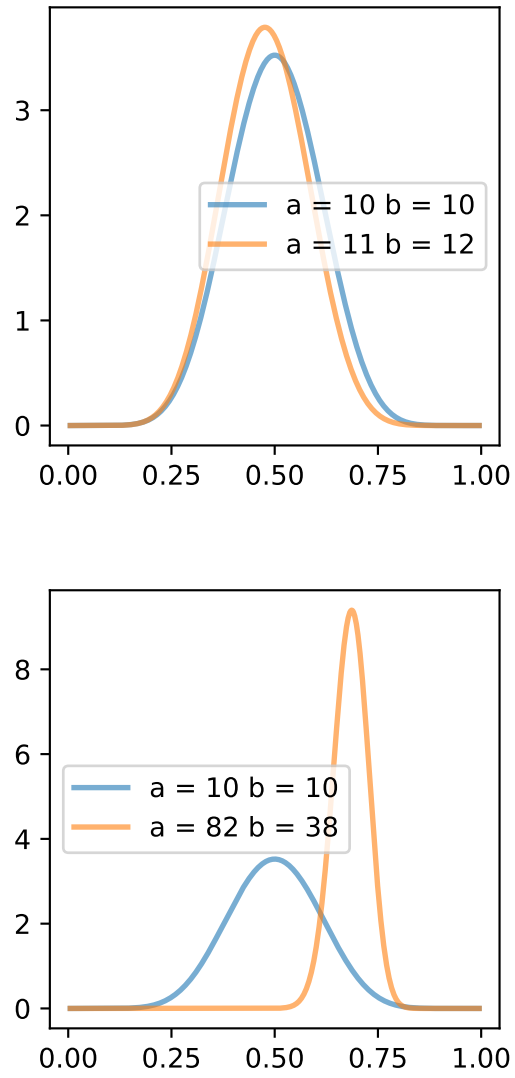


Figure 3.8: Prior and posterior distributions after three observations of a heavily loaded coin $\mu = 0.75$, starting from a relatively broad prior centred around $\mu = 0.5$. Top: $N = 3$ observations, bottom $N = 100$ observations.

observer would be confident in this: they would realise that three observations is not enough to base such a stark conclusion on.

The Bayesian framework allows the inclusion of this prior uncertainty in a way that MLE does not. One could do sequence prediction on the basis of three observations using the posterior distribution for $N = 3$ in Fig. 3.8 top, and most human observers would agree this is a far more reasonable procedure. Even after 100 observations there will be residual *model uncertainty*. In a consequent application of the Bayesian framework this is an extra source of randomness in the parameters that determine a process that is already stochastic. MLE on the hand produces a point value for these estimates.

A common procedure to try an estimate the uncertainty in MLE is *cross validation*. In k-fold cross validation, the dataset is broken into k partitions. Usually $k - 1$ of those are involved in training the model and 1 of them is used to evaluate the model, a process that can be repeated k times.

Bishop points out that the observation of three tails leads to *overfitting* the MLE [2], and a three-fold cross validation would make no difference at all here.

Finally, the issue of normalisation has been treated casually here. The hard work has been done by introducing the Gamma function, but you are recommended to try Exercise 2.5 in [3] to appreciate that getting the normalisation right is possible because of the nice analytic properties of the Beta function and that without such properties, or on an interval that is not $[0, 1]$ we would have to resort to numerical integration. In a one dimensional distribution this is not a problem, but in multivariate distributions this can become too expansive for practical purposes very rapidly.

[2] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

[3] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

4 The Central Limit Theorem and the Ubiquity of the Gaussian distribution

4.1 MLE of Gaussian Parameters

Assume that we have good reason to believe that a sample is generated from a Gaussian distribution with unknown parameters μ, σ. The likelihood for a single data point x_1 is given by:

$$p(x_1 \mid \mu, \sigma) = \mathcal{N}(x_1 \mid \mu, \sigma^2)$$

Again, if we assume that samples are generated *idd* (independently identically distributed), the likelihood for an entire dataset $\mathcal{D} = \{x_1, \cdots, x_N\}$ is:

$$P(\mathcal{D} \mid \mu, \sigma^2) = (\frac{1}{\sqrt{2\pi\sigma^2}})^N e^{-\frac{1}{2\sigma^2}(x_1-\mu)^2} \cdots e^{-\frac{1}{2\sigma^2}(x_N-\mu)^2}$$

This is a product of exponentials. Taking the logarithm reduces this to a relatively simple sum:

$$\ln(\mathcal{D} \mid \mu, \sigma^2) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i - \mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln 2\pi \tag{4.1}$$

Maximising Eq 4.1 with respect to μ gives:

$$\mu_{ML} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.2}$$

Maximising with respect to σ gives

$$\sigma^2_{ML} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_{ML})^2 \tag{4.3}$$

Given a set of N data points, which are random variables, μ_{ML} and σ_{ML} themselves are random variables. One can show that:

$$\mathbb{E}[\mu_{ML}] = \mu \tag{4.4}$$
$$\mathbb{E}[\sigma^2_{ML}] = (\frac{N-1}{N})\sigma^2$$

4.1.1 *The Central Limit Theorem*

It is remarkable that many quantities in nature follow an (approximate) Gaussian distribution. Height distribution often are bell shaped and can be modeled accurately with Gaussian distribution. Scores from IQ tests often are modelled by Gaussian distribution and if the mean and variance are known - by fitting a Gaussian to a histogram of a large number of such scores - it is possible to determine whether this score is exceptionally high or low, and quantify that, e.g. by giving a percentage of the population that this score exceeds.

The ubiquity of the Gaussian distribution can at least be partly explained with the *Central Limit Theorem*. Let $S_N = \sum_{i=1}^{N} x_i$, be a sum of N *idd* variables of almost any distribution [1]. The Central Limit Theorem states that under mild conditions [2] as N increases, the distribution of S_N approaches [3]:

$$p(S_n = s) = \frac{1}{\sqrt{2\pi N\sigma^2}} e^{-\frac{(s-N\mu)^2}{2N\sigma^2}}, \tag{4.5}$$

where μ and σ^2 are the mean and variance of the distribution governing the random variables x_i. This implies that the quantity

$$Z_N \equiv \frac{S_N - N\mu}{\sigma\sqrt{N}} = \frac{\bar{X} - \mu}{\sigma/\sqrt{N}}$$

converges to a *standard normal* distribution, i.e. $\mathcal{N}(0,1)$.

Loosely speaking, if we sample sums of N random variables they follow a Gaussian distribution. The higher N, the more accurate the correspondence. Since averages are sums, they too follow a Gaussian.

In Activity *The Central Limit Theorem in Action* you will experiment with the application of this theorem to concrete datasets.

[1] The are a few exceptions, the Cauchy distribution, which has infinite variance is the best known

[2] C. Gardiner (2009). *Stochastic methods*, volume 4. Springer Berlin

[3] K. P. Murphy (2012). *Machine learning: a probabilistic perspective*. MIT press

4.1.2 *Z-score and Quantiles*

For large populations the mean and variance can often be estimated very accurately. A distribution of male heights in cm for example can be given as $\mathcal{N}(179, 10^2)$. If mean and variance are known we can estimate whether an individual is large compared to the rest of the population. The *cumulative distribution function* $F(x)$ can be given in terms of a probability density function by:

$$F(x) = \int_{-\infty}^{x} f(x)dx$$

The cumulative probability density (CPD) gives the probability $P(X \leq x)$. For the Gaussian, the cumulative probability is closely related to the error function, which is defined as:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt, \tag{4.6}$$

which can easily be calculated in numpy. It is a straightforward exercise to relate this function to the CPD of the Gaussian distribution:

$$\Phi(x; \mu, \sigma) = \frac{1}{2} \left(1 + \text{erf} \left(\frac{x - \mu}{\sigma \sqrt{2}} \right) \right) \tag{4.7}$$

Without reference to μ and σ, the function Φ refers to the CDF of $\mathcal{N}(0, 1)$.

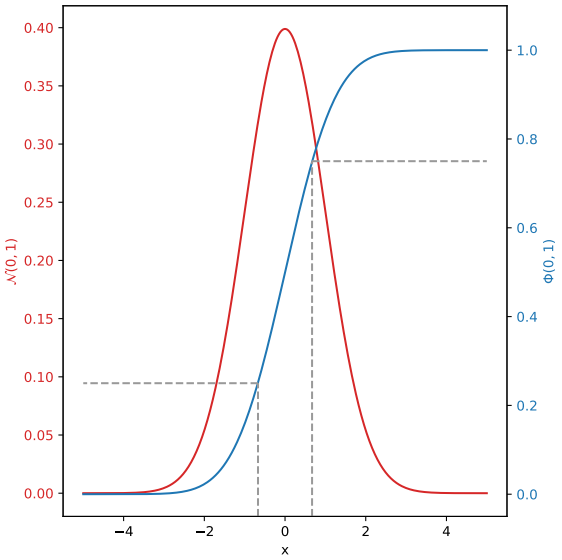


Figure 4.9: Standard Gaussian distribution (red). Its cumulative distribution function (blue). First and last quartile indicated by grey dashed lines.

For any CDF F that is monotonically increasing, there exists an inverse which is denoted by F^{-1}. $F^{-1}(\alpha)$ is called the α *quantile*. By definition, then, this is value x_α for which $P(X < x_\alpha) = \alpha$. The value of $F^{-1}(0.5)$ is called the *median* of the distribution. The values of $F^{-1}(0.25)$ and $F^{-1}(0.75)$ are called the lower and upper *quartiles*. They are shown in Fig 4.9.

The inverse CDF can be used to calculate *tail area probabilities*. Points to the left of $F^{-1}(\alpha/2)$ contain $\alpha/2$ of the probability mass. Points to the right of $F^{-1}(1 - \alpha/2)$ also contain $\alpha/2$ of the probability mass. The *central interval* $(F^{-1}(\alpha/2), F^{-1}(1 - \alpha/2)$ contains $1 - \alpha$ of the probability mass. For example, for $\alpha = 0.05$, the 95 % central interval is given by: $(\Phi^{-1}(0.025), \Phi^{-1}(0.975)) = (-1.96, 1.96)$.

When μ, σ are accurately known, the z-score for a data point x is given by $z = \frac{x-\mu}{\sigma}$. Using the z-score, a data point can be compared

to the $\mathcal{N}(0,1)$ interval. For example, if a data point falls outside the central interval as calculated above, it is atypically large or small. This is only possible when the parameters are accurately known rather than estimated. This is the case for very large populations, or comparing to ideal coins or dice and making predictions about the outcome using the central limit theorem.

As an example, consider the height of UK males, which is given as 174.5 cm for the 50-percentile, 186.0 cm for the 95-percentile and 164.1 cm for the 5-percentile [4] (cited from https://unece.org/fileadmin/DAM/trans/doc/2005/wp29grsp/HR-04-14e.pdf). Assuming that the height distribution is a Gaussian and that these numbers were measured on a large sample of the population we find that apparently $\mu = 174.5$ and since $\Phi^{-1}(0.95) = 1.65$, this gives a z-score which we can find σ by:

$$\sigma = \frac{x_{95} - \mu}{z_{95}} = \frac{186.9 - 174.5}{1.65} = 7.52\text{cm}$$

We now have estimates for μ and σ and should be able to make predictions about the 5-percentile, which we have not used in the calculation. $z_{05} = -1.65$ (as expected), giving $x_{05} = \mu + \sigma * z_{05} = 162.1$ cm which is in the right ball park, but shows a deviation from the reported value of 164.cm, suggesting heights are not distributed as a perfect Gaussian.

The z-score is useful when the the Gaussian parameters are known, or when a score can be compared to a hypothetically perfect distribution like the Bernoulli distribution of a perfect coin. The average of N observations will be Gaussian by the central limit theorem, so when we observe 100 coin tosses and calculate the z-score, we will be able to calculate the percentile function for this given outcome. We can then do a primitive form of hypothesis testing where the null hypothesis is that the coin is fair, and where we use an observation that has less than 5 % probability under the assumption that the coin is fair to reject this hypothesis.

When the parameters are not known, for example when the variance must be estimated from the data itself, the calculated $z-$score no longer follows a Gaussian distribution and other means like t-tests must be used. We will not discuss this further here. Standard statistic textbooks can be consulted, e.g. [5].

[4] D. of Trade and Industry (1998). *ADULTDATA - The handbook of adult anthropometric and strength measurements.*

[5] C. Gardiner (2009). *Stochastic methods*, volume 4. Springer Berlin

5 Multivariate Gaussian Distributions

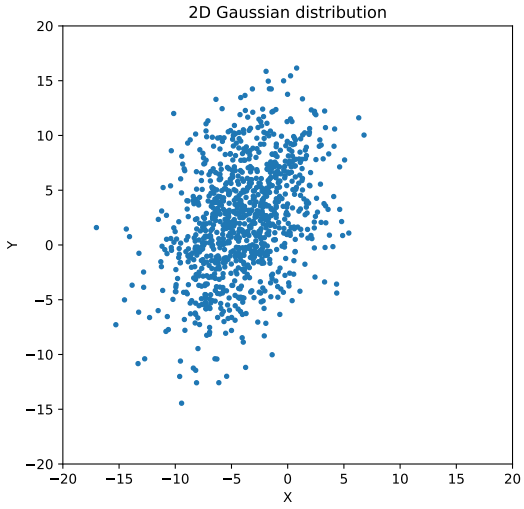


Figure 5.10: Sample of a two dimensional Gaussian distribution in two variable: X and Y. The variables are correlated and not centred at the origin.

5.1 Introduction

Multivariate means that it is a distribution in more than one variable. An example of samples drawn from a two-dimensional distribution are shown in Fig. 5.10. Before giving the formula for the multivariate Gaussian, a few remarks on notation.

A data point in multivariate distribution of dimension N is a tuple of N numbers, which is usually represented as a vector. In much of the machine learning literature, vectors are indicated by boldface lower case symbols. E.g. suppose we record IQ and height of individuals we have a data points where one dimension is used to record the height of the individual and a second one to record the IQ. We

denote the data point by:

$$\mathbf{x}_i = \begin{pmatrix} \text{iq}_i \\ \text{height}_i \end{pmatrix} \tag{5.1}$$

Some books use the vector notation $\vec{x}$, but the boldface notation is more prevalent.

Mathematical purists might object that these data points are tuples rather than vectors and they would have a point, but many of the operations in machine learning rely on so-called matrix-vector manipulations by numerical software and it is easier to stick to the conventions used in the machine learning literature.

The boldface notation indicates a column vector. Transposing it yields a row vector, so:

$$\mathbf{x}^T = (\text{iq}, \text{height}) \tag{5.2}$$

Matrix multiplication rules govern how expressions should be evaluated. If $\mathbf{v}$ and $\mathbf{w}$ are both N-dimensional vectors, then $\mathbf{v}^T\mathbf{w}$ represents the scalar product between these vectors, since it is the product of a row vector with a column vector, e.g.:

$$(v_1, v_2) \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = v_1 w_1 + v_2 w_2.$$

On the other hand, $\boldsymbol{v}\boldsymbol{w}^T$ represents a matrix whose components are $v^i w_j$ as can be seen from:

$$\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \begin{pmatrix} w_1 & w_2 \end{pmatrix} = \begin{pmatrix} v_1 w_1 & v_1 w_2 \\ v_2 w_1 & v_2 w_2 \end{pmatrix}$$

Matrices are indicated by boldface upper case symbols. Where matrix and vector dimensions match matrix vector multiplications are implied by the order in which symbols occur. So, $\mathbf{M}\mathbf{v}$, represents a column vector:

$$\sum_{j=1}^{N} M_j^i v^j,$$

whereas $\mathbf{v}^T M \mathbf{w}$ represents a number:

$$\sum_{i,j} v_i M_j^i w^j$$

The boldface notation without component indices is usually more efficient and often unambiguous. Sometimes it is necessary to resort to components, in particular in proofs.

The multivariate Gaussian distribution is given by:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{\mid \det(\boldsymbol{\Sigma}) \mid^{\frac{1}{2}}} \exp\left\{ -\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}) \right\} \tag{5.3}$$

The functional dependence of the Gaussian on $\mathbf{x}$ is given by the *Mahalanobis distance*:

$$\Delta^2 = (\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}) \tag{5.4}$$

Here, $\boldsymbol{\mu}$ is an M-dimensional vector, and $\boldsymbol{\Sigma}$ an $M \times M$ matrix. Without loss of generality, we can assume that $\boldsymbol{\Sigma}$ is a symmetric matrix.

Any matrix $\boldsymbol{M}$ can be written as the sum of a symmetric and an anti-symmetric matrix:

$$\boldsymbol{M}^i_j = \frac{1}{2}(\boldsymbol{M}^i_j + M^j_i) + \frac{1}{2}(\boldsymbol{M}^i_j - M^j_i)$$

You should verify that the anti-symmetric part does not contribution to the term under exponential, so we can always assume that $\boldsymbol{\Sigma}$ is symmetric.

5.1.1 *Spectral Decomposition of the Covariance Matrix*

The material in this section is relatively abstract. A Jupyter notebook titled: **Eigenvectors of the Covariance Matrix** *contains a worked example of the concepts discussed in this section.*

It is well known from linear algebra that a symmetric matrix can be diagonalised if a coordinate transformation is carried out from the original coordinate system wherein the data points are represented to an orthonormal basis comprised of the eignvectors of the matrix.

The eigenvector equation for the covariance matrix is:

$$\boldsymbol{\Sigma} \boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i,$$

where $i = 1, \cdots, D$.

In practice, in machine learning, we will find both eigenvectors and eigenvalues using numerical method. The notebook *Eigenvectors of the Covariance Matrix* gives examples and shows how the concepts discussed below apply to the Gaussian distribution. In general, any symmetric $D \times D$ matrix has D real eigenvalues. Moreover, the eigenvectors corresponding to these eigenvalues can be chosen to be orthonormal, i.e.

$$\boldsymbol{u}_i^T \boldsymbol{u}_j = \mathbb{1}, \tag{5.5}$$

where $\mathbb{1}$ is the D-dimensional identity matrix. The matrix $\boldsymbol{U}$ is a matrix where row i is given by $\boldsymbol{u}_i^T$

Now introduce:

$$\boldsymbol{y}_i = \boldsymbol{u}_i^T (\boldsymbol{x} - \boldsymbol{\mu})$$

The $\boldsymbol{y}_i$ are new coordinates which are translated and rotated with respect to the old ones. Forming the vector $\boldsymbol{y} = (\boldsymbol{y}_1, \cdots, \boldsymbol{y}_D$, one can write:

$$\boldsymbol{y} = \boldsymbol{U}(\boldsymbol{x} - \boldsymbol{\mu})$$

What this achieves is that, first the distribution is centred at the origin, and then a rotation is to the distribution so that the coordinate axes coincide with the eigenvectors (this is possible because the eigenvectors are orthonormal, so a rotation exists that aligns the eigenvectors with the coordinate axes; the rotation matrix is given by $\boldsymbol{U}$).

In the new coordinate system the distribution is the product of D independent Gaussian distributions:

$$p(\boldsymbol{y}) = \Pi_{j=1}^{D} \frac{1}{(2\pi\lambda_j)^{1/2}} \exp\left\{-\frac{y_j^2}{2\lambda_j}\right\} \tag{5.6}$$

If you are familiar with the process of diagonalising a matrix, this is what we have just done. We have diagonalised $\boldsymbol{\Sigma}$ to the diagonal matrix $\text{diag}(\lambda_1, \cdots, \lambda_D)$. For a diagonalised matrix the distribution factorises.

It is sometimes convenient to build a covariance matrix from a set of eigenvectors and eigenvalues. The spectral decomposition theorem states that covariance matrix can be expressed as an expansion in terms of its eigenvectors:

$$\boldsymbol{\Sigma} = \sum_{i=1}^{D} \lambda_i \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{5.7}$$

For the inverse:

$$\boldsymbol{\Sigma}^{-1} = \sum_{i=1}^{D} \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{5.8}$$

This has useful applications. In *Activity The Central Limit Theorem* you will learn how you can employ this equation to generate synthetic data sets.

5.1.2 *Maximum Likelihood Estimation*

We will derive the maximum likelihood estimators for $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$. It is more important that you are able to read the resulting formulae and can convert them into working code than that you can perform these derivations. We will not assess you on them. Having said that, if you want to be able to engage with the theoretical literature in the future you should be able to follow the derivations below and if not, you should acquire the matrix algebra underlying them. Appendix C of [1] provides a relatively comprehensive summary of the results. If you do not want to follow the details of the derivation, skip until Eqs. 5.21 and 5.26.

[1] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

Given a set of data points, that we suspect are Gaussian, how do we determine its parameters? Assume that we have a data set $\boldsymbol{X} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N)^T$. Note that this is a matrix with each individual

data point constituting one row of the matrix. For N data points, each of dimension D, the log likelihood function is given by:

$$\ln p(\boldsymbol{X} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{ND}{2}\ln(2\pi) - \frac{N}{2}\ln\det(\boldsymbol{\Sigma}) - \frac{1}{2}\sum_{n=1}^{N}(\boldsymbol{x}_n - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_n - \boldsymbol{\mu}) \tag{5.9}$$

In the following we will use the following results from matrix algebra: The *trace* of a matrix is the sum of its diagonal elements:

$$\mathrm{Tr}\boldsymbol{A} = \sum_{i=1}^{D} A_{ii} \tag{5.10}$$

For real symmetric matrices, remember that they can be diagonalised by means of a rotation:

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T, \tag{5.11}$$

where $\Lambda = \mathrm{diag}(\lambda_1, \cdots, \lambda_D)$, the diagonal matrix with the eigenvalues of $\boldsymbol{A}$ at the diagonal entries.

It can be shown that this has as consequence that

$$\det(\boldsymbol{A}) = \Pi_{i=1}^{D}\lambda_i, \tag{5.12}$$

and

$$\mathrm{Tr}(\boldsymbol{A}) = \sum_{i=1}^{D}\lambda_i \tag{5.13}$$

Both these results depend on Eq. 5.11 so hold for real symmetric matrices, but not in general.

The following results from matrix algebra can be proven simply by writing them out in components:

$$\frac{\partial}{\partial \boldsymbol{a}}\boldsymbol{b}^T\boldsymbol{a} = \boldsymbol{b} \tag{5.14}$$

$$\frac{\partial}{\partial \boldsymbol{a}}(\boldsymbol{a}^T\boldsymbol{A}\boldsymbol{a}) = (\boldsymbol{A} + \boldsymbol{A}^T)\boldsymbol{a} \tag{5.15}$$

$$\frac{\partial}{\partial \boldsymbol{A}}\mathrm{Tr}(\boldsymbol{B}\boldsymbol{A}) = \boldsymbol{B}^T \tag{5.16}$$

$$\tag{5.17}$$

also easy to verify is:

$$\mathrm{Tr}(\boldsymbol{A}\boldsymbol{B}\boldsymbol{C}) = \mathrm{Tr}(\boldsymbol{C}\boldsymbol{A}\boldsymbol{B}) = \mathrm{Tr}(\boldsymbol{B}\boldsymbol{C}\boldsymbol{A}) \tag{5.18}$$

With the last rule it is easy to justify the so-called *trace trick*:

$$\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = \mathrm{Tr}(\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}) = \mathrm{Tr}(\boldsymbol{x}\boldsymbol{x}^T\boldsymbol{A}) = \mathrm{Tr}(\boldsymbol{A}\boldsymbol{x}\boldsymbol{x}^T) \tag{5.19}$$

If the first equality baffles you, remember that the first term is a row vector, left multiplying a matrix, left multiplying a column

vector. But a column row can also be interpreted as an $1 \times N$ matrix, and a column vector as an $N \times 1$ vector, so this a product of three matrices.

Based on the spectral decomposition Eq. 5.7, 5.8, and Eq. 5.12 as well as $\boldsymbol{u}_i^T \boldsymbol{u}_j = I$ one can show:

$$\frac{\partial}{\partial x} \ln \det(\boldsymbol{A}) = \operatorname{Tr}\left(\boldsymbol{A}^{-1} \frac{\partial \boldsymbol{A}}{\partial x}\right),$$

which specialises to:

$$\frac{\partial}{\partial \boldsymbol{A}} \ln \det(\boldsymbol{A}) = (\boldsymbol{A}^{-1})^T = \boldsymbol{A}^{-T}, \tag{5.20}$$

where the last equality introduces a convenient shorthand for the transpose of the inverse of a matrix.

To estimate μ_{ML} we set:

$$\frac{\partial}{\partial \boldsymbol{\mu}} \ln p(\boldsymbol{X} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) \mid_{\mu=\mu_{ML}} = 0$$

Only terms that depend on $\boldsymbol{\mu}$ in Eq. 5.9 are relevant:

$$\frac{\partial}{\partial \boldsymbol{\mu}} \sum_{i=1}^{N} (\boldsymbol{x}_i - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}_i - \boldsymbol{\mu}) \mid_{\mu=\mu_{ML}} = 0$$

By virtue of identity 5.15 this is equivalent to:

$$-\sum_{i=1}^{N} (\boldsymbol{\Sigma}^{-1} + \boldsymbol{\Sigma}^{-T})(\boldsymbol{x}_i - \boldsymbol{\mu}) \mid_{\mu=\mu_{ML}} = 0,$$

From this it follows that

$$-2\boldsymbol{\Sigma}^{-1} (\sum_{i=1}^{N} x_i - N\boldsymbol{\mu}) \mid_{\mu=\mu_{ML}} = 0$$

since $\boldsymbol{\Sigma} = \boldsymbol{\Sigma}^T$, and therefore:

$$\boldsymbol{\mu}_{ML} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i \tag{5.21}$$

The MLE of μ is the *empirical mean*.

To produce an MLE of the covariance matrix, it is convenient to introduce the *precision matrix*, which is the inverse of the covariance matrix:

$$\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} \tag{5.22}$$

The only terms in the log likelihood that involve the covariance matrix are:

$$\mathcal{L} = -\frac{N}{2} \ln \det(\boldsymbol{\Sigma}) - \frac{1}{2} \sum_{i=1}^{N} (\boldsymbol{x}_i - \mu)^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}_i - \boldsymbol{\mu})$$

Expressed in terms of the precision matrix Λ this reads:

$$\mathcal{L}(\mathbf{\Lambda}) = \frac{N}{2}\ln\det(\mathbf{\Lambda}) - \frac{1}{2}\sum_{i=1}^{N}(\boldsymbol{x}_i - \mu)^T\mathbf{\Lambda}(\boldsymbol{x}_i - \boldsymbol{\mu}) \tag{5.23}$$

where we used $\det(\mathbf{\Lambda})\det(\mathbf{\Sigma}) = 1$, which follows from:

$$\det(\boldsymbol{AB}) = \det(\boldsymbol{A})\det(\boldsymbol{B})$$

Employing the trace trick on this expression gives:

$$\begin{aligned}\mathcal{L}(\mathbf{\Lambda}) &= \frac{N}{2}\ln\det(\mathbf{\Lambda}) - \frac{1}{2}\sum_{i=1}^{N}\mathrm{Tr}(\boldsymbol{x}_i - \boldsymbol{\mu})^T\mathbf{\Lambda}(\boldsymbol{x}_i - \boldsymbol{\mu}),\\ &= \frac{N}{2}\ln\det(\mathbf{\Lambda}) - \frac{1}{2}\mathrm{Tr}(\boldsymbol{S}_\mu\mathbf{\Lambda}),\end{aligned} \tag{5.24}$$

where the scatter matrix

$$\boldsymbol{S}_\mu \equiv \sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T \tag{5.25}$$

The MLE for the precision matrix is very easy to derive in this form:

$$\frac{\partial\mathcal{L}(\mathbf{\Lambda})}{\partial\mathbf{\Lambda}}\,|_{\Lambda=\Lambda_{ML}} = \frac{N}{2}\mathbf{\Lambda}^{-T} - \frac{1}{2}\boldsymbol{S}_\mu^T = 0.$$

Using $\mathbf{\Lambda}^{-T} = \mathbf{\Lambda}^{-1} = \mathbf{\Sigma}$, we find:

$$\mathbf{\Sigma}_{ML} = \frac{1}{N}\boldsymbol{S}_\mu = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T \tag{5.26}$$

5.2 *Completing the Square - Marginal and Conditional Probabilities of the Gaussian*

In general, when given a high dimensional distribution it is difficult to marginalise with respect to given variables because this entails integration in high dimensional spaces unless one is fortunate enough to be able to integrate analytically. Normalising a distribution can be difficult for the same reason.

Gaussian distributions allow analytic integration and thereby marginalisation and normalisation. The technique occurs often enough in the machine learning literature to warrant an activity centred around this topic. A simple but important observation is that the log likelihood of a Gaussian is a quadratic form. Usually, it is sufficient to manipulate this quadratic form and restore the original distribution after the fact.

To see this, observe that:

$$-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}) = -\frac{1}{2}\boldsymbol{x}^T\mathbf{\Sigma}^{-1} + \boldsymbol{x}^T\mathbf{\Sigma}^{-1}\boldsymbol{\mu} + \text{const} \tag{5.27}$$

When the log likelihood is brought into the form of the lefthand side, the full distribution is easily restored: it just requires exponentiating the quadratic form, and obtaining the overall normalisation factor which a function of the covariance matrix Σ only. When the log likelihood is in the form of the righthand side, it can be brought into the form of the lefthand side by completing the square.

In practice, this means that if the linear and quadratic terms of the log likelihood are known, the entire probability distribution can be reconstructed. Many mathematical operations involving Gaussians can be performed by manipulating quadratic forms. If this is not clear now, in Activity *Completing the Square: Manipulating the Gaussian log likelihood* , we will also show that if prior and likelihood are Gaussian, the posterior is Gaussian again, a fact that can be used immediately in Bayesian Linear Regression.

6 Bayesian Linear Regression

6.1 Introduction

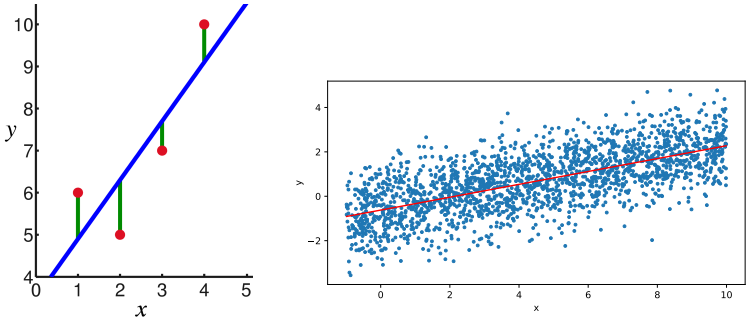


Figure 6.11: Left: observed points in red. A line fit to these points in blue. The residues in green (Wikimedia-ccby). Right: An example of linear regression.

The purpose of this section is to introduce the algebraic structure of linear regression, to contrast it in the next section with Bayesian linear regression. It is not necessary to be able to perform the derivations yourself, but you should be able to apply the resulting formulae. Examples of such applications are given in the notebook *Example 1.7: Linear Regression*.

The technique of linear regression is widely used in statistics and machine learning. You are expected to be familiar with it, because it was presented in the Data Science module. The essence of the method is shown in Fig. 6.11, where a linear functional relationship between two variables x and y is assumed, but where this relationship is imperfect, for example due to noise. It is clear that no linear relationship can fit the data perfectly, but on the other hand representation of the data by a linear function has a number of advantages. First, it requires much less information to store the parameters of the line than the entire data set. The linear function is called a *regression function* and the data is said to be regressed to a linear relationship. Second, the linear function may be used for *prediction*: when a new value of x is presented, the linear relation should present a reasonable prediction of y.

It is clear that some lines will be a good fit, and the line shown in

Fig. 6.11 is the best line in the following sense. It is clear that none of the data points are perfectly represented by the line. Consider a data point (x_i, y_i) and a model line $y = ax + b$ with a, b known parameters. For a given data point $r_i = y_i - (ax_i + b)$, the residue of point i represents a mismatch between the model prediction for value x_i, $ax_i + b$, and its actual value, y_i. Consider the sum of all residues squared: $R = \sum_i r_i^2$. For unknown a, b,R can be considered a function of a and b. OLS finds the values of a, b that minimise this function. Since R is essentially a quadratic function in a and b, we can be assured that a single global minimum exists. This should be expected, if we shuffle the line around in Fig. 6.11, it is clear that some sort of optimum line can be found.

Algebraically, conditions values for a and b can be found by minimising the function:

$$R = \sum_{i=1}^{N} (y_i - (ax_i + b))^2,$$

where N is the number of data points.

Differentiation with respect to a and b and setting the derivatives to o gives conditions for find the minimum:

$$\frac{\partial}{\partial a} R(a, b) = 0$$
$$\frac{\partial}{\partial b} R(a, b) = 0$$

This works out as a linear system of two equations with two unknowns:

$$\begin{aligned} \sum_i x_i y_i &= a \sum_i x_i^2 + b \sum_i x_i \\ \sum_i y_i &= a \sum_i x_i + bN \end{aligned} \tag{6.1}$$

In this particular case, formulae for a and b can easily be found by solving this system, which are the famous linear regression formulae.

A formal solution can be found by writing it in matrix-vector form:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_i x_i^2 & \sum_i x_i \\ \sum_i x_i & N \end{pmatrix}^{-1} \begin{pmatrix} \sum_i x_i y_i \\ \sum_i y_i \end{pmatrix} \tag{6.2}$$

It is perfectly possible to include higher order terms, and for example fit a second degree polynomial to the data. A second degree polynomial has three parameters and would lead three equations with three unknowns that can solved directly. We will formulate this approach in way that allows easy generalisation to arbitrary degree.

Introduce the vector $\phi(x_i)$, defined as:

$$\phi(x_i) = \begin{pmatrix} 1 \\ x_i \\ x_i^2 \end{pmatrix} \tag{6.3}$$

and a vector $\boldsymbol{w}$:

$$\boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}, \tag{6.4}$$

so that:

$$\boldsymbol{w}^T \boldsymbol{\phi}(x) = w_0 + w_1 x + w_2 x^2$$

indeed represents a general second degree polynomial. This idea easily generates to a polynomial of arbitrary degree. If we are fitting an $M-1$ degree polynomial, we need M parameters that we organise in an M-dimensional vector $\boldsymbol{w}$.

The idea of minimising the residual squared sum is completely equivalent to that when regressing to a linear function:

$$R = \sum_{i=1}^{N} (y_i - \boldsymbol{w}^T \boldsymbol{\phi}(x_i))^2 \tag{6.5}$$

Again, we will find the value for $\boldsymbol{w}_{OLS}$ by setting:

$$\frac{\partial}{\partial \boldsymbol{w}} R \mid_{\boldsymbol{w}=\boldsymbol{w}_{OLS}} = 0$$

Calculating the gradient leads to:

$$\sum_{i=1}^{N} (t_i - \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i)) \boldsymbol{\phi}(\boldsymbol{x}_i)^T = 0,$$

or

$$\sum_{i=1}^{N} y_i \boldsymbol{\phi}^T(x_i) = \boldsymbol{w}^T (\sum_{i=1}^{N} \boldsymbol{\phi}(x_i) \boldsymbol{\phi}^T(x_i)) \tag{6.6}$$

For each data point we have a column vector $\boldsymbol{\phi}(x_n)$. The *design matrix* is an $N \times M$, matrix whose elements are given by $\Phi_{nj} = \phi_j(x_n)$. Here $M-1$ is the degree of the polynomial that we intend to fit to the data and N is the number of data points. For the case of fitting a second degree polynomial, $M = 3$.

$$\Phi = \begin{pmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_{M-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_{M-1}(x_2) \\ \vdots & \vdots & & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_{M-1}(x_N) \end{pmatrix} \tag{6.7}$$

Using this matrix, we can write Eq. 6.6 as

$$\boldsymbol{\Phi}^T \boldsymbol{t} = \boldsymbol{\Phi}^T \boldsymbol{\Phi} \boldsymbol{w},$$

so that

$$w_{OLS} = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\boldsymbol{t} \tag{6.8}$$

Some care must be used in applying these equations as they may be numerically unstable. This may happen if two basis functions evaluate to nearly the same vector. In that case an SVD decomposition must be used [1].

[1] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

Although Eq. 6.8 is a very compact notation, the result is not fundamentally different from the linear regression example, nor is the reasoning leading to it. It is instructive to formulate the case of a straight line, which is a polynomial of degree 1, so $M = 2$.

The column vector for data point x_i in this case is:

$$\phi(x_i) = \begin{pmatrix} 1 \\ x_i \end{pmatrix},$$

so the design matrix Φ is given by:

$$\Phi = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{pmatrix}$$

so that its transpose Φ^T is:

$$\Phi^T = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \end{pmatrix},$$

The product of the two is:

$$\Phi\Phi^T = \begin{pmatrix} N & \sum_i x_i \\ \sum_i x_i & \sum_i x_i^2 \end{pmatrix}, \tag{6.9}$$

Since $\boldsymbol{t}$ is:

$$\boldsymbol{t} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix},$$

the formula for finding the appropriate coefficients (weights) for our polynomial is:

$$\begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} N & \sum_i x_i \\ \sum_i x_i & \sum_i x_i^2 \end{pmatrix}^{-1} \begin{pmatrix} \sum_i y_i \\ \sum_i x_i y_i \end{pmatrix} \tag{6.10}$$

Again, you will not be assessed on whether you can derive this result, but you are strongly encouraged to study notebook *Example 1.7: Linear Regression*, to see how the design matrix is built from the data and how Eq. 6.8 works out in practice.

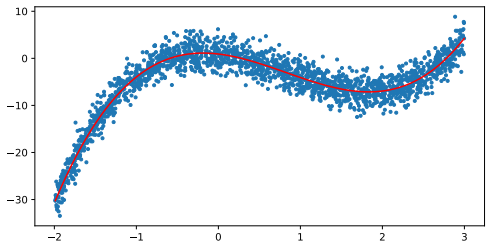


Figure 6.12: The result of linear regression on a cubic polynomial.

Figure 6.12 shows the fit of a cubic

It is important to realise that all examples of this section are linear regression, even if we fit a non linear function to the data. The distinction between linear and non linear regression is not whether or not to use of linear functions to model the data, but whether the fit function is a linear function its weights. In that case Eq. 6.8 applies. An example of non linear regression is *logistic regression*, to be discussed in Unit 2.

Also important is the realisation that polynomials are not the only function we can regress to. Any sufficiently rich set of *basis functions* can be used. In the case of fitting to a polynomial, the basis functions are the *monomials* $1, x, x^2, \cdots$. But Gaussians can also be used as basis functions. For example, if functions in a range $[a, b]$ need to be modeled, one can create a grid in μ, σ representing Gaussians of different means and variances. The notebook *Example 1.7: Linear Regression* gives a simple example of that.

So, the vector of functions $\boldsymbol{\phi}(x)$ can be chosen differently, but the process of constructing the design matrix and the regression procedure remain the same, given $\boldsymbol{\phi}(x)$, which are sometimes called *features*. For this reason linear regression is a powerful framework that is applicable to data with very non linear features.

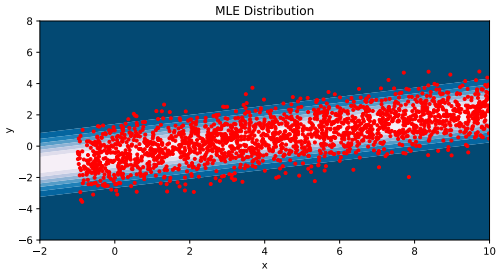


Figure 6.13: The MLE, a single point in (μ, σ) space, determines a Gaussian distribution centred around the line represented by the MLE.

6.2 *Maximum Likelihood and Least Squares*

In the ordinary least squares approach from the previous section no particular assumption was made about the origin of the residuals. Here we will see that it has a natural interpretation as the maximum likelihood estimation of a deterministic model with additive Gaussian noise.

Assume that we are observing data that has been generated by a process that can be described by

$$t = y(\boldsymbol{x}, \boldsymbol{w}) + \epsilon, \tag{6.11}$$

where ϵ is a Gaussian variable with zero mean and *precision* β (precision is the inverse of variance so $\beta = 1/\sigma$, its use is a matter of convenience, unburdening the notation of $^{-1}$s). In general, we will know the function y, but not the parameters $\boldsymbol{w}$, which we will have to infer from the observed data, just as in the case of linear regression.

Eq. 6.11 implies the existence of a probability distribution of target values, condition $\boldsymbol{x}$ and parameters $\boldsymbol{w}$ through the deterministic function y:

$$p(t \mid \boldsymbol{x}, \boldsymbol{w}, \beta) = \mathcal{N}(t \mid y(\boldsymbol{x}, \boldsymbol{w}), \beta^{-1}) \tag{6.12}$$

Consider the data set of inputs $\boldsymbol{X} = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N\}$ with corresponding target values $t_1, \cdots t_N$, we group the targets values $\{t_i\}$ into a column vector $\boldsymbol{t}$. Assuming data points are independently drawn from distribution 6.12, it is possible to define a likelihood function with adjustable parameters $\boldsymbol{w}$ and β

$$p(\boldsymbol{t} \mid \boldsymbol{X}, \boldsymbol{w}, \beta) = \Pi_{i=1}^{N} \mathcal{N}(t_i \mid \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i), \beta^{-1}), \tag{6.13}$$

where, as usual, $\mathcal{N}$ denotes the Gaussian distribution. The log likelihood then is:

$$\begin{aligned} \ln p(\boldsymbol{t} \mid \boldsymbol{w}, \beta) &= \sum_{i=1}^{N} \ln \mathcal{N}(t_i \mid \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i), \beta^{-1}) \\ &= \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi) - \beta E_D(\boldsymbol{w}) \end{aligned} \tag{6.14}$$

Here, the sum-of-squares error function is given by:

$$E_D(\boldsymbol{w}) = \frac{1}{2} \sum_{i=1}^{N} (t_i - \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i))^2 \tag{6.15}$$

But this is the same form as the sum of squared residues Eq. 6.5! So, the estimate for $\boldsymbol{w}$ that minimises the log likelihood and thereby maximises the likelihood, $\boldsymbol{w}_{MLE}$ is the same that minimises the sum of residues squared, and we find:

$$\boldsymbol{w}_{MLE} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{t} \tag{6.16}$$

It is important to note that no particular noise model underlies the OLS estimates, whereas the interpretation of $\boldsymbol{w}_{MLE}$ as maximum likelihood estimate explicitly determines on the Gaussian assumption made in Eq. 6.12. However, when the Gaussian model is appropriate it allows a more extensive analysis for example Bayesian linear regression.

6.3 *Bayesian Linear Regression*

Bayesian linear regression is a vast subject, and here we only give a bare bones introduction. The key idea is the same as in our example of the false coin: we assume that our parameters are uncertain and model this with a probability distribution. You are forced to assume a prior distribution, but as data becomes available, you use it to infer a posterior distribution *given* the data. This distribution will become more and more focused as more data becomes available. In the limit of infinite data, the distribution would become a sharp narrow peak that centres a value. In this limit the maximum likelihood would also converge on this value and both approaches would give the same result: a specific numerical prediction that contains no uncertainty. When not much data is available, the posterior distribution will be wider, representing the uncertainty in our fit parameters. MLE on the other hand will produce point estimate for them. Only with cross validation do we get a feeling for the uncertainty in this estimate.

The Bayesian linear regression case is an important illustration of the fact that: *Bayesian methods are less prone to overfitting than MLE estimates*. The concept of regularisation emerges as a natural consequence of the Bayesian approach, and Bayesian linear regression serves as a good demonstration.

As in the earlier example of the false coin where we had to assume a prior distribution for the parameter μ, we define a prior distribution for the weights $\boldsymbol{w}$:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{m}_0, \boldsymbol{S}_0) \tag{6.17}$$

The likelihood function is given by:

$$\mathcal{L} = \Pi_{i=1}^{N} \mathcal{N}(t_i \mid \boldsymbol{w}^T \boldsymbol{\phi}(x_i), \beta^{-1}), \tag{6.18}$$

which is the same we used in the case of the MLE.

Throughout this example, we will assume that β, the precision, is known. If it is not known, the approach can be extended to infer it as well, although there will be some technical difficulties that we do not want to address here. Here we will use the data to infer the set of weights $\boldsymbol{w}$ that will describe the data best.

In this particular case Bayes' rule can be written as:

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{t}, \alpha, \beta) \sim p(\boldsymbol{t} \mid \boldsymbol{X}\boldsymbol{w}\beta)p(\boldsymbol{w} \mid \alpha) \tag{6.19}$$

It is a relatively straightforward exercise, which you will do yourself in a worksheet, to show that for a Gaussian prior the posterior will also be a Gaussian. In this exercise you will find the mean and covariance of the posterior distribution in terms of the data and the mean and covariance of the prior distribution.

Here, we briefly state the formulae for Bayesian Linear regression. Again we want to describe our data with a deterministic polynomial, whose coefficients we want to determine from the data. In line with the Bayesian approach, we define a prior probability distribution over our weights. Based on the assumption of Gaussian distributed noise 6.13, we assume a Gaussian prior:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w} \mid 0, \alpha^{-1}\mathbb{1}),$$

that is: we centre our weights on zero and allow a single parameter to set the variance in each dimension. In the absence of prior knowledge about the data, this is a reasonable choice.

The posterior distribution is Gaussian again:

$$p(\boldsymbol{w} \mid \mathcal{D}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{m}_N, \boldsymbol{S}_N)$$

Here $\mathcal{D}$ is shorthand for the dataset $\boldsymbol{X}, \boldsymbol{t}$, where

$$\boldsymbol{X} = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N\}$$

and

$$\boldsymbol{t} = \{\boldsymbol{t}_1, \cdots, \boldsymbol{t}_N\},$$

where $\boldsymbol{t}_i$ is the observed value associated with $\boldsymbol{x}_i$, the value that our fit should reproduce as closely as possible.

We see that the posterior distribution is again a Gaussian, with a mean $\boldsymbol{m}_N, \boldsymbol{S}_N$ given by:

$$\boldsymbol{m}_N = \boldsymbol{S}_N(\boldsymbol{S}_0^{-1}\boldsymbol{m}_0 + \beta\boldsymbol{\Phi}^T\boldsymbol{t}) \tag{6.20}$$

$$\boldsymbol{S}_N = \boldsymbol{S}_0^{-1} + \beta\boldsymbol{\Phi}^T\boldsymbol{\Phi} \tag{6.21}$$

Here Φ is again the design matrix. $\boldsymbol{t}$ are the target values of the data set. α is a parameter which represents our prior. This value reflects the *subjective belief* about the weights prior to the data, so it must be chosen by us. If we chose it large, then this reflects a belief that the weights will be small. If we are not certain about this, we should pick a smaller value resulting in a broader prior.

We assume that β is known. If it is not, the likelihood function and prior become more complex than Gaussians, something we will for

now ignore, although estimating β from the data is certainly possible. We happen to know that $\beta = 1$, and will use that value here.

There are at least two possible ways of using these formulae.

1. *Batch learning*. Here we consider $\boldsymbol{m}_0, \boldsymbol{S}_0$ a prior, and the matrix Φ is constructed using the entire data set. A conventional choice is then:

$$p(\boldsymbol{w} \mid \alpha) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{0}, \alpha^{-1}\boldsymbol{I})$$

The formulae simplify somewhat in this case:

$$\boldsymbol{m}_N = \beta \boldsymbol{S}_N \boldsymbol{\Phi}^T \boldsymbol{t} \tag{6.22}$$

$$\boldsymbol{S}_N = \alpha \boldsymbol{I} + \beta \boldsymbol{\Phi}^T \boldsymbol{\Phi} \tag{6.23}$$

2. *Sequential learning*. We may again start with the following prior:

$$p(\boldsymbol{w} \mid \alpha) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{0}, \alpha^{-1}\boldsymbol{I}),$$

i.e. $\boldsymbol{m}_0 = \boldsymbol{0}$ and $\boldsymbol{S}_0 = \alpha^{-1}\boldsymbol{I}$.

and construct Φ from data that has just been acquired (which may be a single data point). We then calculate the N-th step according to:

$$\boldsymbol{m}_N = \boldsymbol{S}_N(\boldsymbol{S}_{N-1}^{-1}\boldsymbol{m}_{N-1} + \beta \boldsymbol{\Phi}^T \boldsymbol{t}) \tag{6.24}$$

$$\boldsymbol{S}_N = \boldsymbol{S}_{N-1}^{-1} + \beta \boldsymbol{\Phi}^T \boldsymbol{\Phi} \tag{6.25}$$

6.3.1 *Interpretation of Bayesian linear regression*

The MLE estimate itself is a point value. In weight space it corresponds to a single point. In data space, this is a distribution, since we model the data with a predictive distribution:

$$\mathcal{N}(t \mid \boldsymbol{w}^T \boldsymbol{\phi}(x), \beta^{-1}),$$

where we have assumed that the precision of the noise, $\beta = 1.0$. For each value of x, the MLE determines a distribution that peaks around the value $\boldsymbol{w}^T \boldsymbol{\phi}(x)$.

Let us recapitulate the MLE estimate for a linear relationship.

The values above give 'the best fitting line', both according to the criterion of a minimal sum of squared residues and the MLE of the likelihood. The predictive distribution is given by

$$\mathcal{N}(t \mid \ w_0 + w_1 x, 1.0),$$

We visualize this distribution as a heat plot, together with the original data.

6.3.2 *The Posterior Distribution*

The posterior distribution is not a single point like the MLE, but a distribution. We will show that the peak of this distribution usually is close to the MLE estimate. Some degree of numerical discrepancy is expected because the Bayesian maximum depends on our choice of prior. But the posterior distribution is not a single point in weight space: it is a distribution in weight space. As such it is a distribution of distributions. The differences between the MLE are more pronounced when not much data is available. We will first demonstrate the posterior obtained from a relatively small number of data points. As Fig. 6.14 shows, after 10 points, parameter w_0 has considerable uncertainty, whereas parameter w_1 has been much more constrained.

In order to visualise the influence of uncertainty in the posterior distribution, we can sample from it. The $\boldsymbol{w}$ values thus sampled each represent a linear relationship. Remember that each $\boldsymbol{w}$ represents an entire probability distribution. By plotting the linear relationships we represent each distribution by its peak value. We see that the gradients of the lines are better constrained than the intercepts, something that is also clear from the posterior distribution itself.

We have an example here of *model uncertainty*, an uncertainty in the model parameters. This comes on top of the noise that is intrinsic in the data, which is modelled by the covariance matrix, and which in the case of the MLE shown above was visible as a zone around the mean.

You are strongly encouraged to experiment with the notebook *Bayesian Linear Regression*.

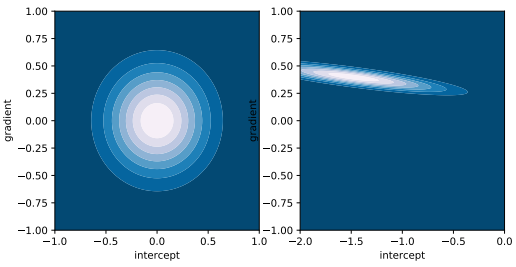


Figure 6.14: Prior and posterior weight distribution after applying Bayesian regression on ten points.

6.4 *Pros and cons of Bayesian Regression*

6.4.1 *Predictive Distribution and Maximum a Posteriori*

Using the MLE of the weights is simple. Once the optimal weights have been obtained, the are inserted in the polynomial. Using the

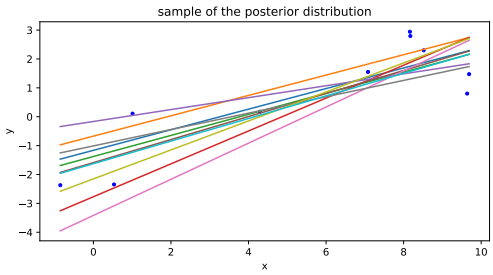


Figure 6.15: This plot shows 10 linear relationships that have been sampled from the posterior distribution, which was obtained by regressing on 10 data points.

regression results for prediction is as simple as inserting a new x value in the polynomial and obtain the corresponding y value, which is the prediction for this x.

To properly use the posterior distribution of weights, one would have to calculate the weighted expectation with respect to the posterior distribution values over all weights:

$$t = \int p(w \mid \boldsymbol{X}, \boldsymbol{t}) \boldsymbol{w}^T \boldsymbol{\phi}(x) dw$$

A simple estimate would probably be to generate a number of samples of the posterior distribution as we've done above, where we sampled 10 linear relationships. We can than average the predictions made by each of these lines as an estimated of the *predictive distribution*.

In many cases this is huge overkill. If the idea is to use regression to get an approximate prediction, it is not necessary to use the posterior distribution. Sometimes a good compromise can be to use its maximum. In this particular case this would be given by the value of $\boldsymbol{m}_N$. This is called the *maximum a posteriori* or *MAP*. Often a reasonable compromise, using the MAP is not entirely without risks, see for example section 5.2.1.2 of Murphy (2012).

In other cases, where little data is available and the penalty on a wrong decision based on MLE is severe it is better to accept the extra cost of the predictive distribution. In Unit 4 we will consider the case of determining whether a given point is an outlier. Such a decision may have financial or even legal consequences and here the importance of a principled estimate may outweigh the cost associated with evaluating the predictive distribution.

6.4.2 *Regularisation*

Bayesian models are far less prone to overfitting than MLE estimates, when applied consistently. The discussion for why this is the case becomes rather technical, but when two models explain the data adequately, the Bayesian approach favours the model with a smaller

number of parameters. This important aspect of Bayesian models is discussed in Section 3.4 and 3.5 of Bishop, but requires that you have digested most of the material earlier in Chapter 3 which goes into greater detail than we can do here.

We have seen that the Bayesian approach gives a posterior distribution of weights:

$$p(\boldsymbol{w} \mid \boldsymbol{t}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{m}_N, \boldsymbol{S}_N),$$

where you have derived the formulae for $\boldsymbol{m}_N, \boldsymbol{S}_N$. You should be able to verify that:

$$\ln p(\boldsymbol{w} \mid \boldsymbol{t}) = -\frac{\beta}{2} \sum_{i=1}^{N} \left\{ t_i - \boldsymbol{w}^T \boldsymbol{\phi}(x_i) \right\}^2 - \frac{\alpha}{2} \boldsymbol{w}^T \boldsymbol{w} + \text{const}$$

Maximising the likelihood is equivalent to maximising the log likelihood, which in turn is equivalent to minimising the quantity:

$$L = \sum_{i=1}^{N} \left\{ t_i - \boldsymbol{w}^T \boldsymbol{\phi}(x_i) \right\}^2 + \frac{\alpha}{2\beta} \boldsymbol{w}^T \boldsymbol{w}$$

The first term is a sum of squares, which also emerges in least squares. The second term effectively is a penalty term for large weights. This is called a *regularisation* term, as it puts constraints on the magnitude of the weights. A perfect fit which reduces the first term can still be spoilt if the weights to achieve it attain large values and in the earlier examples we saw this exactly what happens when we overfit a simple dataset.

L is sometimes called a *loss function* (for training purposes - not to be confused with a prediction loss, see the discussion in Section 1.5.5. of Bishop (2006)). Minimising it can be seen as an optimisation problem. In this unit we have achieved minimisation by analytic means, but in the following units we will often see loss functions that we have to minimise using numerical methods.

Modern neural network frameworks often allow the definition of models and an independent specification of loss functions. The *mean squared error* is a choice that is often made:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - t_i)^2,$$

and given the discussion so far it is not hard to see why it is an obvious choice, although it is by no means the only and often not the best one. Statisticians use a slightly different definition that incorporates the number of parameters.

Neural network frameworks also often the possibility for different forms of regularisation. The research into Bayesian neural networks is in its infancy (see [2] for a recent review), but regularisation offers some protection to overfitting.

[2] L. V. Jospin, W. Buntine, F. Boussaid, H. Laga, and M. Bennamoun (2020). Hands-on bayesian neural networks–a tutorial for deep learning users. *arXiv preprint arXiv:2007.06823*

7 Information Theory and Probability

7.1 Introduction

In machine learning it is important to establish whether or not events are distributed as expected. If not, we cannot make accurate predictions about new events. We may have to adapt the distributions that we are comparing our events against, often by tweaking parameters, a process we call *learning*. The requires that we develop a quantitative measure to what extent past events conform to a given distribution. An interesting approach to this problem comes from information theory.

The key ideas are simple. Imagine that we have events that we measure at some remote station, which may fall in four categories: 'a', 'b', 'c' and 'd'. To transmit the occurances of each event we may use four bits: we can transmit the combinations '00', '01', '10', '11' and use a code book linking these messages to the four categories. Assuming that all occurrences are equally likely, this is the best we can do. If there were not four but sixteen categories, we would need 4 bits: 'oooo', etc. Binary words can use 2^N combinations using N bits. If all combinations are equally likely, it therefore makes sense to measure the information in terms of the number of bits required to transmit an event x:

$$h(x) = -^2 \log p(x).$$

Here $p(x)$ is the probability of the event which is the inverse of the number of possibilities, and therefore a minus sign is necessary to get a positive number of bits. When the probability for different categories is not uniform, it makes sense to adapt the coding scheme. In the example of 8 events $\{a, b, c, d, e, f, g, h\}$, with probabilities $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64})$, it make sense to use a more efficient code [1] (as presented in [2]). If we were to ignore the difference in probabilities, we would have to assign 3 bits to each category to transmit an event. But if we use the following code strings 0, 10, 110, 1110, 111100, 111101, 111110, 111111, then on average we will transmit less bits.

This difference can be quantified using *entropy*: given a probability

[1] T. M. Cover (1999). *Elements of information theory*. John Wiley & Sons

[2] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

distribution $p(x)$ it is given by:

$$\mathbb{H}[x] = -\sum p(x)^2 \log p(x) \tag{7.1}$$

Note that this can be interpreted as the information averaged over all probabilities in the two examples we have given here. If we assign uniform probability to each of the 8 outcomes, we find:

$$\mathbb{H}[x] = -8 \times \frac{1}{8}^2 \log \frac{1}{8} = 3,$$

i.e. 3 bits.

If we calculate the average coding length under the probabilities listed above, using the coding scheme given above we find that it is:

$$\frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \frac{1}{16} \times 4 + 4 \times \frac{1}{64} \times 6 = 2,$$

so 2 bits. If we work out $\mathbb{H}[x]$ for that probability distribution, we also find 2 *by the same calculation*. This suggests an interpretation of $\mathbb{H}$ as average code length. The unequal distribution of probabilities allows a slightly more efficient transmission than the three bits per event that we found for equal probabilities. This idea is reminiscent of Morse code, which uses a short set of dashes and dots for letters that occur often (the 'e' is a '.'). You may wonder about why the particular code was chosen. This code allows the concatenation of events in longer strings, but allows an unambiguous resolution into individual event strings. You should check this.

The use of $^2\log$, logarithms with base 2 is relatively uncommon, and the natural logarithm is often preferred. If we write:

$$\mathbb{H}[x] = -\sum p(x) \ln p(x),$$

this amount to a change in units as changing the base of a logarithm multiplies the previous outcome by a constant factor. When we apply the natural logarithm, we no longer measure entropy in bits, but in nats.

Further credence to the interpretation of entropy as a measure of the average content carried by an event can be lent by the following observations:

- If only a single outcome has probably one, and all other potential possible outcomes have probability 0 then $\mathbb{H} = 0$. This requires an interpretation of $0 \ln 0$ as 0, which is justified since $\lim_{\epsilon \to 0} \epsilon \log \epsilon =$.
- The entropy for a given set of outcomes is maximal if all outcomes are equiprobable.

The analogy between entropy and the amount of information that can be transmitted holds only strictly for probability distributions which can be expressed as powers of $\frac{1}{2}$. For arbitrary distributions the entropy is a lower bound of the the information that can be transmitted using the best possible code. This is the essence of the *noiseless coding theorem* which is due to Shannon [3]. We will nonetheless stick to the code book metaphor as it provides good intuition for some of the measures that we will introduce below.

[3] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

Note that it is possible to express the amount of information associated with a certain event in terms of the probability of its outcome, again this $p_i \ln p_i$, where p_i is the probability of outcome i. This can be interpreted as the amount of nats required to store or transmit the event if an optimal code were to be used. The central role played by the logarithm in information theory is explained by the property of logarithms in any base:

$$\log(ab) = \log(a) + \log(b)$$

It is the only function with this property. It states that when possibilities multiply, information content adds. This should be clear from the examples above and you should now have an intuition for why logarithms play a central role in quantifying amounts of information.

The notion of entropy can be extended to continuous distributions but a subtlety occurs. Assume that the interval on which a continuous distribution is defined is split into a finite number of bins, each of width Δ. Assuming $p(x)$ is continuous, there exists an x_i such that

$$\int_{i\Delta}^{(i+1)\Delta} p(x)dx = p(x_i)dx$$

So, for a given discretisation the entropy can be written as:

$$\mathbb{H}_\Delta = -\sum_i p(x_i)\Delta \ln(p(x_i)\Delta) = \sum_i \Delta \ln p(x_i) - \ln \Delta$$

We omit the second term and then consider the limit $\Delta \to 0$. The first term on the righthand side then converges to

$$\lim_{\Delta \to 0} \sum_i p(x_i)\Delta \ln p(x_i) = -\int p(x) \ln p(x) dx$$

The integral is called *differential entropy*. It differs from the entropy by a term $\ln \Delta$ which diverges when Δ converges to 0. This reflects the fact that in a continuous distribution an infinite number of bits is required to register an event with total precision. In practice we are typically limited by machine precision and $\ln \Delta$ would be finite, but since it would be a constant contribution to every entropy formulation we are not particularly interested and simply omit in from our considerations.

For a multivariate density $p(\boldsymbol{x})$ the differential entropy is given by:

$$\mathbb{H}[\boldsymbol{x}] = \int p(\boldsymbol{x}) \ln p(\boldsymbol{x}) d\boldsymbol{x}$$

The following relationship sometimes appears in the literature, so we give it for completeness: Given a joint distribution $p(\boldsymbol{x}, \boldsymbol{y})$, assume that pairs of $\boldsymbol{x}, \boldsymbol{y}$ are drawn. Assume that the values of $\boldsymbol{x}$ is already known. The additional information needed to specify the corresponding value $\boldsymbol{y}$ is then $-\ln p(\boldsymbol{y} \mid \boldsymbol{x})$. The average information needed to specify $\boldsymbol{y}$ can be written as:

$$\mathbb{H}[\boldsymbol{y} \mid \boldsymbol{x}] = -\int \int p(\boldsymbol{y}, \boldsymbol{x}) \ln p(\boldsymbol{y} \mid \boldsymbol{x}) d\boldsymbol{y} d\boldsymbol{x}$$

using the product rule, this reads as:

$$\mathbb{H}[\boldsymbol{x}, \boldsymbol{y}] = \mathbb{H}[\boldsymbol{y} \mid \boldsymbol{x}] + \mathbb{H}[\boldsymbol{x}]$$

Here $\mathbb{H}[\boldsymbol{x}, \boldsymbol{y}]$ is the differential entropy of $p(\boldsymbol{x}, \boldsymbol{y})$ and $\mathbb{H}[\boldsymbol{x}]$ is the entropy of the marginal distribution $\mathbb{H}[\boldsymbol{x}]$. So the information required to specify $\boldsymbol{y}$ is the information required to specify $\boldsymbol{x}$ together with the extra information to specify $\boldsymbol{y}$ given $\boldsymbol{x}$.

7.2 *Relative Entropy*

Above, we introduced the notion of a code where the number of bits used to represent an event is coded is inversely proportional to the logarithm of the probability of the outcome of that event. That presupposes that we actually know this probability distribution. What if we have designed a code based on a probability distribution $p(x)$, when the actual distribution is a different one $q(x)$. We would be able to tell, because as we would collect events and store them on disk, we would need more information to record them than we would have expected.

Assume we want to transmit events based on a probability distribution $p(x)$. If we were to use a code book based on probability distribution $q(x)$, we will find that in general we will need to store a larger amount of information than if we had used a code book based on the true distribution $p(x)$. The amount of extra information is:

$$\begin{aligned} \mathrm{KL}(p \,||\, q) &= -\int p(x) \ln q(x) df - (-\int p(x) \ln p(x) dx) \\ &= -\int p(x) \ln \{\frac{q(x)}{p(x)} dx \end{aligned} \tag{7.2}$$

This is the *Kullback-Leibler* or *KL-divergence*, sometimes also called *relative entropy*. We will show that

$$\mathrm{KL}(p \,||\, q) \geq 0$$

and that only when $p(x) = q(x)$ for all x , $\mathrm{KL}(p \mid\mid q) = 0$. This underpins the statements made above that when using the 'wrong' code book, messages will be longer than they need to be.

All this implies that the KL-divergence can be used as a measure for how far two distributions diverge. It is not called a metric because it is asymmetric in its arguments, i.e. in general

$$\mathrm{KL}(p \mid\mid q) \neq \mathrm{KL}(q \mid\mid p).$$

As we will see, it derives its usefulness in part from this property.

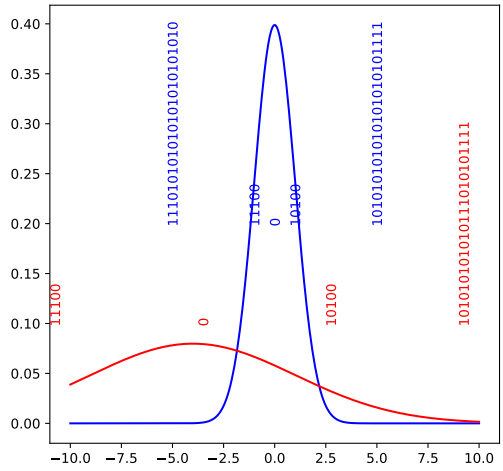


Figure 7.16: Events can be coded by a code book tailored to the red distribution or the blue distribution. When events are distributed according to the red distribution, they are more efficiently coded by the 'red' code book. Most events of the red distribution would fall well outside regular events according to the blue distribution, so using the 'blue' code book would lead to a huge increase in message size. The opposite is also true: events generated by the blue distribution are more efficiently code by the 'blue' code book, but events generated by the blue distribution could have plausibly come from the red distribution. This would lead to a longer message size if the 'red' code book were used although not nearly as much as in the opposite case. The KL-divergence is asymmetric in its arguments.

Consider Fig. 7.16. There is a blue and a red distribution. If events are distributed according to the blue distribution, their values are likely to be close to the peak of the blue distribution. A 'blue' code book would assign shorter bit sequences to those events, and longer ones to events well away from the peak because they are rare. Similarly for the red distribution. If we were to use the 'red' code book for events distributed according to the blue distribution, we would incur a penalty in terms of message length, because these events are somewhat away from the red centre and therefore would be longer sequences. Using the 'blue' code book for red events would be horrendous: most events of the red distribution cannot be plausibly generated by the red blue distribution. Only when the right code book is used for the right distribution of events is the message length minimal.

A real example in statistics would be to assume a Gaussian for events that are distributed according to a student distribution. The latter would be likely to generate outliers in a relatively small sample

that could not have been plausibly generated by a Gaussian. It is this property that makes the KL-divergence an efficient measure for divergence between two probability distributions.

Now suppose we sample data from an unknown distribution $p(x)$, which we may want to represent by a known distribution $q(|\ \theta)$, where θ are adjustable parameters. It would be nice to estimate the KL-divergence but the true distribution $p(x)$ is not available. If we have a finite set of data x_i, $i = 1 \cdots M$ drawn from distribution $p(x)$ we can approximate the KL-divergence by

$$\mathrm{KL}(p \,||\, q) \approx \sum_{i=1}^{N} \{-\ln q(x_n \mid \theta) + \ln p(x_n)\}\,.$$

The second term is independent of θ so minimising the KL-divergence with respect to θ is equivalent to maximising the likelihood function. In Unit 2, we will see how loss functions can be motivated based on the KL-divergence because of this.

7.3 *Jensen's Inequality*

The use of the KL-divergence as a measure that quantifies the inequality of two probability functions as a positive number rests on a simple mathematical theorem called *Jensen's inequality*. You will not be assessed on deriving it, but it occurs in many places in the machine learning literature. Alternatives exist to using the KL-divergence, e.g. *free energy*, but this too relies on Jensen's inequality. It is worth knowing about and being able to look up its proof when you encounter it in the literature.

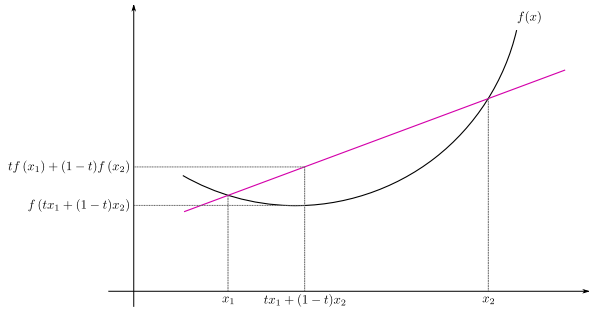


Figure 7.17: Jensen's inequality as a cord above a convex function. Source: Wikimedia.

A convex function is a function whose second derivative is positive everywhere. Examples are $f(x) = x^2$ and $g(x) = x \ln x$. An example is shown in Fig. 7.17. Consider a point of the purple line between x_1 and x_2. It is intuitively obvious that the entire purple line lies above $f(x)$ when $x_1 < x < x_2$. More precisely,

$$tf(x_1) + (1-t)f(x_2) \geq f(tx_1 + (1-t)x_2),$$

with strict inequality for $x_1 < t < x_2$ is $f(x)$ is strictly convex in that interval.

This is Jensen's inequality.

Using induction, this inequality can be proven for higher dimensions as well [4] (exercise 1.38) and then reads:

$$f(\sum_{i=1}^{M} \lambda_i x_i) \leq \sum_{i=1}^{M} \lambda_i f(x_i), \tag{7.3}$$

where $\lambda_i \geq 0$ and $\sum_i \lambda_i = 1$. By taking the values x_i as a discrete set, we can interpret the λ_i as a probability distribution over the discrete variables $\{x_i\}$. In that case Jensen's inequality can be written as:

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

There is a nice graphical interpretation of this inequality that can be found on Wikipedia: Consider Fig. 7.18. An arbitrary probability

[4] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

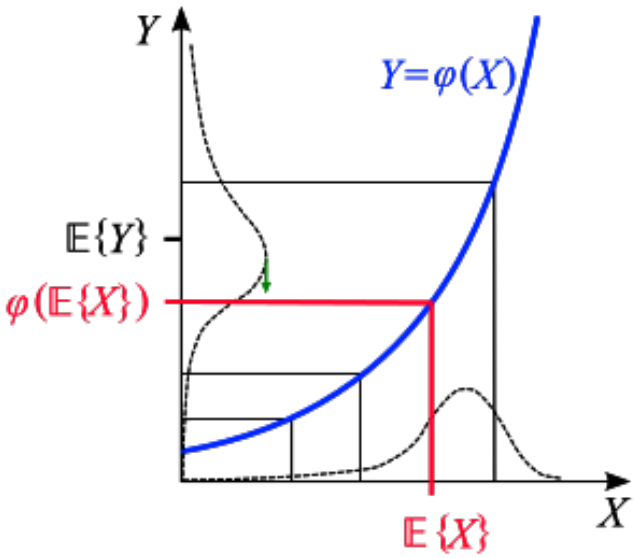


Figure 7.18: A graphical interpretation for $f(\mathbb{E}[x]) \leq \mathbb{E}[f(x)]$. Source: Wikipedia.

distribution can be found in the bottom of the plot with the density displayed as a function of X. The expectation value $\mathbb{E}[X]$ is shown in red and is slightly to the left of the bulge, due to the heavy tail extending to the left. A convex function $\phi(x)$ is the applied to all x values. The variables y are shown on the vertical axis. An interval in x, will be transformed into an interval in y and this interval in y must contain the same fraction of events as the interval in x. This leads to a transformation of the probability density over X into one over Y. It is clearly visible that the distribution is more stretched towards the higher the values in y. So you can look at the transformed value

of the expectation value, $\phi(\mathbb{E}[X])$ and compare it to the expectation value $\mathbb{E}[y]$, the expectation value of the y-distribution. The latter is always higher because the convex function $\phi(x)$ stretches larger values of x more than smaller values. This is the geometrical intuition underlying Jensen's inequality.

A final point that is both trivial and subtle. We can rewrite Jensen's inequality because the actual values of the x_i are immaterial. Suppose there is a function $y = g(x)$. Jensen's inequality only cares about the convexity of the function $f(x)$, not the value of the x_i. So written in terms of y_i, it is still true that:

$$f(\sum_{i=1}^{M} \lambda_i y_i) \leq \sum_{i=1}^{M} \lambda_i f(y_i),$$

In other words, Jensen's inequality can also be written as:

$$f(\sum_{i=1}^{M} \lambda_i g(x_i)) \leq \sum_{i=1}^{M} \lambda_i f(g(x_i)).$$

For continuous variables, this becomes:

$$f(\int \boldsymbol{x} p(g(\boldsymbol{x}))dx) \leq \int f(\boldsymbol{x}) p(g(\boldsymbol{x}))d\boldsymbol{x} \tag{7.4}$$

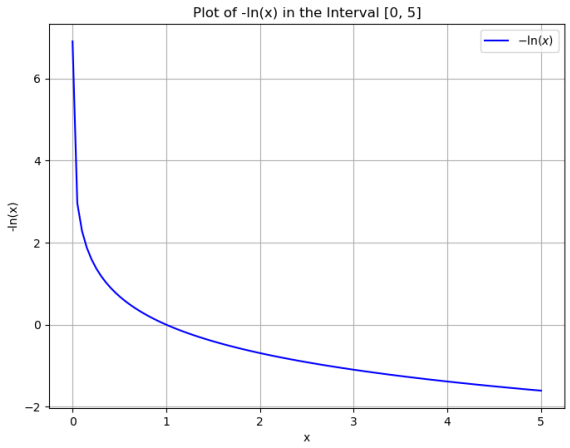


Figure 7.19: $-\ln x$ is a convex function.

Now we can apply this inequality to the KL-divergence. So far, we have discussed Jensen's inequality in terms of an abstract convex function. One such function is:

$$f(x) = -\ln x \tag{7.5}$$

Remember that this function plays a central role in this story due to its appearance in the entropy definitions. For completeness, we show

a plot of $f(x) = -\ln x$ in Fig 7.19. We can now use the definitions:

$$f(x) = -\ln x$$

and

$$g(x) = \frac{q(x)}{p(x)}$$

.

$$\mathrm{KL}(p \,||\, q) = -\int p(\boldsymbol{x}) \ln \left\{ \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} \right\} d\boldsymbol{x} \geq -\ln \int q(\boldsymbol{x}) d\boldsymbol{x} \tag{7.6}$$

Here we used that $\int q(\boldsymbol{x}) d\boldsymbol{x} = 1$, as $q(\boldsymbol{x})$ is a probability distribution and the fact that $-\ln x$ is a strictly convex function, so that equality will only hold when $p(\boldsymbol{x}) = q(\boldsymbol{x})$.

This proof is important as it confirms what we so far based on intuition: using the 'wrong' probability distribution to define a code book *always* leads to longer messages. Moreover, the properties of the KL-divergence are not really dependent on our intuition any more. Although it is useful to think of entropy and KL-divergence in terms of information theory, Eq. 7.6 shows that the properties of the KL-divergence are based on Jensen's inequality and as such can be rigorously defined, independent of a coding length metaphor.

8 *Some Cautions*

The material presented here is intended to provide the theoretical framework for the later material. It is not suitable for the analysis of real world data without further reading. We have not discussed the need for more robust inference when data contains *outliers*. At the very least, you should consult section 7.4 of [1]. In Unit 5 you may find useful techniques to model outliers as a mixture of different processes.

From this unit you may have received the impression that Bayesian analysis is the norm. It is not. Its use, in particular with complex models such neural networks is in its infancy. The main problem is that a neat interpretation of the posterior in terms of the parameters of a distribution is possible in only the simplest of cases. Even in logistic regression (Unit 2), only a numerical estimate of the posterior is possible and this is a huge complication in its own right. Unit 4 and 5 deal with methods to alleviate this problem.

[1] K. P. Murphy (2012). *Machine learning: a probabilistic perspective*. MIT press

9 Introduction

9.1 Connection to Unit 1

In Unit 1 we reviewed basic concepts of statistics and we introduced Bayes' Theorem. We encountered several parameterised probability distributions and we have seen that if we want to use them to model data processes we need to be able to estimate parameters from real data. We have seen that the most widely used method for this is maximum likelihood estimation MLE. MLE gives a point estimate for the model parameters, which is appropriate when sufficient data is available. We have also seen that we can adopt a prior probability distribution over our model parameters, by which we can reflect prior beliefs but also uncertainty in the model parameters. Bayes' Theorem gives us a way to adapt the prior distribution in the face of data, and to arrive at a posterior distribution of model parameters, which in general will be more accurate, but still reflects uncertainty about the model. This uncertainty should be taken into account when making predictions.

In the notebook examples, we have seen that MLE is prone to overfitting. Bayesian models are far less subject to overfitting. We have seen that to some extent in the emergence of *regularisation* when we performed Bayesian linear regression as opposed to ordinary least squares, which is a MLE of model parameters in linear regression under the assumption of a Gaussian noise model. We have seen that OLE corresponds to the minimisation of a *loss function*. In Bayesian linear regression, we minimise the same loss function but a penalty term is introduced that discourages the model parameters from being large, which is called a *regularisation* term.

The emergence of regularisation is nice, but does not really explain why Bayesian models are principally protected from overfitting. Not only are Bayesian models less prone to overfitting, but the evidence for different models can be compared *a posteriori*. This means that for example a linear or a quadratic fit can be compared with each other in terms of how well they are supported by the data without the need for cross validation. This in principle should lead to a more

efficient use of data. Unfortunately, the discussion for the underlying reasons takes up more space than we have available in the module. A good discussion can be found in sections 3.4 and 3.5 of [1]. Moreover, a consequent application of Bayesian principles is often very hard for technical reasons. This sometimes creates a need use to short cuts which then undercut the advantages of the Bayesian approach.

[1] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

We will demonstrate this in *logistic regression*. In linear regression, we were able to write down formulae for the model parameters such that they minimised loss functions. Because we had analytic expressions for these minima, the loss functions were only an aid in obtaining the desired results and we had little direct use for them.

In logistic regression we no longer can rely on analytic results because it requires the solution of non linear equations. Instead we are forced to adopt numerical approximations and find the parameters that minimise the loss functions by an iterative approach.

Bayesian logistic regression is hard for similar reasons. There is no easy way to find a closed analytic expression for the posterior distribution. There are three approximation methods, two of which will be discussed in later units. We will explain the problems in using Bayesian logistic regression.

Logistic regression can also be seen as the simplest *neural network*. Neural network research did not emerge from statistical theory or machining at large, but was inspired by ideas about how the brain might work. For this reason there is still some disparity between statistical and neural network communities. They sometimes have discovered similar concepts and refer to them by a different name. We will sketch the development of *Connectionism*, as the discipline of studying neural networks was called in cognitive science.

9.1.1 *The Unreasonable Effectiveness of Deep Learning in Artificial Intelligence*

After Unit 1, you might reasonably expect that linear regression is the predominant technique in data science, since despite its name, it is able to deal with non linear phenomena by the adoption of non linear functions as basis functions. We have mainly demonstrated regression on univariate models, but in principle we could for example use two dimensional polynomials as basic functions for two-dimensional input data, indeed polynomials of any dimension that matches the dimensionality of the input data. However, modern neural networks deal with high dimensional data, such as pixels. Even a simple image, like a handwritten numeral has $28 \times 28 = 784$ dimensions. The possible number of polynomials of degree d and number of variables n is given by:

$$\begin{pmatrix} d+n \\ n \end{pmatrix}$$

To fit a dataset with polynomials of degree 3 in 784 variables, we need to determine the value of 80 million coefficients! No dataset is rich enough to allow this by linear regression in the way we performed it in Unit 1.

Neural networks in practice have been able to perform reliable classification on images of this size. The reasons for why this is the case are not fully understood. The title of this section is taken from a recent paper by one of the pioneers in the field [2], who freely admits that he nicked this title from Eugene Wigner's *The Unreasonable Effectiveness of Mathematics in the Natural Sciences*. It is eminently readable and you should read this on the side. In this paper some pointers can be found as to why neural networks are efficient. But it mainly emphasises the inspiration of the brain sciences on their development.

[2] T. J. Sejnowski (2020). The unreasonable effectiveness of deep learning in artificial intelligence. *Proceedings of the National Academy of Sciences*, 117(48):pp. 30033–30038

Bishop's view [3] is that neural networks can be considered as regressors that are able to learn their own basis functions from the available data and therefore adapt to the particular training set. They do so with a large number of parameters, but not the astronomical number that would be required for polynomials. Moreover, modern deep learning uses techniques to keep the number of parameters under control, such as weight sharing. These methods will be discussed in detail in the *Deep Learning* module.

[3] C. M. Bishop (2006). *Pattern recognition and machine learning*. springer

The large number of parameters involved in the use of multilayered networks also work against the development of Bayesian methods. Although Bayesian neural networks have been around for years [4], truly Bayesian neural networks are still in their infancy [5]. *Variational methods*, which we will discuss in Unit 5, however have made a considerable impact, in particular *variational autoencoders*.

[4] C. M. Bishop *et al.* (1995). *Neural networks for pattern recognition*. Oxford university press

[5] L. V. Jospin, W. Buntine, F. Boussaid, H. Laga, and M. Bennamoun (2020). Hands-on bayesian neural networks–a tutorial for deep learning users. *arXiv preprint arXiv:2007.06823*

One of the key results of Connectionism, the branch of cognitive science that uses neural networks, was the development of an algorithm called *backpropagation by error*. Although the simple neural networks studied in the eighties of last century have evolved into *deep learning*, it is still useful to study simple examples because the backpropagation algorithm is a mainstay of modern deep learning and retained its importance for the field. Understanding this algorithm and being able to apply it in one of the modern neural network frameworks - we have chosen PyTorch - is the most important learning outcome of this unit.

9.1.2 *Learning Outcomes*

10 A Brief History of Neural Networks

The seat of soul has been placed in various parts of the body throughout history. It is only at the end of the 19th century that a clear hypothesis about the brain as an information processing unit start to emerge. Important influences were von Helmholtz's measurements of nerve speed, which helps to establish nerve function as an electrical phenomenon and the emergence of Golgi staining, which made it possible to study neurons for the first time. Without staining brain tissue is transparent under the microscope leading to a wide variety of theories about its possible constituents. A Spanish neuroscientist, Ramon-y-Cajal used this staining method to great effect to make beautiful and intricate pictures of groups nerve cells (Fig. 10.20).

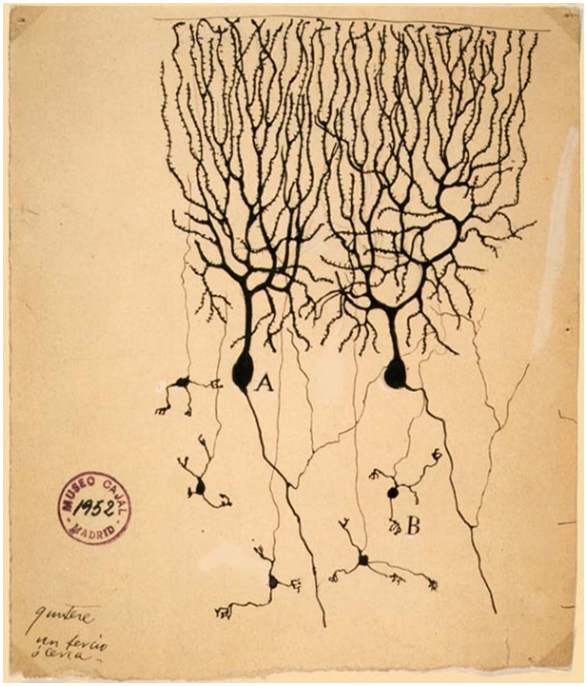


Figure 10.20: Drawing of a Purkinje cell (a type of neuron by Ramon-y-Cajal.

His and others' observations led to the *neural doctrine*, which among others states that https://en.wikipedia.org/wiki/Neuron_

`doctrine`:

- The brain is made up of individual units that contain specialised features such as dendrites, a cell body, and an axon.
- These individual units are cells as understood from other tissues in the body.
- Although the axon can conduct in both directions, in tissue there is a preferred direction for transmission from cell to cell.

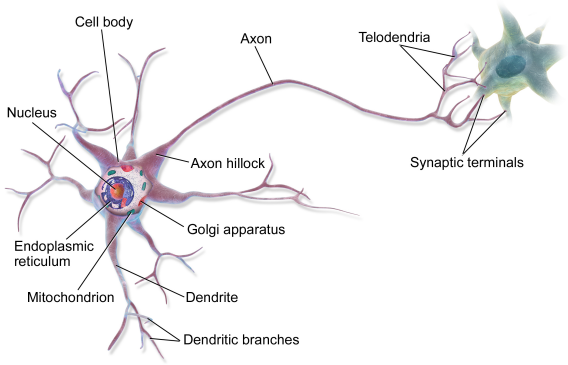


Figure 10.21: A schematic representation of a biological neuron.

The neuron doctrine is an important step forwards because it recognises neurons as the most important functional part of the brain, that neurons are cells, and that there is a clearly defined flow of information. Further experimental progress was made by many experimentalists, notably by Hodgkin and Huxley, who deducted the ion pump mechanisms that allow the neuron to be an electrochemical cell from electrophysiological measurements, later confirmed experimentally. By that time, the idea of the neuron as an information processing unit had taken hold. McCullogh and Pitts (1943) demonstrated that complex Boolean functions can be implemented by networks of artificial neurons, which are an important precursor to the artificial neurons that are used today.

There are many mechanisms by which neurons communicate with each other, but a predominant one is the following: neurons have a dendritic tree. Other neurons connect to the dendritic tree via *synapses*. They can release so-called *neurotransmitters*, which can alter the properties of ion channels of the receiving neurons. The effects can be *excitatory* or *inhibitory*. Ordinarily, a neuron maintains a negative potential difference between the inside and outside of the cell membrane. Upon the reception of an excitatory event neurons locally *depolarise*, i.e. become a little bit less negative. If nothing else happens, this *equilibrium potential* is restored within approximately 10

ms. If several such events heap up, the neuron depolarises so much that a spontaneous chain of depolarisation takes place, that will travel along the cell body and then further along the cable-like *axon*. This process resembles a fuse more than electric pulse conduction. This potential change is rapid and large, and for that reason is called a *spike*. It spreads spatially. This spike, when it has travelled down the axon causes the opening of so-called synaptic vesicles, which contain neurotransmitter. They can travel through the fluid surrounding the neurons and transverse the synaptic cleft to influence other neurons in turn.

If neurons receive only a small number of excitatory events, the depolarisation returns to equilibrium. Inhibitory events prevent depolarisation from building up. Only if a sufficient number of spikes arrive at the dendritic tree the local depolarisations can be so strong that a spike event will be caused. To some extent, one can think of this as a threshold. Unless depolarisations build up to a so-called threshold potential, neurons do not communicate with other neurons. Only spike events are communicated between neurons, not the subthreshold dynamics leading up to them. This is a highly simplified picture for sure: in the famous Hodgkin-Huxley [1] model, a system of differential equations that describes the ion movements across the neuron membrane and the resulting potential differences, you will not find a threshold, but the picture is useful and has stuck:

Key idea: A neuron integrates its input, and only responds if the net input crosses a certain threshold.

So, real neurons are a complex electrochemical system. In the Petri dish, we understand this system reasonably well. There is a whole area of science dedicated to predicting the behaviour of individual neurons in response to electrical and chemical manipulations: *computational neuroscience*. If the brain is a massive pile of neurons, some of which are stimulated by sensory input, driving other neurons, most only communicating with other neurons and some driving muscles, thereby implementing actuations, then where are our thoughts, feelings and memories? Somehow, they must be carried by the distributed activities of the neurons. And in this highly dynamic environment, how do we consolidate memories over our entire lifetime? The last question has an answer to some extent. It is agreed that an important component driving memory formation is formed by changes in the efficacy by which neurons influence each other. Such changes can be driven by the activation of the neurons themselves, so-called spike time dependent plasticity. An important example of this is the observation that neurons that are connected and fire within a given time window in the future are more likely to fire together if one of them receives stimulation, a principle that

[1] A. L. Hodgkin and A. F. Huxley (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):pp. 500–544

is called 'fire together - wire together'. It is the dominant view of the neurosciences as well as cognitive science that responses of the nervous system to outside stimuli are ultimately implemented in the form of synaptic efficacies. The learning of new behaviour and the formation of new memories entails changing these efficacies. There is substantial experimental evidence that behavioural changes indeed are associated changing synaptic efficacies, or the formation of new synapses, or the development of white matter tracts, which are an insulating sheet around the axons, and therefore indicate the formation of new connections. A recent paper [2] contains references to material supporting this view, but also points out that this general idea is very hard to proof conclusively, and that other mechanisms than synaptic plasticity may also underlie *learning*.

[2] W. C. Abraham, O. D. Jones, and D. L. Glanzman (2019). Is plasticity of synapses the mechanism of long-term memory storage? *NPJ science of learning*, 4(1):pp. 1–10

Nonetheless, the field of artificial neural networks has absorbed this idea:

Key idea: learning in an artificial neural networks is enacted by changing the efficacy by which neurons can influence each other.

In artificial neural networks synaptic efficacies are represented by real numbers and we will see examples of how artificial neural networks and their connections are represented in the sections below.

11 Perceptron and Linear Discriminants

11.0.1 The Perceptron

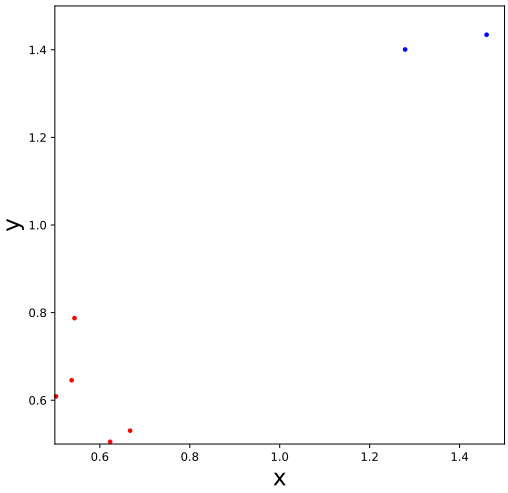


Figure 11.22: A dataset, consisting of measurement values which have been classified as belonging two one of two classes.

A key idea in the development of neural networks is Rosenblatt's perceptron. It is no longer used as a serous machine learning technique, but it is an excellent device to demonstrate some of the key principles that underpin neural networks. Consider Fig. 11.22. Here, two quantities x and y have been measured, and the measurement values have been *classified*. For example, x and y are two chemical concentrations, and the classification can be hazardous or non-hazardous. Based on the data sample that has been classified thus far, it seems that the two classes are well separated in terms of concentration space (x, y). If future measurements behave similar, it will not be difficult to design a *classifier*. One way of doing that is to draw a straight line between the two sets of points, and if a new data point arrives it can be classified according to which side of the line it falls. This is the idea behind a *linear classifier* (or discriminant).

Mathematically, we can implement this by representing the line between the two classes, the so-called *decision boundary* in terms of its parameters. The most general representation of a straight line in two dimensions is:

$$ax + by = c$$

and we can implement a decision by using a so-called *squashing* function, which here we take to be the Heaviside or step function:

$$o = \mathcal{H}(ax + by - c), \tag{11.1}$$

with

$$\mathcal{H}(x) = \begin{cases} 0 : x & < 0 \\ 1 : x & \geq 0 \end{cases}$$

Note that we need all three parameters: without c, we would only be able to represent lines through the origin, which would not do for the dataset of Fig. 11.22. The more commonly used representation of a straight line $y = ax + b$ is unsuitable because vertical lines can not be represented, and near vertical lines would require large parameters, something which is generally undesirable, consider for example, the discussion on regularisation in Unit 1.

Inspired by neuroscience, we can introduce an artificial neuron with two inputs, and a weight associated with each input. In general, we write:

$$o = f(w_1 x_1 + w_2 x_2 - \theta) \tag{11.2}$$

This is nothing but a rewrite of Eq. 11.1 of course. Our parameters w_1, w_2 are now called weights, and θ is called a *threshold* or *bias* (we will treat these words as synonyms). x_1 and x_2 are just our former variable x and y.

The device represented by Eq. 11.2 is called an *artificial neuron*. Calculating the *output state*, the value of variable o is called *updating* the neuron (we often drop the adjective artificial is this is clear from the context). Often, the output state is retained until a new update is made, for example in response to changed inputs.

Note that conceptually there are similarities to the real neuron: one could consider the quantity $w_1 x_1 + w_2 x_2$, the so-called *local field* as a representation of the fact that the weighted input contributions exceed the threshold θ, or not. If the threshold is not crossed, the response will be '0'. Like the real neuron, the artificial one is not affected by sub threshold activities, but if the threshold is exceeded, a non linear response follows: a sudden jump to '1'. Later, it will turn out that these non linearities are very important.

To illustrate both the process and also provide an illustration of McCullogh and Pitts point that networks of artificial neurons can

x_1	x_2	o
0	0	0
0	1	0
1	0	0
1	1	1

Table 11.4: The **AND** gate as a classification problem. It can be considered such because x_1, x_2 can be considered its inputs and the required logical value is the desired output classification.

implement complex Boolean functions, let us look at the logical **AND** gate as a classification problem.

The 4 input values can be represented as points in x_1, x_2 space and can be plotted with a marker to indicate whether the desired classification is '0' or '1'. The resulting figure, Fig. 11.23, is similar to Fig. 11.22 in that it is clear that a straight line can be found that separates the class '1' point from the class '0' points.

The equation of the line in the figure can easily be seen to be

$$y = -x + 1.5$$

We only have to do a simple rearrangement to bring this into perceptron form:

$$x + y - 1.5 = 0$$

We now have to decide which points should have outcome '0', and we chose:

$$o = \mathcal{H}(x_1 + x_2 - 1.5) \tag{11.3}$$

From this, we can read off weights and threshold: $w_1 = w_2 = 1$, $\theta = 1.5$. Note that the value of the threshold itself is a positive value, the minus sign in Eq. 11.2 ensures it works as a threshold.

So far, we have only determined the position of the decision line, but the decision could go the wrong way and we need to check that we have not accidentally chosen the inverse classification! We try the point $(0, 0)$. This gives:

$$1 \cdot 0 + 1 \cdot 0 = 0 < 1.5$$

The weighted sum is less than the threshold, so the output classification is '0', as it should be. The fact that the other points to be classified as '0' are on the same side the line and the one that is to be classified as '1' ensures that our classifier is correct. Repeating the calculation for the other values in the table we see that the only input able to overcome the threshold is: $x_1 = x_2 = 1$. This shows why the classifier works.

Now consider the following questions:

1. Does multiplying all weights and threshold by the same constant change anything in the classifiers output?

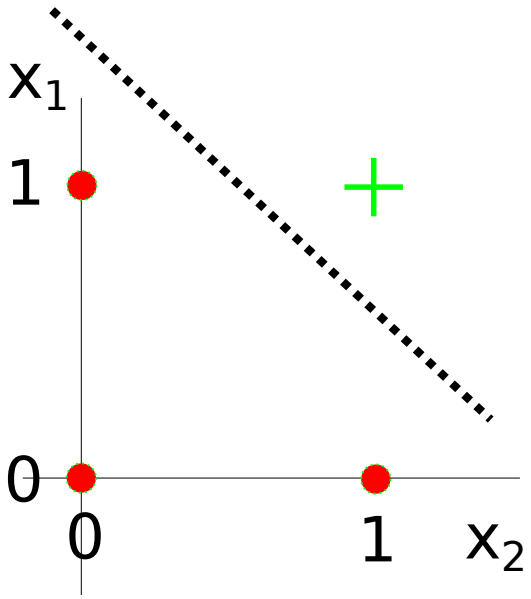


Figure 11.23: The AND gate as a classification problem is linearly separable.

2. Does it matter whether this constant is positive or negative?

From the figure it is clear that our perceptron has a nice property: it is fairly *robust*. If the input values are slightly distorted due to noise, the classification works correct, at least for this line.

It is also clear that the threshold plays an essential role: without it, we would only be able to form decision lines through the origin, and the **AND** problem can not be solved that way. Later, when we start to develop algorithms to adapt weights to the classification problem at hand, it will become inconvenient to distinguish between weights and threshold. Like the weights, the threshold may have to be adapted and it will be awkward to have to treat the weights and threshold separately.

A simple solution consists in creating a third input whose input is always equal to 1. The weight of the third input w_3 can then be chosen to be $-\theta$. This means that we have to create a three input perceptron *without threshold* that is capable of learning the following dataset.

It is clear that the perceptron that is defined by:

$$o = \mathcal{H}(w_1 x_1 + w_2 x_2 + w_3 x_3)$$

x_1	x_2	x_3	o
0	0	1	0
0	1	1	0
1	0	1	0
1	1	1	1

Table 11.5: The dataset of the **AND** classification problem for a perceptron with three inputs without threshold.

for $w_1 = w_2 = 1, w_3 = -1.5$ classifies all points correctly and essentially performs the same computation as the perceptron from Eq 11.3.

This trick also works in higher dimensions. In general, for any arbitrary hyperplane in an N-dimensional space, we can find a corresponding one that goes through the origin of an $N+1$ dimensional space.

This is highly convenient, because computationally the calculation of a perceptron output requires the evaluation of the scalar product:

$$\boldsymbol{w}^T \boldsymbol{x}$$

Remember from Unit 1 that this is a condensed way of writing:

$$\sum_{i=1}^{N} w_i x_i$$

So, to summarise: we need a threshold or bias to obtain a working classifier but can easily implement this by concatenating the input vector with an extra unit whose value is always one, and therefore do not need to consider thresholds as anything else but an extra weight in actual calculations.

11.1 The Perceptron Algorithm

It is clear that if a dataset is linearly separable that a perceptron can be found to classify it correctly. This is true in higher dimensional spaces as well. In N dimensions, we need N weights and one threshold θ. So, we can use a perceptron with N inputs to implement any plane

$$w_1 x_1 + w_2 x_2 + \cdots w_N x_N = \theta$$

as a decision boundary, or alternatively, we can use a perceptron with $N+1$ inputs as long as we guarantee that one of the inputs is clamped to the value 1.

If a hyperplane can be found that separates data points into two classes, again, we call this dataset linearly separable and by definition, a perceptron can be found that will classify a linearly separable dataset correctly. In higher dimensional spaces the dataset becomes

more difficult to visualise, so the graphical method that we used to determine the decision boundary for the **AND** problem is no longer suitable. Ideally, we would want an algorithm that is capable of automatically finding the weights given the dataset. For linearly separable datasets a simple algorithm exists that iterates through the dataset and finds a correct set of weights in a finite number of steps.

The algorithm is of great historical significance and bears an interesting relationship to *logistic regression*, which we will discuss later in this Unit. Nowadays, it is longer the method of choice due to a few drawbacks that we will discuss below. But because it is easy to implement and can serve as a model for more complex algorithms, we will present it here.

Assume that we have D data points in an N dimensional space, and that each point has received a classification into one of two mutually exclusive classes. (This is just a way of saying that every data point belongs to either class '0' or '1', but not to both).

The algorithm is as follows. Start with a perceptron with $N+1$ inputs, one of which is clamped to value +1, and $N+1$ weights so that a threshold is unnecessary.

1. **Start**: Choose a random set of weights: $\boldsymbol{w}_0$. For later reference, denote the current set of weights by $\boldsymbol{w}_i$, so a the start $\boldsymbol{w}_i = \boldsymbol{w}_0$.

2. **Continue:** Pick a random data point $(\boldsymbol{x}_j, d_j), j = 1, \cdots, D$, where $\boldsymbol{x}_j$ is a point for which classification $d_j \in \{0, 1\}$ is given.

3. Evaluate $\boldsymbol{w}_i^T \boldsymbol{x}_j$. If $\begin{cases} \boldsymbol{w}^T\boldsymbol{x_j} < 0 & \textbf{AND} \quad d_j = 0 : \text{goto } \textbf{Continue} \\ \boldsymbol{w}^T\boldsymbol{x_j} < 0 & \textbf{AND} \quad d_j = 1 : \text{goto } \textbf{Add} \\ \boldsymbol{w}^T\boldsymbol{x_j} \geq 0 & \textbf{AND} \quad d_j = 1 : \text{goto } \textbf{Continue} \\ \boldsymbol{w}^T\boldsymbol{x_j} \geq 0 & \textbf{AND} \quad d_j = 0 : \text{goto } \textbf{Subtract} \end{cases}$

4. **Add:** $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \boldsymbol{x}$; Goto **Continue**

5. **Subtract:** $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \boldsymbol{x}$; Goto **Continue**

This is the perceptron algorithm as presented by Minsky & Papert [1]. This book is of great historical interest, not least because it has been accused of setting neural network research by decade! The presentation is clumsy (goto's!), but they key idea is present and simple: if the classification is correct, leave the weights alone, otherwise add or subtract the input pattern so that the weights may do better on the same pattern next time around. You might at this point object by saying that improving the classifier to perform better on one particular pattern may make it worse for other patterns, and that this intuition may be misguided. It is therefore important that this algorithm can be shown to converge for a linearly separable dataset.

[1] M. Minsky and S. A. Papert (2017). *Perceptrons: An introduction to computational geometry*. MIT press

The *perceptron theorem* states that for a linearly separable dataset, the algorithm visits **ADD** and **SUBTRACT** a finite number of times (and then has converged, even if according to the algorithm presented above it is stuck in a endless loop (Minsky and Papert's presentation is really clumsy).

We will present the perceptron theorem for completeness. You will not be assessed on it, but it is useful to study the reasoning behind it, and also because it shows what is being optimised during the performance of the algorithm.

In *Activity: Perceptron Algorithm* you will experiment with the perceptron algorithm.

11.2 The Perceptron Theorem

The *Perceptron Theorem* essentially states that the perceptron algorithm will always converge on linearly separable data. Multiple solutions are usually possible and the Perceptron Theorem states nothing about which solution ultimately will be found. Due to the Perceptron Theorem the perceptron algorithm is actually one of the very few provable results on neural networks. The proof is instructive: it contains a number of ideas that may help you about neural networks at a higher level and that also show up in *support vector machines*. For these reasons and its great historical importance, we present it here.

We assume that we have a data space of dimension N. We assume that we have a number of data points that can belong to two classes: $\mathcal{C}_1$ and $\mathcal{C}_2$. Each data point belongs to one and only one class. We assume that this data is linearly separable, that is, there exist N weights and a bias that separates the points of the two classes. Again, we will use vector notation. There is exist a vector $\boldsymbol{w}$ of dimension N and a bias θ such that for all $\boldsymbol{x}_i \in \mathcal{C}_1$, we have $\boldsymbol{w} \cdot \boldsymbol{x}_i - \theta \geq 0$ and for all $\boldsymbol{x}_j \in \mathcal{C}_2$, we have $\boldsymbol{w} \cdot \boldsymbol{x}_j - \theta < 0$.

We will simply this: first, we want to get rid of the bias in the way we described earlier. That is, we take every data point $\boldsymbol{x}_i$ and extend it by adding the value '1' to the sequence of numbers that is represented by the vector $\boldsymbol{x}_i$, which yields an $N+1$-dimensional vector. Similarly, we 'add' another entry to our weight vector $\boldsymbol{w}$, making it an $N+1$ dimensional vector.

The advantage of this is that linear separability becomes even easier to formulate. It is now a hyperplane *through the origin*, separating the points of the two classes in an $N+1$ dimensional space. So, there exist an $N+1$ dimensional vector $\boldsymbol{w}^*$ such that $\boldsymbol{w}^{*T}\boldsymbol{x_i} \geq 0$ for all $\boldsymbol{x}_i \in \mathcal{C}_1$ and $\boldsymbol{w}^{*T}\boldsymbol{x}_j < 0$ for all $\boldsymbol{x}_j \in \mathcal{C}_2$.

This now allows a further simplification: take a pattern $\boldsymbol{x}_j$ from class $\mathcal{C}_2$. Remove it from class $\mathcal{C}_2$ and add the pattern $-\boldsymbol{x}_j$ to class $\mathcal{C}_1$.

Repeat this for all patterns in $\mathcal{C}_2$, so that this class becomes empty and class $\mathcal{C}_1$ is extended by negative versions of patterns that were previously in $\mathcal{C}_2$.

That this is allowed should be obvious: if $\boldsymbol{w}^{*T} \geq 0$ then $-\boldsymbol{w}^{*T} < 0$. The conclusion therefore is that we can replace any two class dataset that is linearly separable by a one class dataset that consists of patterns that are all on one side of a hyperplane through the origin. This hyperplane is as yet unknown, but linear separability implies it exists.

Lastly, we normalise all input patterns $\S_i$, so that they have length 1. This does not affect the classification, as multiplication by a constant positive factor leaves the classification of $\boldsymbol{x}_i$ unchanged.

We can now work with a simplified version of the algorithm.

1. **Start**: Choose a random set of weights: $\boldsymbol{w}_0$. For later reference, denote the current set of weights by $\boldsymbol{w}_i$, so a the start $\boldsymbol{w}_i = \boldsymbol{w}_0$.
2. **Continue:** Pick a random data point $(\boldsymbol{x}_j, d_j), j = 1, \cdots, D$, where $\boldsymbol{x}_j$ is a point for which classification $d_j \in \{0, 1\}$ is given.
3. Evaluate $\boldsymbol{w}_i^T \boldsymbol{x}_j$. If $\begin{cases} \boldsymbol{w}^T \boldsymbol{x}_j \geq 0 & \text{goto } \textbf{Continue} \\ \boldsymbol{w}^T \boldsymbol{x}_j < 0 & \text{goto } \textbf{Add} \end{cases}$
4. **Add:** $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \boldsymbol{x}$; Goto **Continue**

In this setting, we can formulate precisely what we mean by linear separability:

Definition: there exists an as yet unknown weight vector $\boldsymbol{w}^*$ such that for each $\boldsymbol{x}_i \in \mathcal{C}_1$, $\boldsymbol{w}^{*T} \boldsymbol{x}_i \geq \delta$ for some $\delta > 0$. Without loss of generality, we will assume that this vector is normalised, i.e.,

$$\mid \boldsymbol{w}^* \mid = 1$$

. We can multiply $\boldsymbol{w}$ by any positive factor; this does not affect classification.

Perceptron Theorem: Imagine now that we apply the algorithm and find a sequence of weights. We start with a set of weights $\boldsymbol{w}_0$, which according to the algorithm are random numbers. We then cycle through set $\mathcal{C}_i$ trying out randomly selected patterns. Whenever the algorithm goes through **ADD**, a new set of weights will be produced, moving from $\boldsymbol{w}_i$ to $\boldsymbol{w}_{i+1}$. In this way we produce a sequence of updated weights. The perceptron theorem states that for linearly separable data this sequence is finite, i.e. at some point the algorithm will have found a set of weights that classifies all input patterns correctly.

In order to prove this, we will consider the following quantity:

$$\cos \angle(\boldsymbol{w}^*, \boldsymbol{w}_i) \equiv \frac{\boldsymbol{w}^{*T} \boldsymbol{w}_i}{\mid \boldsymbol{w}_i \mid}$$

By the definition of the scalar product, this is indeed a cosine, so we know that $0 \leq \cos \angle(\boldsymbol{w}^*, \boldsymbol{w}_i) \leq 1$. Because we do not know $\boldsymbol{w}^*$, we cannot calculate this quantity directly, but we can say something about the way it changes, when we move from $\boldsymbol{w}_i$ to $\boldsymbol{w}_{i+1}$.

We will treat the numerator and denominator separately. What happens to the numerator when we move from $\boldsymbol{w}_i \rightarrow \boldsymbol{w}_{i+1}$? We know that this must happen in the **ADD** branch so, we must have hit a pattern $\boldsymbol{x}$ that is wrongly classified.

$$\boldsymbol{w}^{*T}\boldsymbol{w}_{i+1} = \boldsymbol{w}^{*T}(\boldsymbol{w}_i + \boldsymbol{x}) = \boldsymbol{w}^{*T}\boldsymbol{w}_i + \boldsymbol{w}^{*T}\boldsymbol{x}$$

Although we do not know $\boldsymbol{w}^*$, we can say something about $\boldsymbol{w}^{*T}\boldsymbol{x}$. This quantity is positive since the pattern $\boldsymbol{x} \in \mathcal{C}_1$ and by definition $\boldsymbol{w}^{*T}\boldsymbol{x} \geq \delta$ for some $\delta > 0$. So,

$$\boldsymbol{w}^{*T}\boldsymbol{w}_{i+1} >= \boldsymbol{w}^{*T}\boldsymbol{w}_i + \delta$$

Now, δ is a property of the dataset, which is fixed, finite and positive. The numerator increases by *at least* a fixed positive amount, each time the algorithm goes through **ADD**.

What about the denominator? The square of the denominator is given by

$$\boldsymbol{w}_{i+1}^T\boldsymbol{w}_{i+1}^T = (\boldsymbol{w}_i^T + \boldsymbol{x}^T)(\boldsymbol{w}_i + \boldsymbol{x}) = \boldsymbol{w}_i^T\boldsymbol{w}_i + 2\boldsymbol{w}_i^T\boldsymbol{x} + 1$$

Here we used that the vectors in our training set are normalised. Since a weight update took place, it must have been the case that $\boldsymbol{w}_i^T\boldsymbol{x} < 0$, otherwise, no update would have taken place. We are therefore certain that:

$$\boldsymbol{w}_{i+1}^T\boldsymbol{w}_{i+1} < \boldsymbol{w}_i^T\boldsymbol{w}_i + 1$$

This implies that the quantity

$$\frac{\boldsymbol{w}^{*T}\boldsymbol{w}_i}{|\,\boldsymbol{w}_i\,|}$$

has increased by at least $\sqrt{M}\delta$ after M updates. Since this quantity, being equal to a cosine, must remain smaller than one, only a finite number of updates, at most $M = \frac{1}{\delta^2}$ can be made, otherwise we get a contradiction. This proofs the perceptron theorem.

11.3 *Loss Function for the Perceptron*

Given a linearly separable dataset, a hyperplane exists that separates the two classes, but so far we have introduced an iterative algorithm. Would it be possible, like for the linear regression problem to write

an explicit solution for the weights? The answer is no. Just like for the regression problem, we can write down a loss function, and consider the weights as parameters. Optimising the parameters such that the loss function is minimised would then produce weights for a perceptron that can classify a linearly separable dataset. Let us pursue this approach. Let us write a perceptron as:

$$o = f(\boldsymbol{w}^T \boldsymbol{x}). \tag{11.4}$$

Earlier, we used $f(x) = \mathcal{H}(x)$, but later we will consider other functions. Traditionally, the following function was introduced as a loss function in the neural network literature:

$$\mathcal{E} = \frac{1}{2} \sum_{i=1}^{D} (o_i - d_i)^2, \tag{11.5}$$

where the sum runs over all data points in our set $\mathcal{D} = \left\{ (\boldsymbol{x}_i^T, d_i \right\}$, and where $o_i = f(\boldsymbol{w}^T \boldsymbol{x}_i)$, the classification by the perceptron of *input pattern* $\boldsymbol{x}_i$. The neural network literature traditional calls o_i the *observed output* and d_i, the *desired output*.

For a linearly classifiable dataset, the perceptron theorem guarantees that a set of weights exist such that $\mathcal{E} = 0$, but the problem of minimising $\mathcal{E}$ is relatively complex.

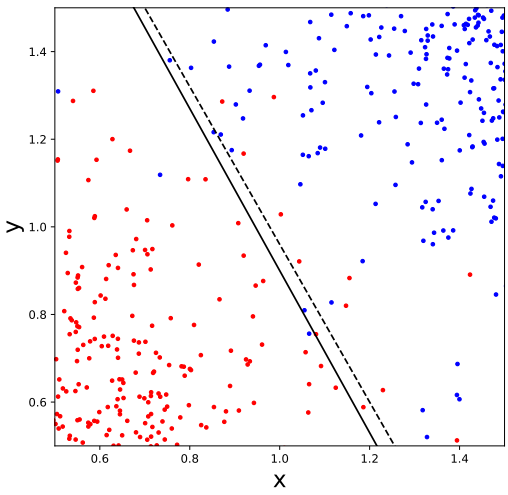


Figure 11.24: Continuous changes for the parameters $\boldsymbol{w}$ is equivalent to rotating and moving the line in a smooth manner. Whenever the decision boundary is crossed, however, $\mathcal{E}$ will jump discontinuously. The decision line for the solid line is: 1.85x + y = 2.75, for the dashed line by: 1.8x+y = 2.76. This small change causes four points to be classified differently. Each time a line that moves from the solid line into dashed one and crosses a red point, the error functions jumps. It does not matter how smoothly the transition is made.

This becomes obvious if one realises that $\mathcal{E}$ essentially counts the number of misclassifications. When the decision boundary is changed smoothly it may cross over a point that previously had been misclassified and is now classified correctly. The loss function $\mathcal{E}$

jumps. It is clear that $\mathcal{E}$ is not a continuous function of the weights for the classical perceptron. This makes it hard to write down conditions on w that minimise $\mathcal{E}$ and nearly impossible to use numerical methods such as *steepest gradient descent*.

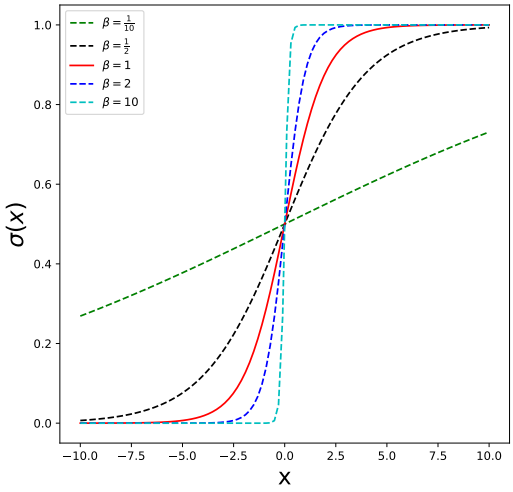


Figure 11.25: The logistic function for various values of the noise parameter β. Observe that for high values of β the function resembles a step function.

At heart the cause of this problem is the use of the Heaviside function, which is discontinuous in $x = 0$. One way of avoiding it, is to replace by a smooth function. Several such functions have been used in the past, including the logistic function:

$$\sigma(x) = \frac{1}{1 + e^{-\beta x}} \tag{11.6}$$

In the following we will use $f(x)$ for an arbitrary squashing function, whose exact form we will determine later. $\sigma(x)$ will always determine the logistic function (Eq. 11.6).

The logistic function can be considered a soft version of the step function. This means that a perceptron using it will no longer make a hard decision, but one that depends continuously on the input function. The 'hardness' of the decision can be influenced with the choice of parameter β, which is sometimes called a noise parameter. Figure 11.25 demonstrates that high values of β lead to a function $f(x)$ that almost is equivalent to a step function. Low values of β leads to a function which is flat in a considerable interval around zero input.

Another function that has been popular with very similar properties is tangent hyperbolic:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{11.7}$$

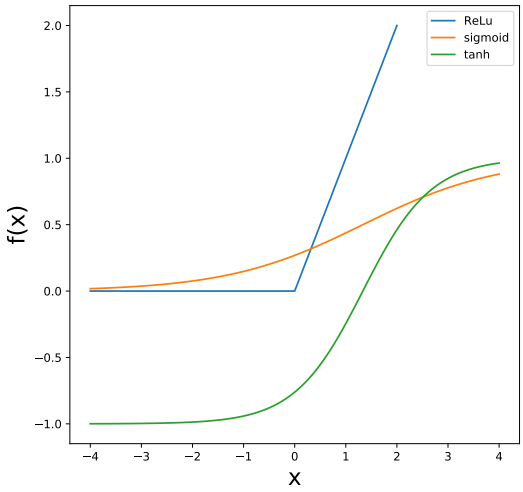


Figure 11.26: Examples of squashing functions that are commonly used in neural networks. They all contain a non-linearity, which is essential in multi-layer perceptrons.

It looks similar to the logistic function, but has the interval $[-1, 1]$ as a range whilst the logistic function has range $[0, 1]$. Moreover the tangent hyperbolic is anti-symmetric in its input. It maps 0 to 0, unlike the logistic function, which maps 0 to 0.5. Because of their s-shaped form, these functions are sometimes called *sigmoids*.

The function $f(x)$ are sometimes called *squashing functions* because they compress the input, which theoretically can be in the range $(-\infty, \infty)$ to a smaller, often finite range. ReLu, or rectified linear unit, nowadays has become quite popular is shown in Fig. 11.26.

All of the functions are continuous and almost every where differentiable. This has the important consequence that the condition for minimisation of the loss function can now be written down:

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} = 0$$

Let us work this out for the logistic squashing function. We need to apply the chain rule:

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} = \sum_i (o_i - d_i) \frac{\partial}{\partial \boldsymbol{w}} o_i. \tag{11.8}$$

Since

$$o_i = \sigma(\boldsymbol{w}^T \boldsymbol{x}_i),$$

so that

$$\frac{\partial}{\partial \boldsymbol{w}} o_i = \sigma'(\boldsymbol{w}^T \boldsymbol{x}_i) \frac{\partial \boldsymbol{w}^T \boldsymbol{x}_i}{\partial \boldsymbol{w}}$$

When we write out $\boldsymbol{w}^T\boldsymbol{x} = \sum_j w_j x_j$, it is easy to see that

$$\frac{\partial}{\partial w^i}\sum_j w^j x_j = x_i,$$

or in vector notation:

$$\frac{\partial}{\partial \boldsymbol{w}}\boldsymbol{w}^T\boldsymbol{x} = \boldsymbol{x}$$

Substituting this back into Eq. 11.8 gives:

$$\frac{\partial}{\partial \boldsymbol{w}} o_i = \sigma'(\boldsymbol{w}^T\boldsymbol{x}_i)\boldsymbol{x_i}$$

Now, for the logistic equation, it is easy to show that

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{11.9}$$

This very elegant property means that once you have calculated $\sigma(x)$ for a given value of x, calculating the derivative is a simple multiplication! This means that the condition for minimising the loss function can be written as:

$$\frac{\partial}{\partial \boldsymbol{w}}\mathcal{E} = \sum_i (\sigma(\boldsymbol{w}^T\boldsymbol{x}) - d_i)\sigma(\boldsymbol{w}^T\boldsymbol{x})(1-\sigma(\boldsymbol{w}^T\boldsymbol{x}))\boldsymbol{x}_i = 0$$

This is a non linear system of equations - the $\boldsymbol{w}$ appear in the function argument of $f(x)$, which is a non linear sigmoid - and no analytic techniques for solving them are available. We therefore have to resort to numerical techniques. We are faced with the problem of function minimisation. An often used technique is *steepest gradient descent*.

11.4 A Learning Rule for the MSE Loss Function

Imagine a function $\mathcal{E}(\boldsymbol{w})$, which we aim to minimise, but if the function is non linear, it is almost always difficult to find algebraic conditions. However, we usually can calculate the *gradient* $\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} \mid \boldsymbol{w}_0$ in some point $\boldsymbol{w}_0$. It is important to realise that the represents a direction (i.e. a vector) in weight space. Now consider that we move away from the point $\boldsymbol{w}_0$ in a direction given by a vector whose direction is unconstrained, but whose magnitude is fixed to a small value h. Depending on the direction, the function $\mathcal{E}$ will have a new value, $\mathcal{E}(\boldsymbol{w}+\boldsymbol{h})$. Calculus tells us that the gradient is *the direction of maximum change*. If we are not at a maximum, a small step in the direction of the gradient will lead to a higher value of $\mathcal{E}$. Conversely, stepping against the gradient will lead to a lower value of $\mathcal{E}$. Steepest gradient descent consists of repeatedly make small moves in the direction of the negative gradient. When repeated often enough, one may arrive

at a minimum or a saddle point. The saddle point can usually be avoided, by means explained below. Steepest gradient descent will find generally find a *local* minimum.

In general, steepest gradient can be expressed as:

$$\boldsymbol{w} \to \boldsymbol{w} - \lambda \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} \tag{11.10}$$

Here λ is called the *learning rate*. Theory sets no other constraints than that it be 'small'. Its choice in practice will have to be determined by experimentation. Too small and convergence to a minimum takes a long time, too large and it may overshoot the minimum. When λ has been chosen appropriately, repeated application of Eq. 11.10 will lead to a set of weights for which the loss function is locally minimal. Eq. 11.10 is an example of a *learning rule*, an iterative algorithm that uses parts of a dataset to improve weights for classification.

Given that we already have evaluated the gradient of the loss function for the case of a logistic squashing function, we can immediately write down the the steepest gradient descent rule:

$$\boldsymbol{w} \to \boldsymbol{w} - \lambda \sum_i o_i(1 - o_i)(d_i - o_i)\boldsymbol{x}_i \tag{11.11}$$

This is sometimes written as

$$\boldsymbol{w} \to \boldsymbol{w} - \lambda \sum_i \Delta_i x_i,$$

with

$$\Delta_i \equiv o_i(1 - o_i)(d_i - o_i).$$

This is not a great learning rule, certainly not for classification problems. We will look at improvements below. One reason is immediately obvious: the desired classification is either '0' or '1', but once the weights start to approach values such that the perceptron starts to produce these values, learning slows down, due to the factor $o_i(1 - o_i)$, which approaches 0 if o_i starts to approach 0 or 1. Nevertheless, the example is important. Here, we have seen the first example of how steepest gradient descent is employed to minimise a loss function. Apart from an overall constant, this loss function is equivalent to the Mean Squared Error (MSE) loss function that we already encountered in Unit 1.

Note that for the problem at hand, classification of data points into two classes, this learning rule is the perceptron rule in disguise.

The perceptron learning rule essentially states that the weights should not be changed if classification is correct, that the weights should be changed in the direction of the input if the desired classification is '1' and current classification is '0', and away from the

input in the opposite case. If it were not for the the factor $o_i(1-o_i)$, Eq. 11.11 states the same thing. But the factor $o_i(1-o_i)$ changes the magnitude of the correction to the weights, not its direction, which is in the direction of the input factor. And since the magnitude can be manipulated by the learning rate λ, Eq. 11.11 is not very different from the perceptron algorithm with a learning rate.

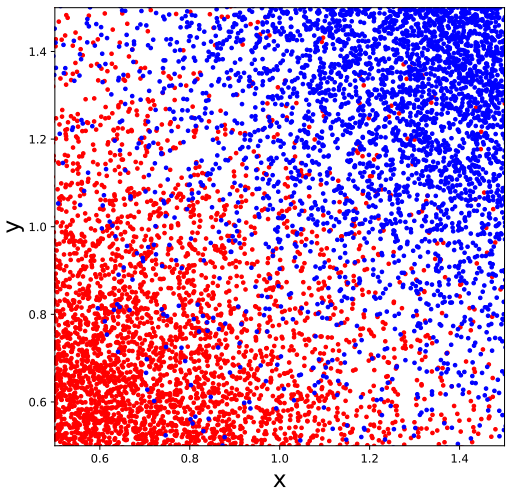


Figure 11.27: A data classification problem where a linear classifier is a reasonable solution, although technically the dataset is not linearly separable.

.

The approach presented here, steepest gradient descent nevertheless represents an improvement over the classical perceptron algorithm in two important ways.

1. The classical perceptron algorithm cannot handle data that is not linearly separable. It will just loop forever. There are genuinely complex classification problems that a single perceptron cannot be expected to handle. But consider the classification problem shown in Fig. 11.27. A linear classifier would be appropriate, but the original perceptron algorithm cannot handle it.

2. The perceptron algorithm just stops when it finds a set of weights that work. It does not try to find the 'best' weights in any sense.

By using a steepest gradient descent version of the algorithm, these problems are alleviated. First, one can *monitor* the performance of the algorithm by occasionally evaluating the loss function for the current set of weights. This makes it possible to create a *stopping criterion*. For linearly separable data one immediately stops when the loss function is zero. For non linearly separable data the loss function

can never become zero, but it will typically start to hover around a small value, at least smaller than for a purely random set of weights. A little experimentation can give a sense for what a good stopping value is. In a non linearly separable dataset the stopping criterion ensures that the solution if not perfect, is at least reasonable.

11.4.1 *Batches, Minibatches and Momentum*

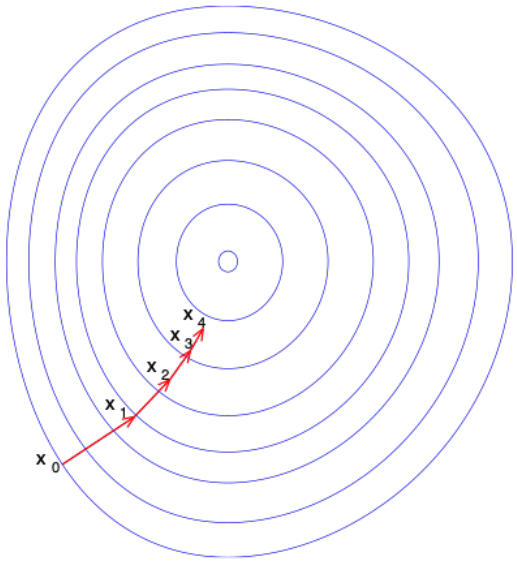


Figure 11.28: Steepest gradient descent in a favourable loss landscape. Source: Wikipedia

In the steepest gradient descent version of the algorithm the gradient was calculated over the entire dataset. Although correct according to calculus, this is usually not the most efficient way of training a neural network. The gradient only tells you what *locally* is the best way to go in weight space, but how the algorithm performs depends on the entire landscape in weight space.

For a linearly separable dataset the perceptron theorem informs us that there is a *global* minimum of the loss function. After all, a set of weights exists that allow for no misclassifications at all. Also, there are no local minima. This is also plausible if we look at Fig. 11.27. It is clear that whenever we move the decision boundary to another position, overall the number of misclassifications increases. We can think of this loss landscape as being concave, and since the loss function is quadratic in the weights, its contours resemble an ellipsoid like in Fig. 11.28.

So, in this case steepest gradient descent is guaranteed to find the minimum. However, even though there is a quadratic minimum, this

is not necessarily spherically symmetric, but the minimum may be quite elongated. In that case, the local gradient may not point in the direction.

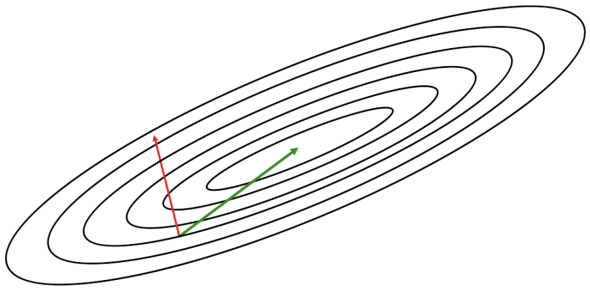


Figure 11.29: An elongated minimum may lead to slow convergence of steepest gradient descent, because the gradient does not really point to the minimum (left).

This may lead to a very slow convergence. Ideally, steepest gradient descent should traverse the contours of the loss landscape quickly, as shown in Fig. 11.28. However, elongated shapes may lead to a zigzag course in the loss landscape, due to the gradient locally not pointing to the centre. An extreme example is shown in Fig. 11.30, where due to zigzagging the algorithm almost follows a contour in the landscape, which means it is converging extremely slowly.

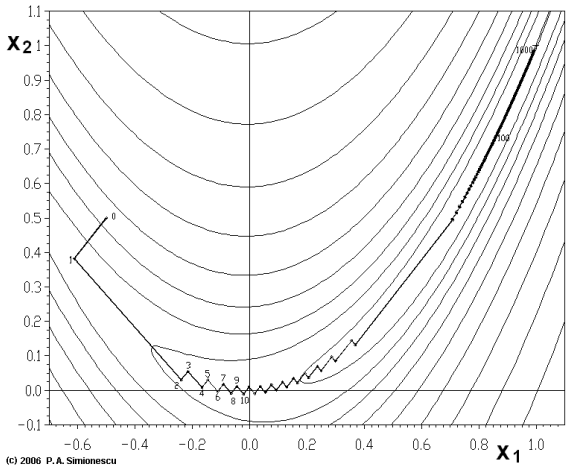


Figure 11.30: Very slow convergence of steepest gradient descent, due to zigzagging (source Wikipedia).

These problems are well known in the neural network literature. One strategy is not to use the entire training dataset in the calculation of the gradient, a procedure called *batch training*, as the entire batch of training patterns is used. The other end of the spectrum is to calculate the gradient on single data points, which is called *online learning*. If each data point is chosen randomly from the training set, this adds some random jittering to the direction of the descent, which is often useful in practice. Most neural network training adopts a

strategy somewhat in between. A number of m patterns is selected, called a *minibatch*. The gradient is then estimated over this batch. This approach is called *stochastic gradient descent*.

Another approach aiming to alleviate the zigzagging is to use *momentum*. When using momentum, the gradient that was calculated in a previous iteration of the algorithm is saved. In the current iteration the gradient is calculated again, but the change implemented of the weights is a weighted average of the 'old' and the 'new' gradient. Symbolically, the learning rule can be represented as follows:

$$\Delta \boldsymbol{w}_t = -\lambda \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} + \alpha \Delta \boldsymbol{w}_{t-1} \tag{11.12}$$

Here $\Delta \boldsymbol{w}_t$ is the current change of weights. Without momentum term ($\alpha = 0$), this is steepest gradient descent as usual.

Momentum and stochastic gradient descents are two examples of optimising. There is a large number of optimisation techniques, too large to cover here. Before you start training neural networks in anger, you should consult a recent review of modern optimisation techniques, for example Chapter 8 of [2].

[2] I. Goodfellow, Y. Bengio, and A. Courville (2016). *Deep learning*. MIT press

12 Logistic Regression

12.1 Logistic Regression: Discriminative Modelling

In Sec. 11.3 we have provided the historical argument for a perceptron with a soft decision function, which happened to be the logistic function, which was in fact selected for its neat algebraic properties, in particular the fact that the derivative can be calculated by simple multiplication when the function value is known (Eq. 11.9). This leads to elegant expressions for learning rules but not always to efficient learning.

Logistic regression does not use a hard decision function to make a decision, but models the probability of an outcome instead. The sigmoid function, which we reproduce for convenience in Fig. 12.31 is suitable for this interpretation.

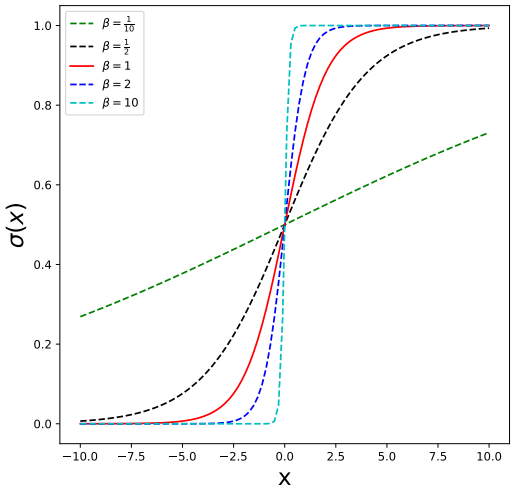


Figure 12.31: The logistic function for various values of the noise parameter β. Note that because the output is between 0 and 1, it can be interpreted as a probability.

The definition of a logistic regression classifier is now given by:

$$p(\mathcal{C}_1 \mid \boldsymbol{\phi}) = \sigma(\boldsymbol{w}^T \boldsymbol{\phi}),$$

with

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

and our task is to find weights that produce the best classification possible.

To this end, we define a likelihood function. Let the dataset be $\{\phi_i, t_i\}$ with $\phi_i = \phi(\boldsymbol{x}_i)$ and $t_i \in \{0, 1\}$. The labels t_i are given: the dataset is *labelled*. Writing $\boldsymbol{t}$ for $(t_1, \cdots t_N)$ we write:

$$p(\boldsymbol{t} \mid \boldsymbol{w}) = \Pi_{i=1}^{N} y_i^{t_i} (1-y_i)^{1-t_i}, \tag{12.1}$$

where $y_i = p(\mathcal{C}_1 \mid \phi_i) = \sigma(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})_i)$.

In line with our experiences from Unit 1, this formula simplifies when we consider the log likelihood:

$$\mathcal{E}(\boldsymbol{w}) = \ln p(\boldsymbol{t} \mid \boldsymbol{w}) = \sum_{i=1}^{N} t_i \ln y_i + (1-t_i) \ln(1-y_i) \tag{12.2}$$

This is again an error or a loss function, as the set of $\boldsymbol{w}$ that minimises it minimises prediction loss. Minimising this function with respect to $\boldsymbol{w}$ is equivalent to maximising the likelihood function. In line with our approach so far, we can calculate the gradient and perform steepest or stochastic gradient descent:

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} = \sum_{i=1}^{N} (y_i - t_i)\phi_i \tag{12.3}$$

This is a nice and tidy result. It is similar to the perceptron learning rule, note that we wrote o_i there, instead of y_i. We maintain the distinction, reflecting that o_i is traditionally used in the Connectionism literature, and y_i in machine learning but they both refer to the quantity:

$$\sigma(\boldsymbol{w}^T \phi_i).$$

If we compare the two rules, there no factor $o_i(1-o_i)$ here. This is due to the difference in loss functions Eq. 11.5 and 12.2. It is worthwhile commenting on the difference. Let us recall the definition of the Kullback-Leibler divergence from Unit 1.

$$\mathrm{KL}(p \mid\mid q) = -\int p(\boldsymbol{x}) \ln q(\boldsymbol{x}) d\boldsymbol{x} - (-\int p(\boldsymbol{x}) \ln p(\boldsymbol{x}) d\boldsymbol{x}) \tag{12.4}$$

For a discrete probability distribution this works out as:

$$\sum_{i \in C_i} p_i \ln q_i + \sum_{i \in C_i} p_i \ln p_i$$

The sum here is over the different outcomes and their probabilities - it is not a sum over events. The first term of this sum is the *cross entropy*, the second one is (self)-entropy. For the two class outcomes

$\mathcal{C}_1$ and $\mathcal{C}_2$, with probabilities p_1, and $1 - p_1$. The cross entropy of a probability distribution with two outcomes is therefore:

$$p_1 \ln q_1 + (1 - p_1) \ln(1 - q_1)$$

We remember that the KL-divergence is a measure between probability, in this case a 'true' distribution p_i and another distribution q_j. If we amend the probability distribution q_j in a way that reduces the KL-divergence, q_j will be more than p_i in a well defined manner. We can now recognise Eq. 12.2 as the cross entropy. Eq. 12.3 would allow us to design a gradient-based learning rule that ensure minimisation of the cross entropy and thereby the KL-divergence, since the second term in the KL-divergence Eq. 12.4 is a only function of the true distribution, which we use as reference.

12.2 *Generative Models and the Logistic Function*

So far, have used a graded perceptron as a way for training a classifier that makes a hard decision. Devices that take a hard decision, based on which side of the hyperplane a data point lies are called *discriminants*. However, a probabilistic interpretation of the sigmoid squashing function, which after all yields numbers between 0 and 1 leads to *logistic regression*, an important statistical technique in its own right.

Logistic regression arises naturally in the context of *probabilistic generative models* Bishop (2006). Assume that we are generating a dataset where the points belong to one of two classes $\mathcal{C}_1, \mathcal{C}_2$, and that conditional probabilities $p(\boldsymbol{x} \mid \mathcal{C}_1)$ and $p(\boldsymbol{x} \mid \mathcal{C}_2)$ as well as prior probabilities $p(\mathcal{C}_1), p(\mathcal{C}_2)$ are given. We can then simulate a dataset: we first simulate a Bernoulli process to determine which of the two classes the point belongs to, and then we use the selected conditional probability to generate the point. For example, $\mathcal{C}_1$ might correspond to a genetic factor that influences the height distribution of mature individuals. Lacking this factor, individuals belong to a class $\mathcal{C}_2$, whose individuals may have a different height distribution. Often, $p(x \mid \mathcal{C}_1)$, $p(x \mid \mathcal{C}_2)$ are known and we are faced with the problem of how likely it is that a given individual of height x_i belongs to $\mathcal{C}_1$ or $\mathcal{C}_2$, when all the information we have available is the person's height.

Bayes' law gives an answer, since:

$$p(\mathcal{C}_1 \mid x) = \frac{p(x \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(x \mid \mathcal{C}_1)p(\mathcal{C}_1) + p(x \mid \mathcal{C}_2)p(\mathcal{C}_2)} \tag{12.5}$$

This can be rearranged to:

$$p(\mathcal{C}_1 \mid x) = \frac{1}{1 + \exp(-a)}, \tag{12.6}$$

with

$$a = \ln \frac{p(x \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(x \mid \mathcal{C}_2)p(\mathcal{C}_2)} \tag{12.7}$$

In Eq. 12.6, we see that the sigmoid function emerges quite naturally.

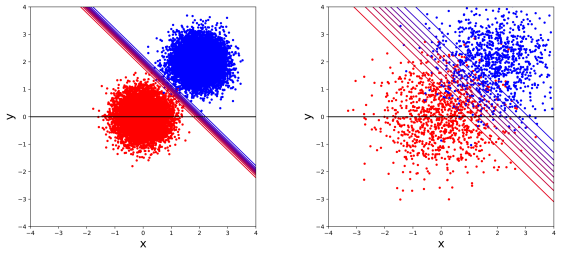


Figure 12.32: Two stochastic processes each generate data points that are Gaussian distributed. The red dots belong to class $\mathcal{C}_1$, the blue dots to class $\mathcal{C}_2$. Isolines for the probability $p(\mathcal{C}_1) = 0.1 \cdots 0.9$ (probability for a point being 'red') are given, calculated using Eq. 12.11. When we restrict x_2 to 0 (black horizontal line), a one dimensional sigmoid emerges.

As an example, consider Gaussian class-conditional functions:

$$p\left(\mathbf{x} \mid \mathcal{C}_k\right) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\mathbf{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}\left(\mathbf{x}-\boldsymbol{\mu}_k\right)^{\mathrm{T}} \mathbf{\Sigma}^{-1}\left(\mathbf{x}-\boldsymbol{\mu}_k\right)\right\} \tag{12.8}$$

A two-dimensional two class example is given in Fig. 12.32, where two Gaussian blobs are visible. When presented with a new data point $\boldsymbol{x}$, what is the probability that it belongs to class 1? From Eq. 12.6 and 12.7 we find:

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + w_0) \tag{12.9}$$

with:

$$\begin{aligned} \mathbf{w} &= \mathbf{\Sigma}^{-1}\left(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\right) \\ w_0 &= -\frac{1}{2}\boldsymbol{\mu}_1^{\mathrm{T}}\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2^{\mathrm{T}}\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_2 + \ln\frac{p\left(\mathcal{C}_1\right)}{p\left(\mathcal{C}_2\right)} \end{aligned} \tag{12.10}$$

The example uses $\boldsymbol{\mu}_1^T = (0,0)$ and $\boldsymbol{\mu}_2^T = (2,2)$ and a covariance matrix of the form:

$$\mathbf{\Sigma} = \begin{pmatrix} k & 0 \\ 0 & k \end{pmatrix}$$

If we assume that there are as many points in class $\mathcal{C}_1$ as there are in class $\mathcal{C}_2$, i.e. $p(\mathcal{C}_1) = p(\mathcal{C}_2)$, then one can calculate that for an arbitrary point $\boldsymbol{x}^T = (x_1, x_2)$ the probability of being a 'red' point is given by:

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \sigma(-\frac{2}{k}(x_1 - x_2 - 2)) = \frac{1}{1 + e^{-\frac{2}{k}(x_1 + x_2 - 2)}} \tag{12.11}$$

This result is plausible: the line were the probability $P(\mathcal{C}_1 \mid \boldsymbol{x}) = 0.5$, is given is exactly the centre line between the two clusters. The noise factor $\beta = \frac{2}{k}$ is large for small k. This too makes sense, if k

is small, the clusters are tight and there will not be many points that overlap. A large β will correspond to a hard decision (see Fig. 11.25). If k is large, the clusters extend and will partially overlap. β will be small in this case, corresponding to a softer decision, as is appropriate when points could come from either cluster. We make this explicit in Fig. 12.32, where we plot iso probability lines for $P(\mathcal{C}_1) = 0.1, \cdots, 0.9$. If we restrict x_2 to 0 (black horizontal line in plots), then a one dimensional sigmoid remains with a noise factor β that is inversely proportional to k, the 'width' of the blobs.

In the calculation of Eq. 12.10 we have relied on both classes having the same covariance matrix $\mathbf{\Sigma}$ associated with them, which leads to a cancellation of terms quadratic in $\boldsymbol{x}$. If this is not the case, a quadratic discriminant emerges, as explained in Chapter 4 of Bishop (2006). We will not pursue this here, but it does make the case for the introduction of basis functions that can be higher order polynomials, just as in the case of liner regression in Unit 1.

Exercise: Use Eq. 12.9 and 12.10 to derive this result.

Note that the graded form of the perceptron, that is a perceptron with a continuous squashing function, emerges here from probabilistic considerations. Also note that the output here has a natural interpretation as a probability, which originates from Bayes' rule. In order to base a *decision* on class membership we still need a rule to decide how probabilities convert to class membership. A natural boundary would be to assign a data point to $\mathcal{C}_1$ as soon as $p(\mathcal{C}_1) \geq 0.5)$ but other decisions *could* be made, for example if $\mathcal{C}_1$ corresponds to patients having a particular decease, one may want to reduce the probability of false negatives, accepting a larger probability of false positives. In that sense, a probabilistic interpretation is more subtle than the hard decision taken by the original perceptron. This was a discriminant, i.e. a function that simply returns a decision that no longer contains information about the underlying probabilities.

Imagine we are given a dataset with points being classified as belonging to one of two classes, i.e. a labelled dataset. Two build a classifier based on Eq. 12.10 we would have to estimated means and covariances of both classes. We can do this with maximum likelihood estimation (MSE). Denote the dataset by $\{\boldsymbol{x}_n, t_n\}$, where $t_n = 1$ if data point n belongs to class $\mathcal{C}_1$ and $t_n = 0$ if it belongs to class $\mathcal{C}_2$. A tedious but straightforward calculation gives MSE estimators for the mean and covariance:

$$\boldsymbol{\mu}_1 = \frac{1}{N_1} \sum_{i=1}^{N} t_i \boldsymbol{x}_i, \tag{12.12}$$

where $N_1 = \sum_{i=1}^{N} t_i$, and

$$\boldsymbol{\mu}_2 = \frac{1}{N_2} \sum_{i=1}^{N} (1 - t_i) \boldsymbol{x}_i, \tag{12.13}$$

with $N_1 + N_2 = N$, so that these simply are the mean of all points assigned to class $\mathcal{C}_1$ and $\mathcal{C}_2$.

The covariance matrix for both classes can be found by:

$$\begin{aligned}
\mathbf{S} &= \frac{N_1}{N}\mathbf{S}_1 + \frac{N_2}{N}\mathbf{S}_2 \\
\mathbf{S}_1 &= \frac{1}{N_1} \sum_{i \in \mathcal{C}_1} (\mathbf{x}_i - \boldsymbol{\mu}_1) (\mathbf{x}_i - \boldsymbol{\mu}_1)^{\mathrm{T}} \\
\mathbf{S}_2 &= \frac{1}{N_2} \sum_{i \in \mathcal{C}_2} (\mathbf{x}_i - \boldsymbol{\mu}_2) (\mathbf{x}_i - \boldsymbol{\mu}_2)^{\mathrm{T}}
\end{aligned}$$

This result is quite plausible, given the result of Unit 1 for MSE of a single Gaussian.

We can now use Eq. 12.10 to find the weights. This elaborate process of finding weights requires a two-step process: first to find the parameters of the class conditional probabilities and second to use them to calculate the weights. This is clearly a more complex process than traditional logistic regression, which you will have seen in the *Data Science* module, and to which we will return later. In the next section, we will explain why generative modelling is important.

12.2.1 *Generative Models*

What have seen that two processes controlled by conditional Gaussian probability functions can lead to a class probabilities that are described by the logistic function. In a way, we have simulated the data generation process that leads to the logistic function, and we can infer its parameters this way. This means that we have to set up a relatively complicated workflow. From the data, we need to determine $2M$ parameters for the means and $M(M+1)/2$ parameters for the covariance matrix, where M is the dimension of the data. In total, this gives us $M(M+5)/2+1$ parameters. This number increase rapidly with M, and for higher dimensional datasets it is better to infer the weights directly, using *logistic regression* as a *discriminative model*, which we will discuss in Sec. 12.1.

Given the relative simplicity of *discriminative* modelling you night wonder why we bother with generative models. For many data science applications, discriminative logistic regression is adequate: for example death within 5 years can be considered to be a binary variable, and discriminative logistic regression can be an adequate way

to develop a predictor. A model of the true causes underlying this outcome may require considerable insight in the disease trajectory itself, and may be very difficult to construct. Finding the weights for a logistic regressor may yield a valuable prediction model without claiming any insight in the true causes.

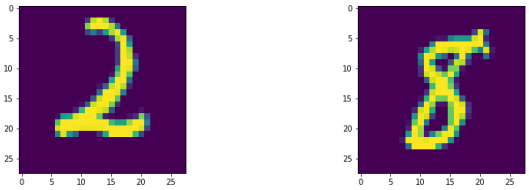


Figure 12.33: Two numerals, handwritten by humans; they are part of the MNIST dataset.

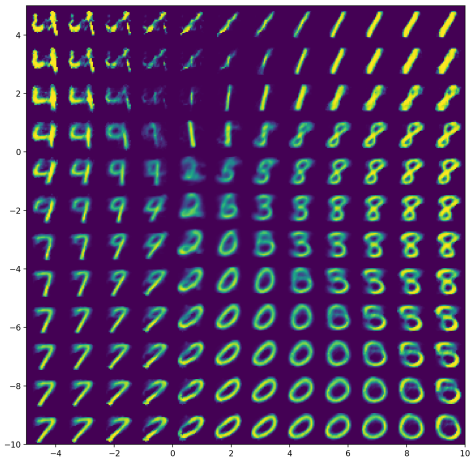


Figure 12.34: Images that have been sampled from a generative model of human handwriting.

Yet, the construction of generative models is arguably one of the major developments of the last decade in artificial intelligence. You will have seen examples of synthetically generated faces that look like interpolations of real people. A simpler example is shown in Fig. 12.33 where a number of handwritten numerals are shown together with one that has been synthetically generated (Fig 12.34).

The image generation process, to which we will return in Unit 5 and 6, considers the space of all possible pixel values. The images shown consist of 28×28 pixels. For simplicity, assume that pixels are black or white (in the actual dataset they are grey-scale values, but this does not really affect the argument). The space of all possible images, $\mathcal{I}$, where each separate image is a different point, has dimen-

sion 2^{784}. In generative modelling, it is assumed that an unknown probability distribution $p(\boldsymbol{x})$ exists over the space $\mathcal{I}$. This probability distribution assigns a small positive probability to each image that can plausibly represent a handwritten numeral, and probability zero to images that do not. Next, a number of humans actually produce handwritten numerals, that are digitised and brought into the 28×28 format. These images are collected in the so-called MNIST dataset `http://yann.lecun.com/exdb/mnist/`, an important dataset for benchmarking machine learning algorithms. These human-drawn images are then considered to be samples, drawn from distribution $p(\boldsymbol{x})$, which are then used to used to learn an approximation to this function. There are many ways of doing this. One of them learns a mapping from images to a set of Gaussian distributions through a so-called *encoder* network. A user can then sample from this Gaussian and feed the sample into a *decoder* network that converts the sample into an image.

We will later encounter examples how *encoder-decoder* networks are trained, but this architecture is now a core element of artificial intelligence techniques. In some such techniques, such as *variational autoencoders* the generative model described above is still recognisably present. It is hard to overestimate its importance.

12.3 Fixed Basis Functions

Before starting with the formalism, it is useful to realise that just like in linear regression, discussed in Unit 1, we often do not regress on the data points directly, but on fixed functions of the data points called *basis functions*. The motivation is straightforward and illustrated in Fig. 12.35: The original of dataset, shown in the left figure does not even appear to be connected, but a simple transformation of the data renders it linearly separable, as shown on the right. For this reason, we develop the formalism on basis functions of the data points rather than the data points themselves. Mathematically, it is just as complicated to work with fixed basic functions as directly with the data points as we shall see.

How does one find these basis functions? Often they suggest themselves: the transformation in Fig. 12.35 would be hard to miss and any data scientist should be on the lookout for opportunities to simplify their problem. But here, neural networks come into their own. They can be seen as methods that are capable of learning basis functions that are appropriate for a given dataset. We will give examples when we have developed the *multi-layer perceptron* later.

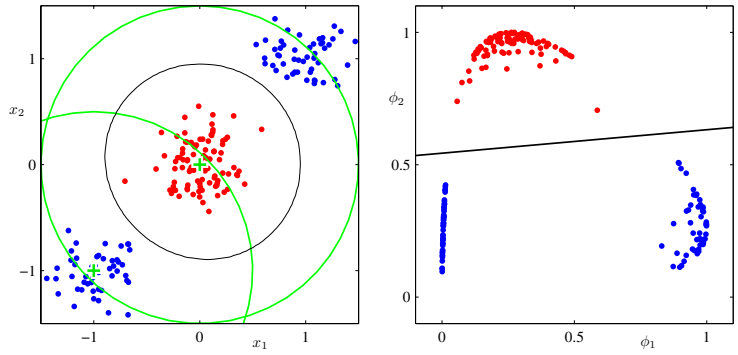


Figure 12.35: A well chosen non linear transformation of the data can often reduce in a much simpler classification problem. Two 'Gaussian' basis functions are shown, their centres represented by green crosses, and contours by green circles in the left hand plot. The right hand plot shows the data in feature space (ϕ_1, ϕ_2). Here the problem is linearly separable. This figure is Fig. 4.12 from Bishop (2006).

12.4 *Multi-class Logistic Regression*

Often, we want to create classifiers that can handle more than one class. An example is the so-called iris dataset. It is possible to distinguish three varieties of irises, *virginica*, *setosa* and *versicolor* by measuring their sepal length, sepal width, petal length and petal width (SL, SW, PL, PW). The dataset is a well known toy problem in machine learning. It is shown in Fig. 12.36. Suppose we would want to build a classifier that accepts the four numbers (SL, SW, PL, PW) and predicts one of the three varieties *setosa*, *virginica*, *versicolor* Source: Wikipedia.

The iris dataset can serve as a model here: any iris in the dataset belongs to one and only one class, but there are three classes. In

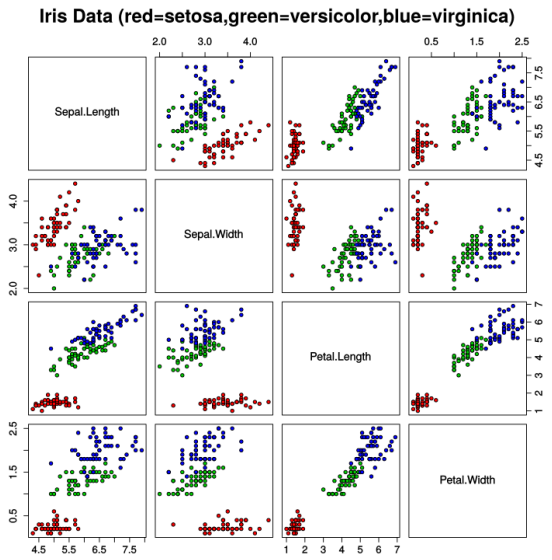


Figure 12.36: The Iris dataset represented, wastefully, as a set of scatter plots between the four dimensions of each flower.

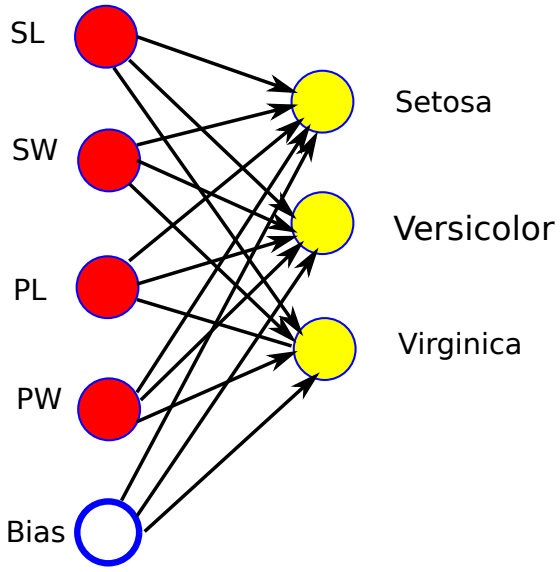


Figure 12.37: A two-layer neural network built from three perceptrons, run in parallel. It has four input nodes: *petal length*, *petal width*, *sepal length*, *sepal width* and a single bias node, whose value is always 1. Each perceptron takes a yes/no decision about one of the three classes: setosa, versicolor and virginica. If each of the perceptrons were perfect, a one hot encoding would emerge were the activation of a single node would indicate which iris variety the input dimensions belong to. The perceptron trying to recognise versicolor will not work well.

multi-class logistic regression the goal is for a given data point to find the probability that it belongs to class $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$. This gives three probability values which must sum to 1. For the K-class case, one finds, taking into account the possible use of basis functions:

$$p(\mathcal{C}_k \mid \boldsymbol{\phi}) = y_k(\phi) = \frac{\exp a_k}{\sum_j \exp a_j}, \tag{12.14}$$

with

$$a_k = \boldsymbol{w}_k^T \phi.$$

Eq. 12.14 is called a *soft-max function*. It is not hard to verify that the $0 < y_k < 1$ and $\sum_k y_k = 1$, so they have a natural interpretation as probabilities. Overflows can sometimes occur in numerical evaluations of Eq. 12.14 as the exponentials may evaluate to large numbers. This problem is easy to avoid if you are aware of it. The notebook *Logistic Regression on MNIST* gives an example of a safe implementation.

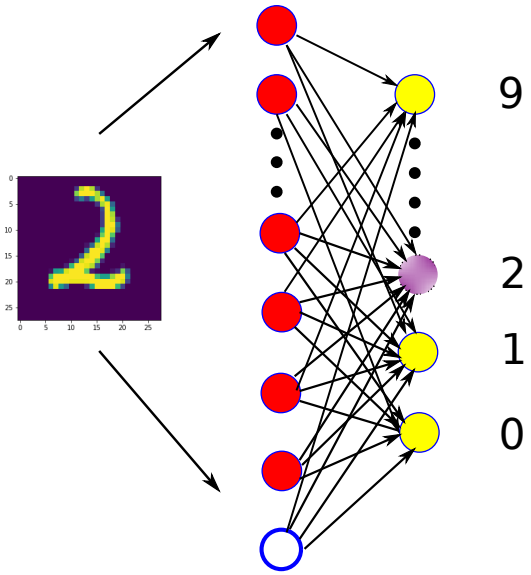


Figure 12.38: An example of multi-class logistic regression. There are 785 input variables: $28 \times 28 = 784$ pixels and a bias node. There are ten classes. The desired output is a 1-over-10 ('one hot') representation of the number that the regressor believes the image represents. For the regressor the spatial representation of the image does not play a role. The input image is 'flattened' to a vector.

In classification problems, a one-hot encoding, also called one-over-k encoding is natural for labelled data. For example, for the iris data set *setosa*, *virginica* and *versicolor* may be represented as $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$, which may be compared to softmax outputs as these to can be interpreted as a set of probabilities. Another example is the MNIST dataset, where a one-over-10 encoding is used to provide labels for image classification. The regressor is trained to infer the numerical value that is represented by the image using the labels provided as part of the dataset. As Fig. 12.38 shows, the regressor may be interpreted as a two-layer neural network.

Since the softmax outputs are meant to represent probabilities and add to one, in the derivation of the gradient there is a subtlety in that the output values are not independent. In order to find the gradient, one needs the derivative of the output values with respect to the activation values. This is given by:

$$\frac{\partial y_k}{\partial a_j} = y_k(\delta_{kj} - y_j), \tag{12.15}$$

where δ_{kj} are the components of the identity matrix.

Also, the likelihood function requires adaption. We adopt the 1-of-K coding scheme. For K nodes there will be one that is active '1', whilst the others are silent 'o'. So the target vector t_i for a feature vector $\boldsymbol{\phi}_i$ belonging to class $\mathcal{C}_k$ is a binary vector with all elements zero, except for element k, which equals one.

The likelihood function is then given by:

$$p(\boldsymbol{T} \mid \boldsymbol{w}_1, \cdots, \boldsymbol{w}_k) = \Pi_{k=1}^{K} \Pi_{i=1}^{N} p(C_k \mid \boldsymbol{\phi}_i^{t_{ik}}) = \Pi_{k=1}^{K} \Pi_{i=1}^{N} y_{ik}^{t_{ik}}.$$

Here $y_{ik} = y_k(\boldsymbol{\phi}_i)$ and $\boldsymbol{T}$ is an $N \times K$ matrix of target variables with components t_{ik}. The negative log likelihood is given by:

$$E(\boldsymbol{w}_1, \cdots \boldsymbol{w}_K) = -\ln p(\boldsymbol{T} \mid \boldsymbol{w}_1, \cdots \boldsymbol{w}_K) = -\sum_{k=1}^{K} \sum_{i=1}^{N} t_{ik} \ln y_{ik}.$$

The gradient follows from a calculation that is similar to the two-class case:

$$\frac{\partial E}{\partial \boldsymbol{w}_j} = \sum_{i=1}^{N} (y_{ij} - t_{ij}) \boldsymbol{\phi}_i.$$

Having calculated the gradient, we can apply *stochastic gradient descent* as earlier. We give an example in the notebook *Logistic Regression on MNIST*, where we build a classifier that can recognise the number represented by a handwritten image with reasonable (87 %) accuracy. In the notebook, you can look at images that are wrongly classified, which is instructive.

The second order methods work, in principle, but if the number of classes increases, straightforward IRLS becomes impossible as the memory demands scale as $D(C-1) \times D(C-1)$, where D is the number of data points and C the number of classes. The matrices for a 10-class logistic regression problem are 25 times as large as for a 2-class problem. Section 8.3.7 of [1] discusses this problem and mitigating strategies.

[1] K. P. Murphy (2012). *Machine learning: a probabilistic perspective*. MIT press

In *Activity MNIST Logistic Regression* you will investigate whether the iris dataset can be classified using logistic regression.

12.4.1 *A Comment on the Choice of Loss Functions.*

Throughout logistic regression, we have used the cross entropy as a loss function. Could we have used the MSE function? Yes, but this is treating classification as regression. Although the one-over-K vectors can be used as a target for regression, these values have no probabilistic interpretation. They are just numbers. In the softmax output and the cross-entropy loss function, the probabilistic interpretation is baked in. Moreover, we have seen in Unit 1 that predicting outputs that are unlikely to have been generated by the true distribution lead to very high costs. For classification the cross entropy is the better choice of loss function.

13 Naïve Bayes

13.1 Introduction

Naïve Bayes is a way of building fast and cheap classifiers by making an assumption about the data generation mechanism that is completely unrealistic in general, but that often achieves decent classification results. Historically, an important application has been spam filters: adaptive classifiers that are able to pick up spam messages if particular messages are flagged often enough as spam by a human. They can be trained online, meaning they can continue to train on novel messages. The naïve Bayes assumption states that events are independent conditional on the class that belong to. This statement probably needs unpacking. We already have encountered one example of the naïve Bayes assumption: the iris dataset. Consider, again Fig. 12.36: we see that there are three classes corresponding to the three iris varieties and that a data point comprises four features and a classification. The naïve Bayes assumption states that the probabilities for generating these data points are independent conditional on class. Mathematically, this means that:

$$p(\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_{N_i} \mid \mathcal{C}_i) = p(\vec{x}_1 \mid \mathcal{C}_i) p(\vec{x}_2 \mid \mathcal{C}_i) \cdots p(\vec{x}_{N_i} \mid \mathcal{C}_i). \qquad (13.1)$$

Here $\vec{x}_i$ is a four-dimensional vector (P_L, P_W, S_L, S_W) and $\mathcal{C}_i$ is one of the the three classes that the flower represented by data point i belongs to. So there are three classes: $\mathcal{C}_1 =$ virginica etc. Eq. 13.1 hides a number of assumptions about the data generation mechanism. First, we assume that we can find probability density functions that are valid for each class individually, meaning that we assume that we can find independent data generation mechanisms for each iris variety. This is not unreasonable: there is probably a lot that the different varieties have in common, but there must be some biological factor that controls which variety the flower will belong to. A second, much stronger assumption is represented by Eq. 13.1: the in-class probabilities for generating flower dimensions are independent and described by the same distribution $p(\vec{x}_i \mid \mathcal{C}_i)$. This is a strong assumption: it states that the data generation for each data point is independent

of another data point once the class is determined and that you can sample a single distribution for a given iris variety to obtain data points that are representative for that variety. In more technical language: every pair of features is conditionally independent given class membership.

The distributions of the three coloured markers are blobs that are somewhat reminiscent of Gaussian distributions. Thus inspired, we might guess that:

$$p(x_i \mid \mathcal{C}_i) = \mathcal{N}(\mu_i, \Sigma_i),$$

that is: each of the blob is described by a Gaussian distribution with a different mean and covariance for each of the three classes. Now we can use Bayes' law:

$$\begin{aligned} p(\mathcal{C}_i \mid \vec{x}_1, \vec{x}_2, \cdots \vec{X}_N) \mid \mathcal{C}_i) &\sim p(\vec{x}_1, \vec{x}_2, \cdots \vec{x}_N \mid \mathcal{C}_i) p(\mathcal{C}_i) \\ &\sim \Pi_{i=1}^{N} \mathcal{N}(\vec{x}_i \mid \mathcal{C}_i) p(\mathcal{C}_i) \end{aligned} \tag{13.2}$$

In particular, this is true for a single data point $\vec{x}$:

$$p(\mathcal{C}_i \mid \vec{x}) \sim p(\vec{x}|C_i) p(\mathcal{C}_i)$$

This now opens the way for a very simple classification procedure:

1. For each class estimate the prior probabilities $p_{virginica}$, p_{setosa}, $p_{versicolor}$, simply by counting the number of entries for each of the varieties and dividing by the number of total entries. For the iris dataset there are 50 data points of each class so these prior probabilities are equal to $\frac{1}{3}$.
2. For each of the varieties estimate the means and the covariance matrices. The standard estimators introduced in unit 1 will do, but *scikit-learn* or *scipy* offer functions to do this.
3. With these estimates you calculate $\mathcal{N}(\vec{x}\mu_{virginica}, \Sigma_{virginica}) p_{virginica}$, $\mathcal{N}(\vec{x}\mu_{setosa}, \Sigma_{setosa}) p_{setosa}$, and $\mathcal{N}(\vec{x}\mu_{versicolor}, \Sigma_{versicolor}) p_{versicolor}$.
4. Whichever of these three is largest determines which class $\vec{x}$ should belong to.

The resulting classifier is quite good. We show the result of a notebook *A Naïve Bayes Classifier for the Iris Dataset* in Fig. 13.39. Exploration of this notebook will demonstrate the workings of a naïve Bayes classifier for this dataset.

13.1.1 *Naïve Bayes and Logistic Regression*

The naïve Byes classifier described in the previous section is equivalent to the generative interpretation of logistic regression. Imagine

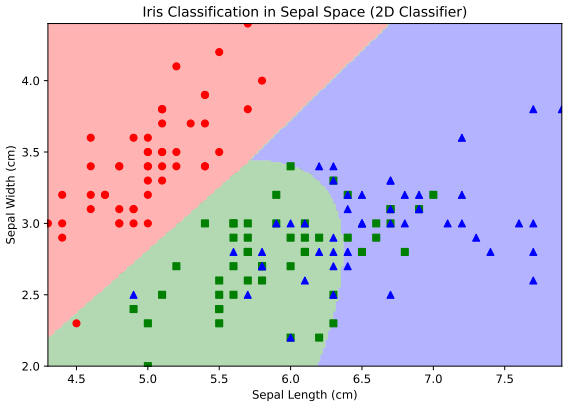


Figure 13.39: A 2D scatter plot of the sepal length/sepal width feature of the iris dataset. The markers indicate different iris varieties. A 2D naïve Bayes classifier shows the different classification volumes and shows decent performance. The 4D version, which uses all four features performs even better but does not have a simple visualisation.

that we would not be faced with a three-class classification problem as in the case of the iris dataset, but with a two-class problem. Then we would have exactly the method described in Sec. 12.2. There too, we assumed a generative model defined by two Gaussian distributions, one for the 'red' class of points and one for the 'blue' class of points, and we proved mathematically that this leads to a logistic regressor.

For the three-class classification it is more work, but here too you can show that it leads to a logistic classifier. Please be aware of the distinction between the naïve Bayes assumption and the generative model itself. The generative model is just an algorithm for constructing a dataset. The naïve Bayes assumption is an assumption about real data, in this case that the iris dataset can be reasonably described by a generative model that is class dependent but that has independent probabilities for data generation within any class.

13.1.2 *A Spam Filter*

The naïve Bayes assumption does not look completely ridiculous. Here, we will show an application where it is highly questionable, but nonetheless leads to a good classifier. Let's consider the case of whether we want to establish whether an email is genuine ('ham') or spam ('spam'). This means that we two classes $\mathcal{C}_{spam}$ and $\mathcal{C}_{ham}$. Now consider a very simple vocabulary: ['profit','sex', 'curtain',

'phone', 'notebook', 'hair extension', 'vindaloo']. This is now a word list: $w_i, i = 0, \cdots, 6$. E.g. w_2 = *curtain*.

According to the *bag of word* model, every message can be written as a binary feature vector. Simply look at a message, determine if the w_i is present. If it is, set a variable $b_i = 1$, otherwise $b_i = 0$. Do this for all words in your vocabulary and your message will be transformed in a binary feature vector $\vec{b}$. So the text message: "Your notebook is part of your hair extension" would translate into the vector $\vec{b} = (0,0,0,0,1,1,0)^T$. Importantly, the order in which the elements of the vector are kept doesn't matter. There is no attempt to preserve the original word order from the message.

Imagine that we have a dataset of text messages that we now can represent as a collection of such vectors. So we can 5 'ham' messages and 4 'spam' messages.

$$B_{ham} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The spam messages are represented by:

$$B_{spam} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Your very welcome to speculate on the actual text content. This is not necessarily a realistic example, but it serves to identify the main probabilities that play a role.

A reasonable estimate is:

$$p_{ham} = \frac{6}{10}$$

and

$$p_{spam} = \frac{4}{10}$$

It is equally easy to estimate the probabilities $P(w_i \mid C_j)$. As an example, look at $P(w_1|'ham') = \frac{3}{6}$. Why? Add the number of ones in the first column of B_{ham}. We see that word 1 occurred three times out of a total of 6. You should be able to verify easily that:

We now consider the following generative model: for each word in the vocabulary, the probability of the word being absent ($b_i = 0$) or present ($b_i = 1$) is determined by a Bernoulli process. The generative process works as follows:

Word	Ham	Spam
1	3/6	2/4
2	1/6	3/4
3	4/6	3/4
4	5/6	1/4
5	1/6	3/4
6	2/6	2/4
7	3/6	1/4

Table 13.6: The probabilities $P(w_i \mid C_j)$ for $i = 1, \cdots, 7$, $j =$ ham, spam, estimated from the feature representations of the set of messages.

1. Decide whether a message is 'ham' or 'spam'. This is a Bernoulli process where the probability of $x = 1$, i.e. the message is 'spam' is given by $P('ham')$.

2. The class $\mathcal{C}_i$ is now given as the outcome of the sampling process: $i = 0$: 'ham'; $i = 1$: 'spam'. The probability for generating a single word is again given by the Bernoulli process, and the *bag of words* assumption states that these probabilities should be independent, meaning that we can write down the probability of generating a feature vector given the class as:

$$p(\boldsymbol{b}|\mathcal{C}_j) = \Pi_{i=1}^{N} p(w_i|\mathcal{C}_j)_i^b (1 - p(w_i|\mathcal{C}_j)^{1-b_i}$$

Here N is the number of words in the vocabulary and i labels the words in there.

13.2 *The Naïve Bayes Assumption*

The *Naïve Bayes* assumption is strongly related to the bag of words model, but Naive Bayes is an assumption about how real messages are generated. The assumption is that also in real emails the features are independent when conditioned on class. How does that help?

What we're really interested in is:

$$p(C_i \mid \boldsymbol{b}),$$

Now $\boldsymbol{b}$ actually stands for a real e-mail, reduced to its features. or rather we are interested in whether

$$p(C_1 \mid \boldsymbol{b}) > p(C_0 \mid \boldsymbol{b})$$

because if that's the case we classify the message as 'spam', and if it's not we label it as 'ham'.

By now your instinct should be to invoke Bayes' law:

$$p(C_i \mid \boldsymbol{b}) \sim p(\boldsymbol{b} \mid C_i) p(C_i)$$

This ignores the normalisation constant, something we will deal with below. It is now important to realise that $\vec{b}$ is the feature vector of a

real e-mail message, s o $p(\boldsymbol{b} \mid C_i)$ is the probability of an actual e-mail message occurring. It is difficult to estimate the probability in practice: an e-mail may be unique.

The *naïve Bayes assumption* is that features are independent given the class, i.e. conditionally independent. This should immediately raise suspicions. It states that if we know that a message is 'ham', the probabilities for each of the vocabulary words occurring are independent of each other. This is patently untrue in the case of natural language, which is why this assumption is naïve. Take a moment to find counterexamples that demonstrate this. For classification purposes, nonetheless, t his is ignored and the assumption is made anyway. It turns out that for classification purposes this often works reasonably well.

The naive Bayes assumption here states that:

$$P(\boldsymbol{b} \mid C_j) = \Pi_{i=1}^{N} P(b_i \mid C_j) = \Pi_{i=1}^{N} P(w_i \mid C_j)^{b_i} (1 - P(w_i \mid C_j)^{1-b_i}$$

Now the probability for finding a certain message, which we cannot reasonably be expected to estimate, has been replaced by products of probabilities that certain words are occurring, which we can estimate.

Note that the formulae we arrived at are the same as for the generative model, but that the ideas are different. In the generative model, we made a *bag of word* assumption, deliberately generating documents that do not really look like real messages. They are really only realistic in that the word frequencies in ham or spam messages are about right. In a generative model, we are free to make such assumption.

The naive Bayes assumption is an assumption about real natural language in real email messages. This assumption is wrong and only defensible on the basis that it seems to do a reasonable job in classification. It is important to realise that both ideas lead to the same formulae but are fundamentally different.

13.2.1 *The Bayesian Two-class Classification Problem*

As you will have seen in the notebooks of Unit 1, the determination of posterior probabilities using Bayes' Law requires the determination of a normalisation constant, which can be tricky in practice. In a two-class classification problem we can bypass this problem, because we're only interested in which of the two posterior probabilities is the larger one and we do not really care about their numerical value. We will show this in detail here.

In spam classification we are interested in the function $p(C_i \mid w_1, w_2 \cdots w_N)$ (what is the the probability that this is spam given this

set of features?). Bayes then tells us that:

$$p(C_1 \mid w_1, w_2 \cdots w_N) = \frac{p(w_1, w_2 \cdots, w_N \mid C_1)}{p(w_1, w_2, \cdots w_N)} p(C_1)$$

and similarly:

$$p(C_2 \mid w_1, w_2 \cdots w_N) = \frac{p(w_1, w_2 \cdots, w_N \mid C_2)}{p(w_1, w_2, \cdots w_N)} p(C_2)$$

Now observe something that is generally true in two-class classification problems: the normalisation factor is identical in both cases. We can make a decision, 'spam' or 'ham', based on whether the posterior probability is larger for ham or for spam. In other words, we can base that decision on which of the two is the larger one and we can get infer that from dividing both equations and observe whether this ratio is larger or smaller than one:

$$\frac{p('spam' \mid w_1, w_2, \cdots, w_N)}{p('ham' \mid w_1, w_2, \cdots, w_N)} = \frac{p(w_1, w_2 \cdots, w_N \mid' ham')p(('ham')}{p(w_1, w_2 \cdots, w_N \mid' spam')p(('spam')}$$

The overall normalisation constant drops out and can be ignored. If this ratio is larger than 1, we decide 'spam', otherwise we decide 'ham'. This idea applies generally to any Bayesian analysis of a two-class classification problem and is not restricted to naive Bayes scenarios.

13.3 *Putting it All Together*

When we put everything together we find that once we have estimated the prior probabilities $p_{'ham'}$ and the probabilities $p_{'spam'}$, which we simply do by counting the number of spam and ham messages and dividing by the total, and once we have estimated the probabilities $p(w_i \mid C_j)$, the classifier is trained. An interesting question is how to define the vocabulary. This can require some experimentation. The simplest solution is to build the vocabulary out of every word encountered in all messages. This is what we did in the notebook *Naïve Bayes - The Bernoulli Document Model*, where we apply this to a set of SMS messages which have been classified 'ham or 'spam' (link is in the notebook). This works well enough but could be refined. For example, the word 'the' is probably not very informative and might be left out.

Once the vocabulary has been defined, the messages can be reduced to feature vectors and for every message you find whether

$$\Pi_{i=1}^{N} P(w_i \mid C_{'spam'})^{b_i} (1 - P(w_i \mid C_{'spam'})^{1-b_i} p_{'spam'} > \Pi_{i=1}^{N} P(w_i \mid C_{'ham'})^{b_i} (1 - P(w_i \mid C_{'ham'})^{1-b_i} p_{'ham'} \tag{13.3}$$

Here i runs over the N vocabulary words. If this inequality holds, the message is labeled 'spam', if not 'ham'.

14 Multilayer-Perceptrons

14.1 Introduction

Here, we will discuss a generalisation of logistic regression: the multilayer-perceptron. Imagine that we would decide to use three perceptrons in parallel in the iris classification problem. We can build this classifier by placing three perceptrons in parallel, each one taking 4 numbers as input, and making a decision whether or not a data point is member of a particular variety. If each perceptron were perfect about its own decision (setosa/non setosa, etc.), the output nodes of the three perceptrons could be grouped in a single three node layer which would form a 'one-hot' representation for the variety prediction, because if each perceptron would do its job perfectly, only one of the nodes would be '1' and the other two would be 'o'. This would produce a two-layer network with four input and three output nodes, shown in Fig. 12.37. The network can be expressed concisely as follows:

$$o_i = f(\sum_{j=1}^{4} w_{ij} x_j), i = 1, 2, 3 \tag{14.1}$$

We immediately see that mathematically the heart of the network calculation is a matrix-vector multiplication. The squashing function f is applied node-wise on the the result of that application.

From Fig. 12.36 it is clear that we would be able to find a perceptron that classifies setosa vs. non-setosa. We can also find a perceptron that will make reasonable decisions on virginica vs. non virginica, but a perceptron that will try to separate versicolor vs non versicolor will not do well. This should be clear, just from looking at the data in Fig. 12.36, and came up in *Activity: Perceptron on Iris dataset*. This group of data points cannot be easily separated from the other two by a single line, or in the full four dimensional input space by a single hyperplane.

One can make a much more reliable network by adding another layer, which uses the output layer of our previous network as an input. This layer can implement the following logic: the output layer of our two layer network can be trusted if it has decided that input

point is setosa or virginica. If neither of the nodes corresponding to these varieties are active, then, by elimination, it is likely to be versicolor. The logic {(**if not** *setosa***if not** *virginica* **then** *versicolor*} can be implemented in a single perceptron.

The new output layer can then consist again of three nodes, two which simply relay the decision of the setosa and the versicolor, and a third one implementing the logic we just outlined.

14.2 *Multilayer Perceptrons*

In the last section we saw that artificial nodes can be grouped into layered networks, and we have provided some heuristic arguments for why it is useful to include a so-called *hidden layer* in between the input and output layers. It can be shown that neural networks with a hidden layer are *universal approximators*: on a compact domain they can approximate almost any function with enough hidden nodes. In practice things are not that simple. A neural network that is used naively can require a large number of weights that can lead to severe overfitting. Important heuristics have been discovered that reduce the number of weights. In *convolutional neural networks*, layers are not fully connected, but relatively small layers called filters are used which have the same weight values across the network, which is achieved through a procedure called weight sharing. Other tricks such as *pooling*, *drop out* and *regularisation* are also important. These heuristics are so extensive that most of the *Deep Learning* module is dedicated to it.

Clearly, designing networks by hand, as we have done in the examples, is not a viable procedure. We want something akin to regression, where we used labelled data and use an iterative algorithm to find weights that minimise a predetermined loss function. To keep matters relatively simple, here we will focus on fully connected networks with one hidden layer. Our primary objective is to introduce the *backpropagation by error*, also backpropagation or backprop for short. The backpropagation algorithm allows us to calculate the gradient of the loss function with respect to the weights for multilayer networks. Even if the architecture of neural networks has changed radically since their introduction, the backpropagation algorithm has remained a mainstay for training them.

Mathematically, the relation between input and output can be described as follows:

$$\boldsymbol{o} = f(\boldsymbol{W}\boldsymbol{h}) = f(\boldsymbol{W}g(\boldsymbol{V}\boldsymbol{i})) \tag{14.2}$$

Here $\boldsymbol{i}$ is an array of I nodes, the values which are set by the user. The is called *entering an input pattern into the network.* There are H

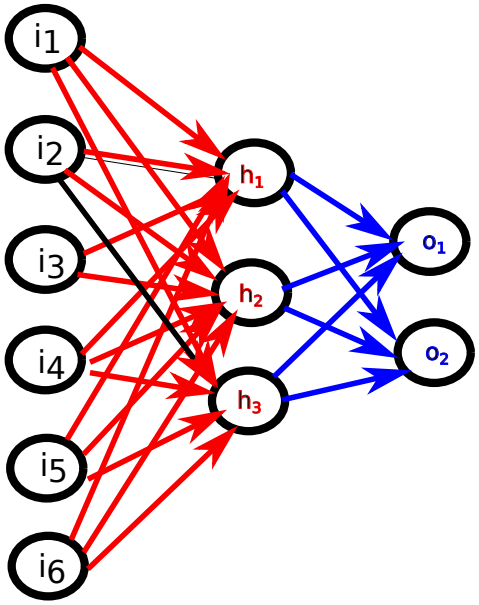


Figure 14.40: A fully connected network. The red connections can be organised in 3×6 matrix $\boldsymbol{V}$, the blue connections in 1 2×3 matrix $\boldsymbol{W}$.

hidden nodes, and O output nodes. $\boldsymbol{V}$ is a $H \times I$ matrix, and $\boldsymbol{W}$ is a $O \times H$ matrix. The products $\boldsymbol{Vi}$ is matrix-vector multiplication, so is of the form: $\sum_{q=1}^{H} V_{pq} i_q$, where V_{pq} are the components of matrix $\boldsymbol{V}$ and i_q is a component of vector $\boldsymbol{I}$. A squashing function $g(x)$ is applied node-wise to the output of the matrix vector multiplication, which defines the values of vector $\boldsymbol{H}$. A second calculation takes place that carries the values of the hidden layer into the output layer. This entails a matrix vector multiplication of the matrix $\boldsymbol{W}$ with the values of $\boldsymbol{H}$ and the squashing function $f(x)$ is applied node-wise on the result of this calculation.

Writing this explicitly in terms of the components makes clear that this is a relatively convoluted expression:

$$o_i = f(\sum_j w_{ij} g(\sum_k v_{jk} i_k)) = f(\sum_j w_{ij} h_j) \tag{14.3}$$

Eq. 14.3 is a very fundamental equation. It describes how multilayer perceptrons operate on an input pattern. Computationally the most expensive part of this are the matrix-vector multiplications as the matrices v, w can be very large. Moreover, we have restricted ourselves for demonstration purposes to a three-layer network. There

is no need for this restriction, and indeed networks can be much deeper, requiring dozens of matrices. One of the technical developments that have made neural networks viable as a computational technique is that matrix-vector multiplications can be highly parallelised. Each node in the output vector can be calculated independent of any other node. Since these operations are very typical in graphics operations, modern computers contain a graphics card with dedicated hardware called a GPU, which is designed to perform a large number of such operations in parallel. It was quickly realised that neural networks can parallelised in a similar way because within each layer computation of node values can be done in parallel. Nowadays, large networks are always trained on a GPU. Modern neural network software makes the use of GPUs completely transparent, as we will see.

As before, we can define a loss function, which could be an MSE loss function, or cross entropy. We will use the MSE loss in the discussion below, but the same ideas apply to cross entropy or other loss functions.

For MSE the loss function is defined as:

$$\mathcal{L} = \sum_i \frac{1}{2}(\boldsymbol{o}_i - \boldsymbol{d}_i)^2 \tag{14.4}$$

We will discuss the backpropagation from a regression point of view, later returning to classification. We assume that for each data point $\boldsymbol{x}_i$, we have a desired regression value $\boldsymbol{d}_i$. Note that $\boldsymbol{x}$ and $\boldsymbol{d}$ are vectors, and that the summation in Eq. 14.4 runs over *data points*, whereas the summations in Eq. 14.3 run over nodes for a single given input vector $\boldsymbol{i}$.

As in logistic regression, we want to calculate the gradient of the loss function with respect to the weights $\boldsymbol{v}, \boldsymbol{w}$, so that we can incorporate it in a stochastic gradient descent algorithm. This might seem like a major problem for the weights $\boldsymbol{v}$, which appear deeply embedded in two functions f and g that both could be non linear. The existence of a relatively simple algorithm to solve this problem was one of the conceptual breakthroughs that made neural networks possible.

14.3 *The Backpropagation Algorithm*

The algorithm that we will discuss in this section is called backpropagation by error. It refers to a method for calculating the gradient of the loss function with respect to the weights, and the reason for this name will become clear below. Its derivation is nothing more than a repeated application of the chain rule of calculus. You should

review this, if you are not confident in its use, e.g.. here: https://tutorial.math.lamar.edu/problems/calci/diffformulas.aspx.

The gradient with respect to the $\boldsymbol{w}$ matrix is similar to the procedure we followed for a single neuron. The main difference here is that we can have more than one output neuron. Our vector of input weights now becomes a matrix, but apart from that nothing changes:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \sum_i (\boldsymbol{o}^{(i)} - \boldsymbol{d}^{(i)}) \frac{\partial \boldsymbol{o}^{(i)}}{\partial \boldsymbol{w}} \tag{14.5}$$

We derive the batch version of the algorithm where we calculate the gradient over all data points, we will discuss variations later. To distinguish between data points indices, which tells us which input output pair from the dataset we are dealing with, and node labels that tells us which node we are discussing, we use brackets in the data point indices.

Next, we will use the fact that derivatives are linear. So we can calculate the gradient for every data point, do this for each data point and then add the result to obtain the gradient of the batch. Below, we will show the gradient calculation for a single data point, say point (1), but since the calculation is identical for all data points, we drop the label for notational convenience. Every index in the calculations below will label a node until further notice.

For a single data point, our loss function is:

$$\mathcal{L} = \frac{1}{2} \sum_k (o_k - d_k)^2,$$

i.e. the sum of the node-wise difference squared divided by 2. The gradient with respect weight w_{pq} is:

$$\frac{\partial \mathcal{L}}{\partial w_{pq}} = \sum_k (o_k - d_k) \frac{\partial o_k}{\partial w_{pq}} \tag{14.6}$$

Since,

$$\frac{\partial o_k}{\partial w_{pq}} = f'(\sum_j w_{kl} h_l) \frac{\partial \sum_l w_{il} h_l}{\partial w_{pq}} = f'(\sum_j w_{kl} h_l) \sum_l \delta_k^p \delta_l^q h_l \tag{14.7}$$

The Kronecker delta is defined by:

$$\delta_j^i = \begin{cases} i \neq j: & 0 \\ i = j: & 1 \end{cases}$$

This is a convenient way of expressing that $\frac{\partial w_{pq}}{\partial w_{kl}}$ in general is zero, unless $p = i$ and $q = j$. The result can be written as:

$$\frac{\partial o_k}{\partial w_{pq}} = f'(\sum_j w_{kj} h_j) \delta_i^p h_q \tag{14.8}$$

Substituting this back into Eq. 14.6 gives:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{pq}} &= (o_p - d_p) f'(\sum_l w_{kl} h_l) h_q \\ &= \Delta_p^W h_q, \end{aligned} \tag{14.9}$$

This is essentially the graded perceptron learning rule that we derived earlier, plus a Δ symbol that expresses that the gradient of node p does not depend on weights that lead to other nodes than node p (remember: in general, there will be other nodes in the output layer, but their weights do not affect the gradient of this node).

When f' is the logistic function, $f'(\sum_j w_{kj} h_j) = o_k(1 - o_k)$, but we will no longer make the assumption that the logistic function is the only squashing function that is generally applied. Once the squashing function has been decided the awkward looking f' terms always reduce to something simple. For now, we will leave them in.

The derivative with respect to the matrix $\boldsymbol{V}$ is a repeated application of the chain rule. We now need to make the dependency explicit:

$$o_k = f(\sum w_{kl} h_l) = f(\sum_l w_{kl} g(\sum_m v_{lm} l_m)).$$

The since the $\boldsymbol{V}$ matrix is embedded inside two function calls, we need to apply the chain rule twice:

$$\frac{\partial o_k}{\partial v_{pq}} = f'(\sum_l w_{kl} h_l) \sum_r w_{kr} g'(\sum_m v_{rm} i_m) \sum_s \frac{\partial}{\partial v_{pq}} v_{rs} i_s,$$

which works out as:

$$\frac{\partial o_k}{\partial v_{pq}} = f'(\sum_l w_{kl} h_l) w_{kp} g'(\sum_m v_{pm} i_m) i_q.$$

This gives:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{pq}} &= \sum_k (o_k - d_k) f'(\sum w_{kl} h_l) w_{kp} g'(\sum_m v_{pm} i_m) \\ &= \sum_k \Delta^w w_{kp} g'(\sum_m v_{pm} i_m) i_q \\ &= \Delta_p^W i_q, \end{aligned} \tag{14.10}$$

Here, we introduced:

$$\Delta^V = \sum_k \Delta_k^W w_{kp} g'(\sum_m v_{pm} i_m) \tag{14.11}$$

Eq. 14.11 is our central result. Observe that the sum runs over the first index, contrary to most formulae we introduced so far.

You will not be assessed on replicating this formula, although it is useful that you try to follow the derivation. Without at least sketching the backpropagation algorithm, it would be hard to understand

how modern machine learning frameworks help you to use it. We will discuss this in detail in Sec. 14.5 and will give examples of its practical use in notebook *Backpropagation in Action*. Before we do that, we will give a heuristic explanation for why a hidden layer provides much of the computational power of neural networks.

14.4 The Role of the Hidden Layer: A Heuristic Explanation

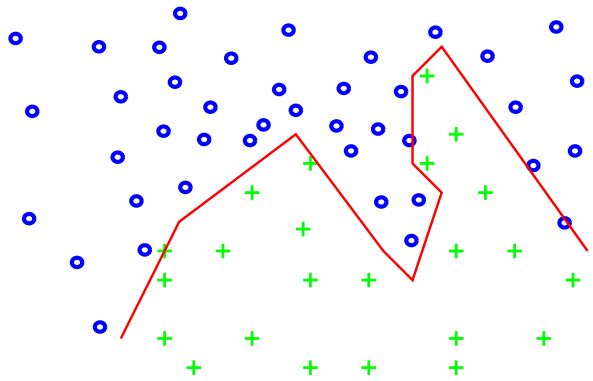


Figure 14.41: A multilayer perceptron can be naively thought of as using higher layers to combine lower level decision boundaries, which are linear, into more complex non-linear ones.

One way of thinking about multilayer perceptrons is to consider the higher layers as organising decisions made on the basis of linear decision layers of individual perceptrons into more complex ones (Fig. 14.41. Adding hidden noes therefore allows more combinations and therefore increases the computational power of the network. An example is given in *Activity Multilayered Perceptron*, where you will explore the increased possibilities of a multilayer neural network. The non-linearity of these hidden nodes is essential. Consider Eq. 14.3 when $g(x) = x$. It reads:

$$o_i = f(\sum_j w_{ij} \sum_k v_{jk} i_k)),$$

but $\sum_j w_{ij} v_{jk}$ is just another matrix u_{ik}, so:

$$o_i = f(\sum_k u_{ik} i_k),$$

but this describes just a two-layer network. If the hidden nodes are linear they just carry out a matrix-vector multiplication and we already have seen that a two-layer neural network is simply a group of perceptrons that has been switched in parallel with all the weaknesses of a single perceptron.

What is true for classification networks is also true for regression networks.

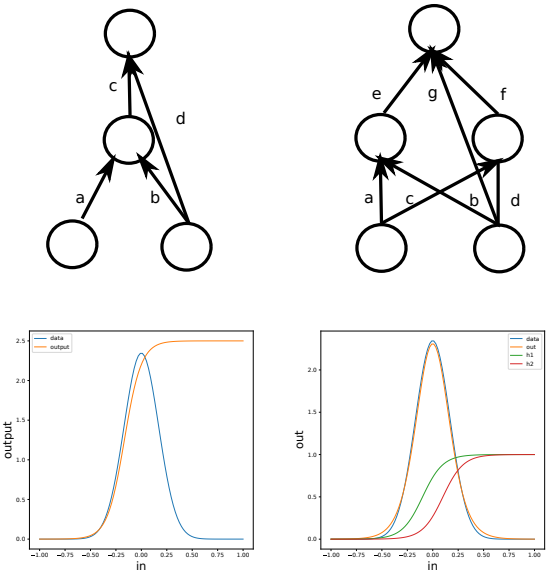


Figure 14.42: Two of the simplest neural network architectures conceivable. The one with one node is only able to capture one trend in the data. At least two nodes are necessary to capture data with a peak. Observe that the two hidden nodes respond in a similar way, but their responses combined captures the data quite well.

14.5 *Interpretation of the Backpropagation Algorithm*

Equation 14.3 has a clear network interpretation: information is entered in the input layer. The hidden layer is then evaluated, essentially by a matrix vector multiplication, and then the output layer, again by a matrix vector multiplication. This can easily be pictured as information flowing through the network.

In a similar way, Eq. 14.11 can be interpreted as a measure of error that is transported back through the network. To see this, consider the quantity Δ^w in Eq. 14.10. It would be 0 if $d_k = o_k$, and is only non-zero when the desired output of the network and the actual output differ.

Equation 14.11 can be interpreted as the error being entered in the output nodes, and propagated back to the hidden layer. To see this have a look at the fully connected network that we reproduce here, together with a reverse network.

In a forward pass we first update the hidden layer by:

$$h_i = \sum_{j=1}^{6} v_{ij} i_j,$$

and then update the output layer:

$$o_k = \sum_{l=1}^{3} w_{kl} h_l$$

We use the convention here that in v_{ij}, the index i labels the *row* of the matrix, and j the column.

Now consider the reverse network, which has its input layer on the right. If we now where to update the hidden layer, we would calculate:

$$h_i = \sum_{j=1}^{2} w_{ji}$$

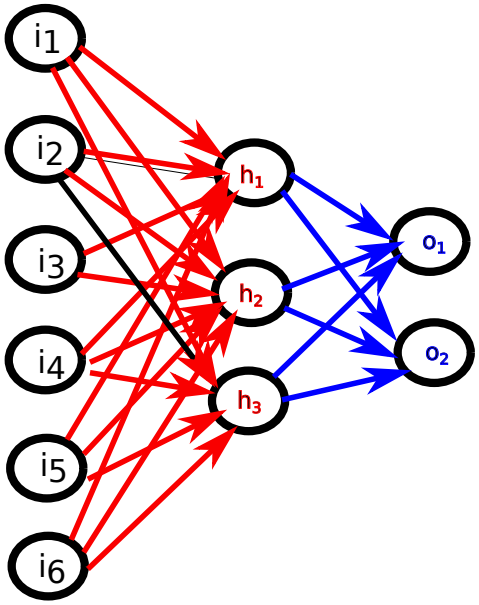
The summation here runs over the first index! Now consider Eq. 14.11. It has exactly this form. At the output nodes we calculate Δ^W, which we feedback in the network. Then we do something at the nodes. Note that we only have to propagate ΔW back to the hidden layer. The gradient for the connection v_{pq} is the product $i_p \Delta_q^V$.

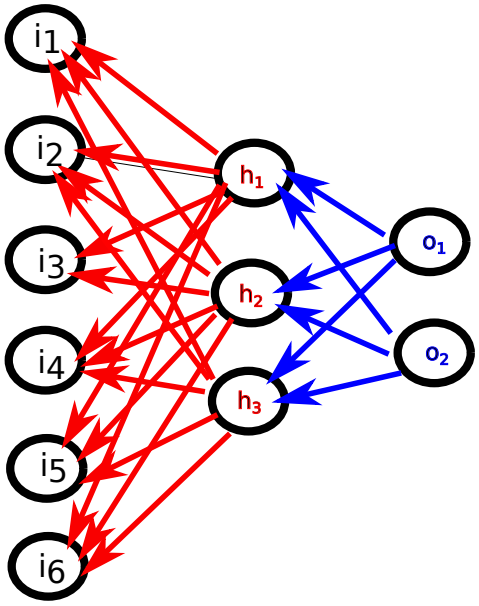
The name backpropagation is therefore warranted. Interestingly, the algorithm works similar for networks with more than one hidden layer: the error will be propagated back until the first hidden layer.

As you can see, the backpropagation algorithm is relatively complex and needs to be adapted to the architecture of the network, the loss function and the squashing functions used in the network. The development of new neural networks always used to require considerable amount of software development. Modern neural network frameworks such as *TensorFlow*, *Keras*, *PyTorch* or *Theano*, have a so-called *autograd* facility. Networks can be created in terms of symbolic code, and learning algorithms can be automatically derived when loss function, architecture or other aspects of the network are changed. It is still important to understand the backpropagation algorithm, but it is no longer necessary to implement this yourself. In *Activity: Autograd* you will experiment with the *autograd* functionality of *PyTorch* that can perform the calculation of gradients automatically.

In *Activity Multilayered Perceptron* you will be guided into building a neural network that can classify elements of the MNIST dataset using a multilayer perceptron.

Figure 14.43: The update rule for the forward network is $o = f(\boldsymbol{W}g(\boldsymbol{V}\boldsymbol{i}))$. The update rule for the reverse network is $i = g(\boldsymbol{V}^T f(\boldsymbol{W}^T \boldsymbol{o}))$.





Bibliography

W. C. Abraham, O. D. Jones, and D. L. Glanzman (2019). Is plasticity of synapses the mechanism of long-term memory storage? *NPJ science of learning*, 4(1):pp. 1–10.

C. M. Bishop (2006). *Pattern recognition and machine learning*. springer.

C. M. Bishop *et al.* (1995). *Neural networks for pattern recognition*. Oxford university press.

T. M. Cover (1999). *Elements of information theory*. John Wiley & Sons.

C. Gardiner (2009). *Stochastic methods*, volume 4. Springer Berlin.

I. Goodfellow, Y. Bengio, and A. Courville (2016). *Deep learning*. MIT press.

A. L. Hodgkin and A. F. Huxley (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):pp. 500–544.

E. T. Jaynes (2003). *Probability theory: The logic of science*. Cambridge University Press.

L. V. Jospin, W. Buntine, F. Boussaid, H. Laga, and M. Bennamoun (2020). Hands-on bayesian neural networks–a tutorial for deep learning users. *arXiv preprint arXiv:2007.06823*.

M. Minsky and S. A. Papert (2017). *Perceptrons: An introduction to computational geometry*. MIT press.

K. P. Murphy (2012). *Machine learning: a probabilistic perspective*. MIT press.

J. Pearl, M. Glymour, and N. P. Jewell (2016). *Causal inference in statistics: A primer*. John Wiley & Sons.

J. Pearl and D. Mackenzie (2018). *The book of why: the new science of cause and effect*. Basic books.

T. J. Sejnowski (2020). The unreasonable effectiveness of deep learning in artificial intelligence. *Proceedings of the National Academy of Sciences*, 117(48):pp. 30033–30038.

D. of Trade and Industry (1998). *ADULTDATA - The handbook of adult anthropometric and strength measurements.*