MARC DE KAMPS

MACHINE LEARNING - UNIT 1 (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.

We will investigate the multivariate Gaussian distribution. You should be able to:

- Provide maximum likelihood estimates for mean and covariance matrix.
- Apply the spectral decomposition theorem in creating synthetic data sets.

Then, we will revisit linear regression. You should be able to

- Provide a Bayesian interpretation of linear regression.

Finally, you will learn an important technique for comparing probability distributions. You should be able to:

- Give an interpretation of entropy in terms of coding length.
- Recognise the KL-divergence as a techniques for comparing probability distributions.

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).

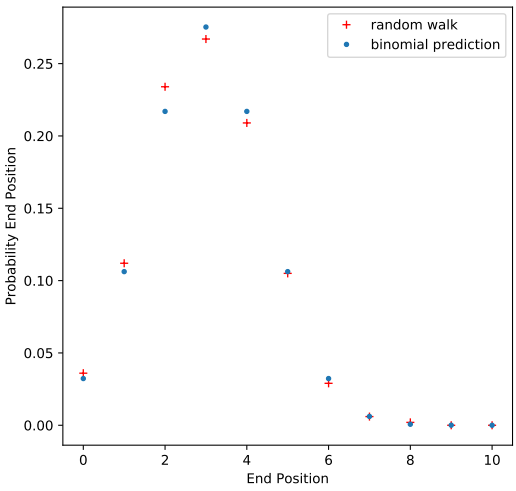


Figure 2.1: Example of a random walk as a binomial process. In total 10 steps are considered. For each step, there is a probability μ to move to the right and a probability 1 - μ to stay in place. This 10 step process is simulated with a Bernoulli process and the end position is recorded 1000 times. 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*

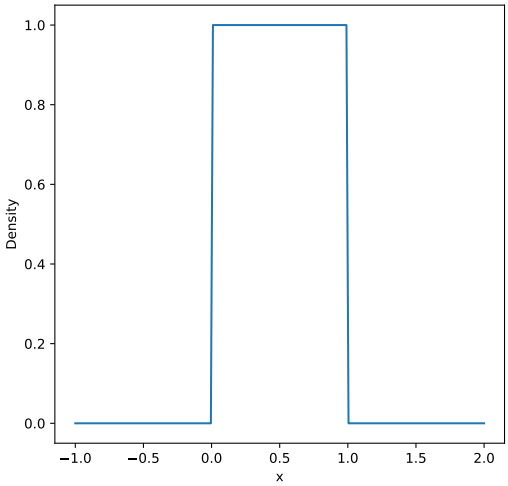


Figure 2.2: Top: the uniform distribution.

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*

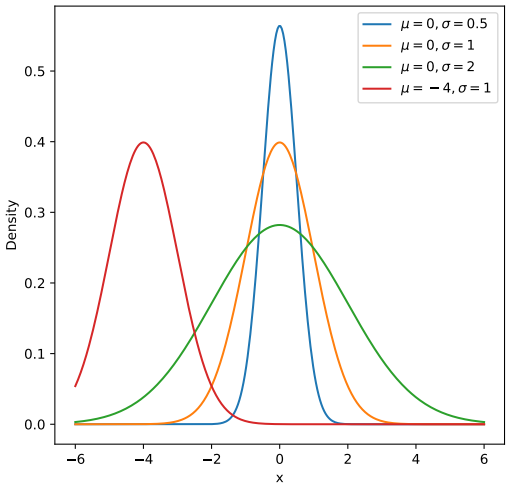


Figure 2.3: The Gaussian or normal distribution for several values of μ and σ.

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; *idd*).

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 minimum of the log likelihood, moreover it is the only minimum. 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

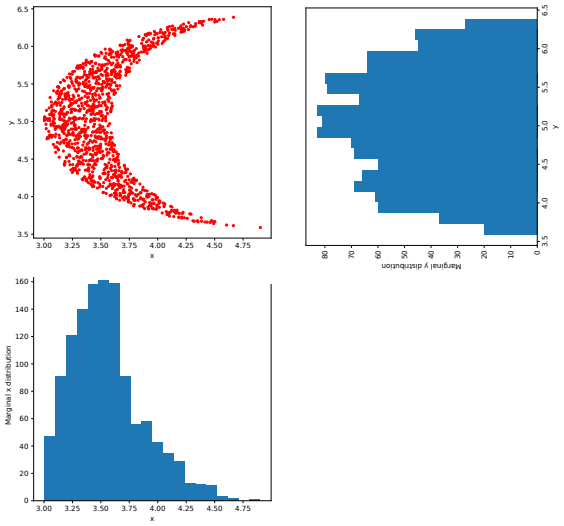


Figure 3.4: Joint probability distribution function in two variables x and y. The marginal distributions are shown as a histogram in the margins of the joint distribution.

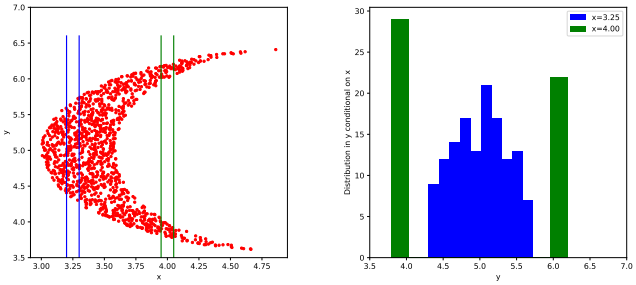


Figure 3.5: The same joint probability distribution conditioned on two different values of x. Each value of x results in a one dimensional conditional distribution which can be unimodal or bimodal.

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: $N_0 = 23, N_1 = 67, N = 100$

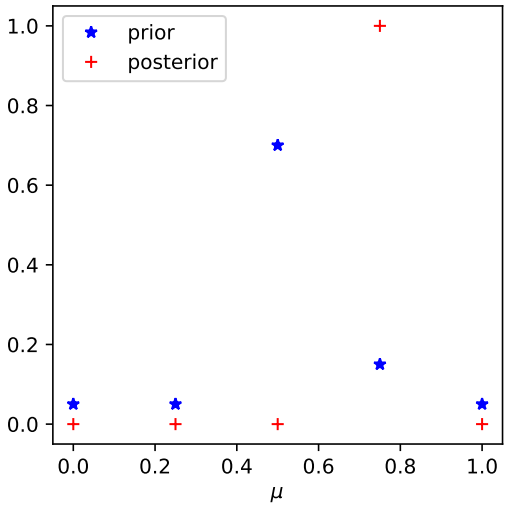


Figure 3.6: The prior and posterior over a discrete set of μ values. The prior distribution was by our own choosing, reflecting the fact that we believed that the coin is fair, but allowing for uncertainty. The posterior distribution results from a consequent application of Bayes rule. It has shifted 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:

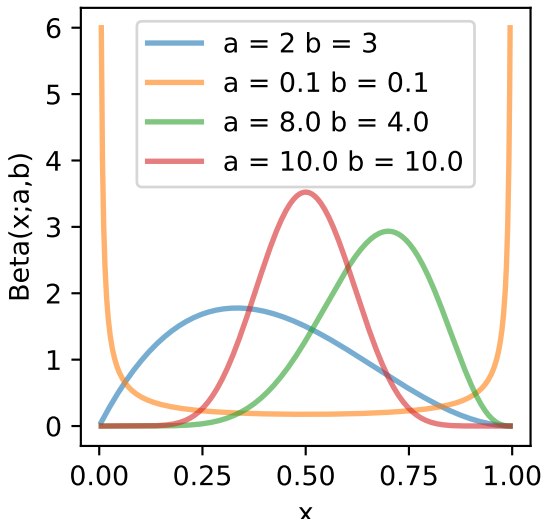


Figure 3.7: The Beta distribution for several values of a and b.

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

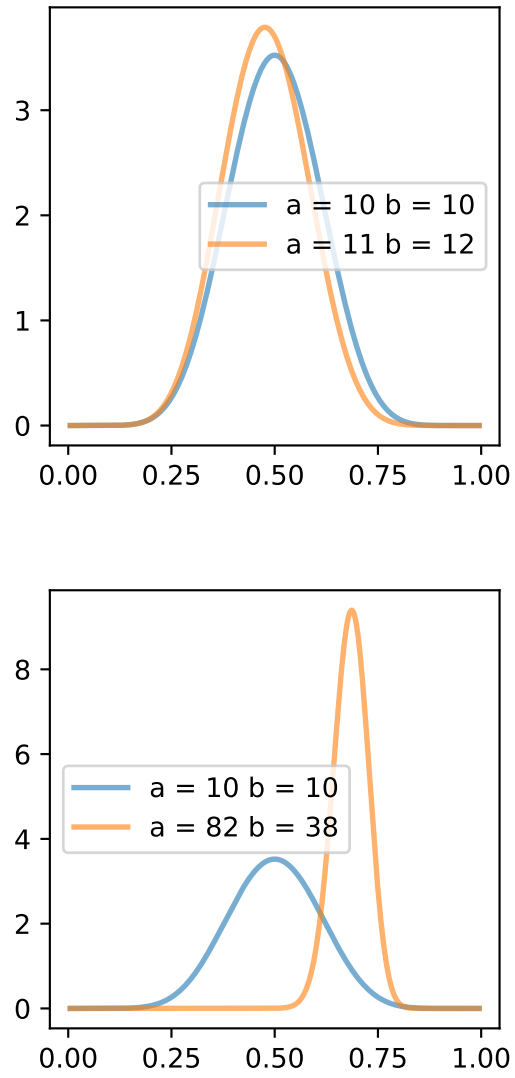


Figure 3.8: Prior and posterior distributions after three observations of a heavily loaded coin $\mu = 0.75$, starting from a relatively broad prior centred around $\mu = 0.5$. Top: $N = 3$ observations, bottom $N = 100$ observations.

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

Bibliography

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

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

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

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

MARC DE KAMPS

MACHINE LEARNING - UNIT 1 (v1.0)

UNIVERSITY OF LEEDS

Contents

List of Figures

List of Tables

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

1.1 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:

$$\ln(\mathcal{D} \mid \mu, \sigma^2) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i - \mu)^2 - \frac{N}{2}\ln\sigma^2 - \frac{N}{2}\ln 2\pi \tag{1.1}$$

Maximising Eq 1.1 with respect to μ gives:

$$\mu_{ML} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{1.2}$$

Maximising with respect to σ gives

$$\sigma^2_{ML} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_{ML})^2 \tag{1.3}$$

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

$$\mathbb{E}[\mu_{ML}] = \mu \tag{1.4}$$
$$\mathbb{E}[\sigma^2_{ML}] = (\frac{N-1}{N})\sigma^2$$

1.1.1 *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]:

$$p(S_n = s) = \frac{1}{\sqrt{2\pi N\sigma^2}} e^{-\frac{(s-N\mu)^2}{2N\sigma^2}}, \tag{1.5}$$

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

1.1.2 *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:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt, \tag{1.6}$$

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

$$\Phi(x; \mu, \sigma) = \frac{1}{2} \left(1 + \text{erf} \left(\frac{x - \mu}{\sigma \sqrt{2}} \right) \right) \tag{1.7}$$

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

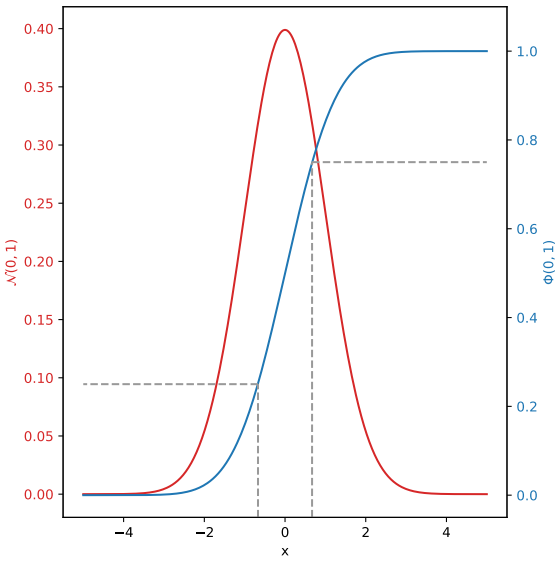


Figure 1.1: Standard Gaussian distribution (red). Its cumulative distribution function (blue). First and last quartile indicated by grey dashed lines.

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 1.1.

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

2 Multivariate Gaussian Distributions

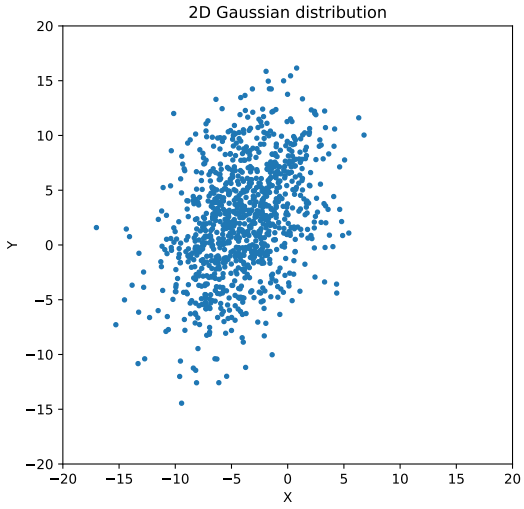


Figure 2.2: Sample of a two dimensional Gaussian distribution in two variable: X and Y. The variables are correlated and not centred at the origin.

2.1 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. 2.2. 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:

$$\mathbf{x}_i = \begin{pmatrix} \text{iq}_i \\ \text{height}_i \end{pmatrix} \tag{2.1}$$

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:

$$\mathbf{x}^T = (\text{iq}, \text{height}) \tag{2.2}$$

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:

$$\mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{\mid \det(\boldsymbol{\Sigma}) \mid^{\frac{1}{2}}} \exp\left\{ -\frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}) \right\} \tag{2.3}$$

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

$$\Delta^2 = (\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}) \tag{2.4}$$

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.

2.1.1 *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.

$$\boldsymbol{u}_i^T \boldsymbol{u}_j = \mathbb{1}, \tag{2.5}$$

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:

$$p(\boldsymbol{y}) = \Pi_{j=1}^{D} \frac{1}{(2\pi\lambda_j)^{1/2}} \exp\left\{-\frac{y_j^2}{2\lambda_j}\right\} \tag{2.6}$$

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:

$$\boldsymbol{\Sigma} = \sum_{i=1}^{D} \lambda_i \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{2.7}$$

For the inverse:

$$\boldsymbol{\Sigma}^{-1} = \sum_{i=1}^{D} \frac{1}{\lambda_i} \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} \tag{2.8}$$

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

2.1.2 *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. 2.21 and 2.26.

[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:

$$\ln p(\boldsymbol{X} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{ND}{2}\ln(2\pi) - \frac{N}{2}\ln\det(\boldsymbol{\Sigma}) - \frac{1}{2}\sum_{n=1}^{N}(\boldsymbol{x}_n - \boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_n - \boldsymbol{\mu}) \tag{2.9}$$

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

$$\mathrm{Tr}\boldsymbol{A} = \sum_{i=1}^{D} A_{ii} \tag{2.10}$$

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

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T, \tag{2.11}$$

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

$$\det(\boldsymbol{A}) = \Pi_{i=1}^{D}\lambda_i, \tag{2.12}$$

and

$$\mathrm{Tr}(\boldsymbol{A}) = \sum_{i=1}^{D}\lambda_i \tag{2.13}$$

Both these results depend on Eq. 2.11 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:

$$\frac{\partial}{\partial \boldsymbol{a}}\boldsymbol{b}^T\boldsymbol{a} = \boldsymbol{b} \tag{2.14}$$

$$\frac{\partial}{\partial \boldsymbol{a}}(\boldsymbol{a}^T\boldsymbol{A}\boldsymbol{a}) = (\boldsymbol{A} + \boldsymbol{A}^T)\boldsymbol{a} \tag{2.15}$$

$$\frac{\partial}{\partial \boldsymbol{A}}\mathrm{Tr}(\boldsymbol{B}\boldsymbol{A}) = \boldsymbol{B}^T \tag{2.16}$$

$$\tag{2.17}$$

also easy to verify is:

$$\mathrm{Tr}(\boldsymbol{A}\boldsymbol{B}\boldsymbol{C}) = \mathrm{Tr}(\boldsymbol{C}\boldsymbol{A}\boldsymbol{B}) = \mathrm{Tr}(\boldsymbol{B}\boldsymbol{C}\boldsymbol{A}) \tag{2.18}$$

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

$$\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} = \mathrm{Tr}(\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}) = \mathrm{Tr}(\boldsymbol{x}\boldsymbol{x}^T\boldsymbol{A}) = \mathrm{Tr}(\boldsymbol{A}\boldsymbol{x}\boldsymbol{x}^T) \tag{2.19}$$

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. 2.7, 2.8, and Eq. 2.12 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:

$$\frac{\partial}{\partial \boldsymbol{A}} \ln \det(\boldsymbol{A}) = (\boldsymbol{A}^{-1})^T = \boldsymbol{A}^{-T}, \tag{2.20}$$

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. 2.9 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 2.15 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:

$$\boldsymbol{\mu}_{ML} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i \tag{2.21}$$

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:

$$\boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} \tag{2.22}$$

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:

$$\mathcal{L}(\mathbf{\Lambda}) = \frac{N}{2}\ln\det(\mathbf{\Lambda}) - \frac{1}{2}\sum_{i=1}^{N}(\boldsymbol{x}_i - \mu)^T\mathbf{\Lambda}(\boldsymbol{x}_i - \boldsymbol{\mu}) \tag{2.23}$$

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:

$$\begin{aligned}\mathcal{L}(\mathbf{\Lambda}) &= \frac{N}{2}\ln\det(\mathbf{\Lambda}) - \frac{1}{2}\sum_{i=1}^{N}\mathrm{Tr}(\boldsymbol{x}_i - \boldsymbol{\mu})^T\mathbf{\Lambda}(\boldsymbol{x}_i - \boldsymbol{\mu}),\\ &= \frac{N}{2}\ln\det(\mathbf{\Lambda}) - \frac{1}{2}\mathrm{Tr}(\boldsymbol{S}_\mu\mathbf{\Lambda}),\end{aligned} \tag{2.24}$$

where the scatter matrix

$$\boldsymbol{S}_\mu \equiv \sum_{i=1}^{N}(\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^T \tag{2.25}$$

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:

$$\mathbf{\Sigma}_{ML} = \frac{1}{N}\boldsymbol{S}_\mu = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T \tag{2.26}$$

2.2 *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:

$$-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu}) = -\frac{1}{2}\boldsymbol{x}^T\mathbf{\Sigma}^{-1} + \boldsymbol{x}^T\mathbf{\Sigma}^{-1}\boldsymbol{\mu} + \text{const} \tag{2.27}$$

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.

3 Bayesian Linear Regression

3.1 Introduction

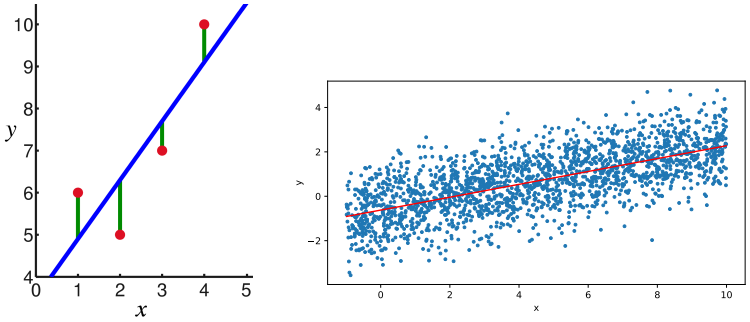


Figure 3.3: Left: observed points in red. A line fit to these points in blue. The residues in green (Wikimedia-ccby). Right: An example of linear regression.

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. 3.3, 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. 3.3 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. 3.3, 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:

$$\begin{aligned} \sum_i x_i y_i &= a \sum_i x_i^2 + b \sum_i x_i \\ \sum_i y_i &= a \sum_i x_i + bN \end{aligned} \tag{3.1}$$

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:

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_i x_i^2 & \sum_i x_i \\ \sum_i x_i & N \end{pmatrix}^{-1} \begin{pmatrix} \sum_i x_i y_i \\ \sum_i y_i \end{pmatrix} \tag{3.2}$$

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:

$$\phi(x_i) = \begin{pmatrix} 1 \\ x_i \\ x_i^2 \end{pmatrix} \tag{3.3}$$

and a vector $\boldsymbol{w}$:

$$\boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}, \tag{3.4}$$

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:

$$R = \sum_{i=1}^{N} (y_i - \boldsymbol{w}^T \boldsymbol{\phi}(x_i))^2 \tag{3.5}$$

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

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

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

$$\sum_{i=1}^{N} y_i \boldsymbol{\phi}^T(x_i) = \boldsymbol{w}^T (\sum_{i=1}^{N} \boldsymbol{\phi}(x_i) \boldsymbol{\phi}^T(x_i)) \tag{3.6}$$

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, $M = 3$.

$$\Phi = \begin{pmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_{M-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_{M-1}(x_2) \\ \vdots & \vdots & & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_{M-1}(x_N) \end{pmatrix} \tag{3.7}$$

Using this matrix, we can write Eq. 3.6 as

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

so that

$$w_{OLE} = (\mathbf{\Phi}^T\mathbf{\Phi})^{-1}\mathbf{\Phi}^T\boldsymbol{t} \tag{3.8}$$

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. 3.8 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:

$$\Phi\Phi^T = \begin{pmatrix} N & \sum_i x_i \\ \sum_i x_i & \sum_i x_i^2 \end{pmatrix}, \tag{3.9}$$

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:

$$\begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} N & \sum_i x_i \\ \sum_i x_i & \sum_i x_i^2 \end{pmatrix}^{-1} \begin{pmatrix} \sum_i y_i \\ \sum_i x_i y_i \end{pmatrix} \tag{3.10}$$

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. 3.8 works out in practice.

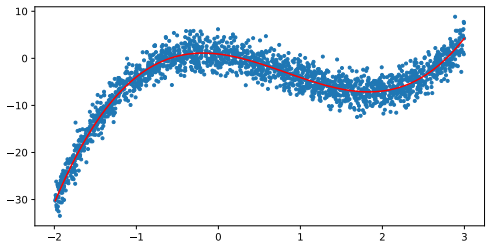


Figure 3.4: The result of linear regression on a cubic polynomial.

Figure 3.4 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. 3.8 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.

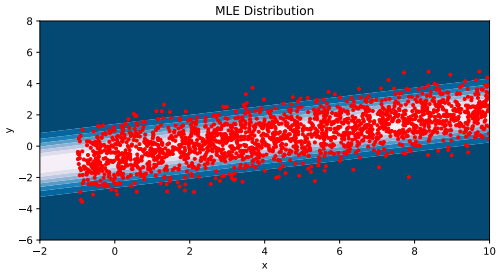


Figure 3.5: The MLE, a single point in (μ, σ) space, determines a Gaussian distribution centred around the line represented by the MLE.

3.2 *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

$$t = y(\boldsymbol{x}, \boldsymbol{w}) + \epsilon, \tag{3.11}$$

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

$$p(t \mid \boldsymbol{x}, \boldsymbol{w}, \beta) = \mathcal{N}(t \mid y(\boldsymbol{x}, \boldsymbol{w}), \beta^{-1}) \tag{3.12}$$

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 3.12, it is possible to define a likelihood function with adjustable parameters $\boldsymbol{w}$ and β

$$p(\boldsymbol{t} \mid \boldsymbol{X}, \boldsymbol{w}, \beta) = \Pi_{i=1}^{N} \mathcal{N}(t_i \mid \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i), \beta^{-1}), \tag{3.13}$$

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

$$\begin{aligned} \ln p(\boldsymbol{t} \mid \boldsymbol{w}, \beta) &= \sum_{i=1}^{N} \ln \mathcal{N}(t_i \mid \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i), \beta^{-1}) \\ &= \frac{N}{2} \ln \beta - \frac{N}{2} \ln(2\pi) - \beta E_D(\boldsymbol{w}) \end{aligned} \tag{3.14}$$

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

$$E_D(\boldsymbol{w}) = \frac{1}{2} \sum_{i=1}^{N} (t_i - \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}_i))^2 \tag{3.15}$$

But this is the same form as the sum of squared residues Eq. 3.5! 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:

$$\boldsymbol{w}_{MLE} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{t} \tag{3.16}$$

It is important to note that no particular noise model underlies the OLE estimates, whereas the interpretation of $\boldsymbol{w}_{MLE}$ as maximum likelihood estimate explicitly determines on the Gaussian assumption made in Eq. 3.12. However, when the Gaussian model is appropriate it allows a more extensive analysis for example Bayesian linear regression.

3.3 *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 $\boldsymbol{w}$:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{m}_0, \boldsymbol{S}_0) \tag{3.17}$$

The likelihood function is given by:

$$\mathcal{L} = \Pi_{i=1}^{N} \mathcal{N}(t_i \mid \boldsymbol{w}^T \boldsymbol{\phi}(x_i), \beta^{-1}), \tag{3.18}$$

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:

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{t}, \alpha, \beta) \sim p(\boldsymbol{t} \mid \boldsymbol{X}\boldsymbol{w}\beta)p(\boldsymbol{w} \mid \alpha) \tag{3.19}$$

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 3.13, 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:

$$\boldsymbol{m}_N = \boldsymbol{S}_N(\boldsymbol{S}_0^{-1}\boldsymbol{m}_0 + \beta\boldsymbol{\Phi}^T\boldsymbol{t}) \tag{3.20}$$

$$\boldsymbol{S}_N = \boldsymbol{S}_0^{-1} + \beta\boldsymbol{\Phi}^T\boldsymbol{\Phi} \tag{3.21}$$

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:

$$\boldsymbol{m}_N = \beta \boldsymbol{S}_N \boldsymbol{\Phi}^T \boldsymbol{t} \tag{3.22}$$

$$\boldsymbol{S}_N = \alpha \boldsymbol{I} + \beta \boldsymbol{\Phi}^T \boldsymbol{\Phi} \tag{3.23}$$

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:

$$\boldsymbol{m}_N = \boldsymbol{S}_N(\boldsymbol{S}_{N-1}^{-1}\boldsymbol{m}_{N-1} + \beta \boldsymbol{\Phi}^T \boldsymbol{t}) \tag{3.24}$$

$$\boldsymbol{S}_N = \boldsymbol{S}_{N-1}^{-1} + \beta \boldsymbol{\Phi}^T \boldsymbol{\Phi} \tag{3.25}$$

3.3.1 *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.

3.3.2 *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. 3.6 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*.

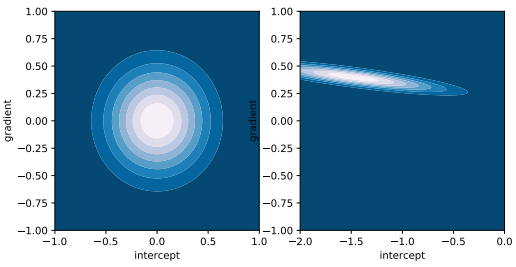


Figure 3.6: Prior and posterior weight distribution after applying Bayesian regression on ten points.

3.4 *Pros and cons of Bayesian Regression*

3.4.1 *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

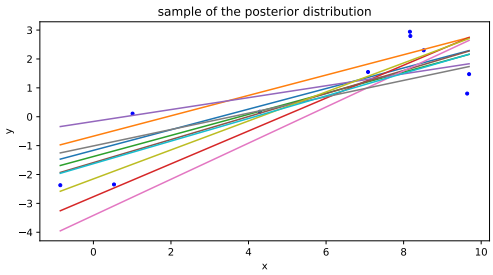


Figure 3.7: This plot shows 10 linear relationships that have been sampled from the posterior distribution, which was obtained by regressing on 10 data points.

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.

3.4.2 *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*

4 Information Theory and Probability

4.1 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:

$$\mathbb{H}[x] = -\sum p(x)^2 \log p(x) \tag{4.1}$$

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 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 $\boldsymbol{x}$.

4.2 *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:

$$\begin{aligned} \mathrm{KL}(p \,||\, q) &= -\int p(x) \ln q(x) df - (-\int p(x) \ln p(x) dx) \\ &= -\int p(x) \ln \{\frac{q(x)}{p(x)} dx \end{aligned} \tag{4.2}$$

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

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

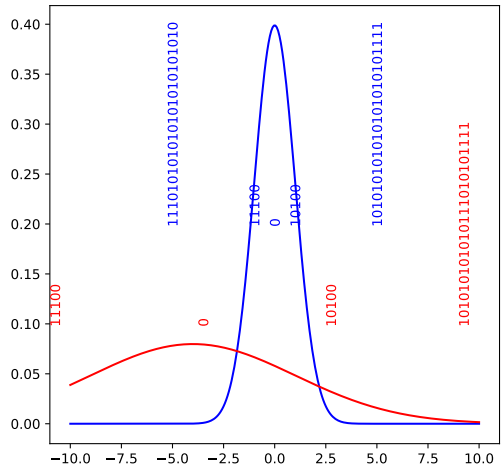


Figure 4.8: Events can be coded by a code book tailored to the red distribution or the blue distribution. When events are distributed according to the red distribution, they are more efficiently coded by the 'red' code book. Most events of the red distribution would fall well outside regular events according 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.

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

Consider Fig. 4.8. 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.

4.3 *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.

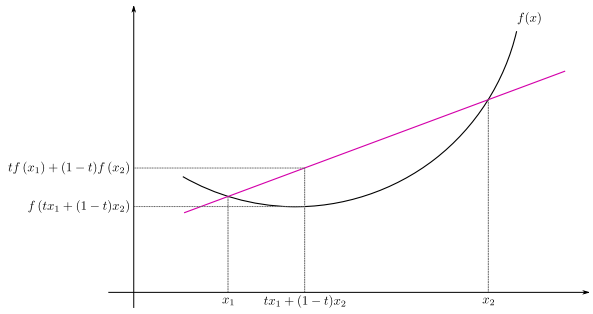


Figure 4.9: Jensen's inequality demonstrated. Source: Wikimedia.

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. 4.9. 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:

$$f(\sum_{i=1}^{M} \lambda_i x_i) \leq \sum_{i=1}^{M} \lambda_i f(x_i), \tag{4.3}$$

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

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:

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

For continuous variables, this becomes:

$$f(\int \boldsymbol{x} p(\boldsymbol{x}) dx) \leq \int f(\boldsymbol{x}) p(\boldsymbol{x}) d\boldsymbol{x} \tag{4.4}$$

Now we can apply this inequality to the KL-divergence:

$$\mathrm{KL}(p \mid\mid q) = -\int p(\boldsymbol{x}) \ln \left\{ \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} \right\} d\boldsymbol{x} \geq -\ln \int q(\boldsymbol{x}) d\boldsymbol{x} \tag{4.5}$$

Here we used that $intq(\boldsymbol{x})d\boldsymbol{x} = 1$, 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. 4.5 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.

5 *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

Bibliography

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

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

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

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*.

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

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

MARC DE KAMPS

MACHINE LEARNING - UNIT 3 (PART 1) (v1.0)

UNIVERSITY OF LEEDS

Contents

List of Figures

List of Tables

1 Introduction

1.1 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.

1.1.1 *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.

1.1.2 *Learning Outcomes*

2 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. 2.1).

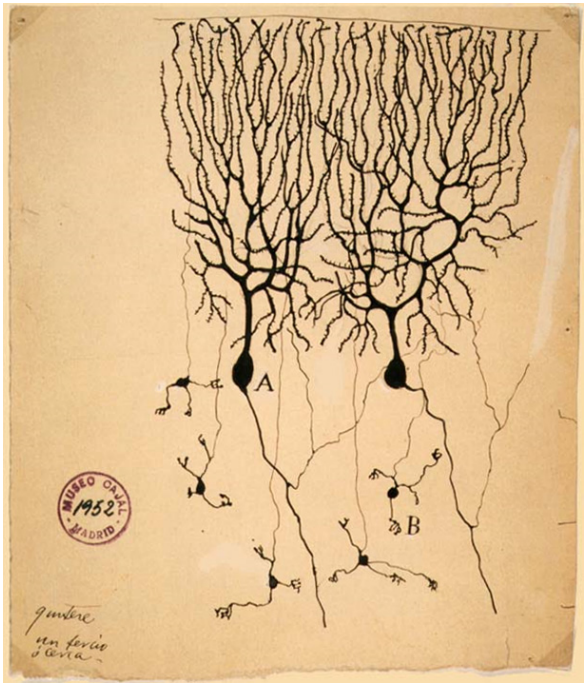


Figure 2.1: Drawing of a Purkinje cell (a type of neuron by Ramon-y-Cajal.

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 specialized 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.

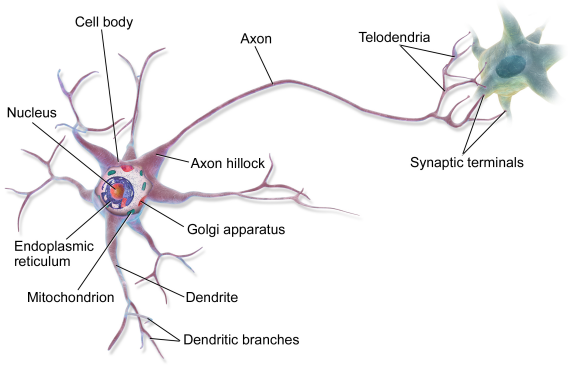


Figure 2.2: A schematic representation of a biological neuron.

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.

3 Perceptron and Linear Discriminants

3.0.1 The Perceptron

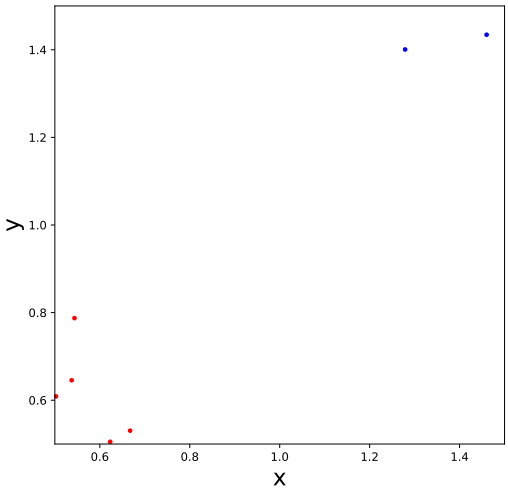


Figure 3.3: A dataset, consisting of measurement values which have been classified as belonging two one of two classes.

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. 3.3. 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:

$$o = \mathcal{H}(ax + by - c), \tag{3.1}$$

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. 3.3. 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:

$$o = f(w_1 x_1 + w_2 x_2 - \theta) \tag{3.2}$$

This is nothing but a rewrite of Eq. 3.1 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. 3.2 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 3.1: 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. 3.4, is similar to Fig. 3.3 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:

$$o = \mathcal{H}(x_1 + x_2 - 1.5) \tag{3.3}$$

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. 3.2 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?

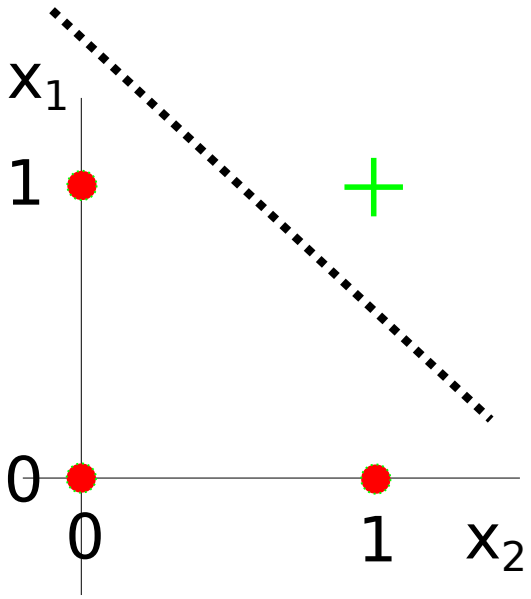


Figure 3.4: The AND gate as a classification problem is linearly separable.

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 3.2: 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 3.3.

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.

3.1 *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.

3.2 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.

3.3 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:

$$o = f(\boldsymbol{w}^T \boldsymbol{x}). \tag{3.4}$$

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:

$$\mathcal{E} = \frac{1}{2} \sum_{i=1}^{D} (o_i - d_i)^2, \tag{3.5}$$

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.

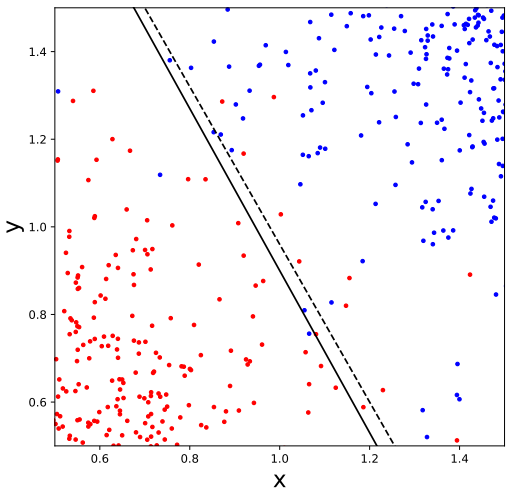


Figure 3.5: Continuous changes for the parameters $\boldsymbol{w}$ is equivalent to rotating and moving the line in a smooth manner. Whenever the decision boundary is crossed, however, $\mathcal{E}$ will jump discontinuously. The decision line for the solid line is: 1.85x + y = 2.75, for the dashed line 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*.

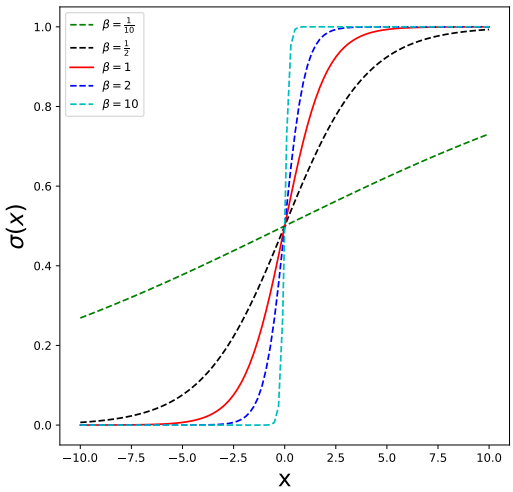


Figure 3.6: The logistic function for various values of the noise parameter β. Observe that for high values of β the function resembles a step function.

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:

$$\sigma(x) = \frac{1}{1 + e^{-\beta x}} \tag{3.6}$$

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. 3.6).

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 3.6 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:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.7}$$

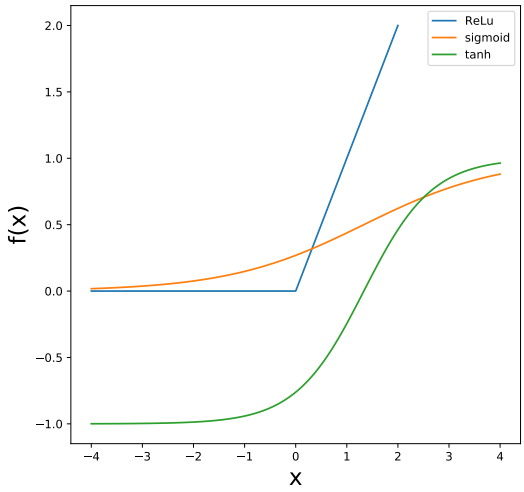


Figure 3.7: Examples of squashing functions that are commonly used in neural networks. They all contain a non-linearity, which is essential in multilayer perceptrons.

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. 3.7.

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:

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} = \sum_i (o_i - d_i) \frac{\partial}{\partial \boldsymbol{w}} o_i. \tag{3.8}$$

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. 3.8 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

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{3.9}$$

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*.

3.4 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:

$$\boldsymbol{w} \to \boldsymbol{w} - \lambda \frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} \tag{3.10}$$

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. 3.10 will lead to a set of weights for which the loss function is locally minimal. Eq. 3.10 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:

$$\boldsymbol{w} \to \boldsymbol{w} - \lambda \sum_i o_i(1 - o_i)(d_i - o_i)\boldsymbol{x}_i \tag{3.11}$$

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. 3.11 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. 3.11 is not very different from the perceptron algorithm with a learning rate.

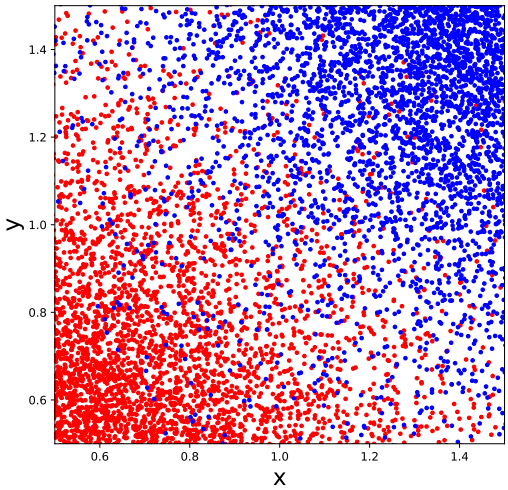


Figure 3.8: A data classification problem where a linear classifier is a reasonable solution, although technically the dataset is not linearly separable.

.

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. 3.8. 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.

3.4.1 *Batches, Minibatches and Momentum*

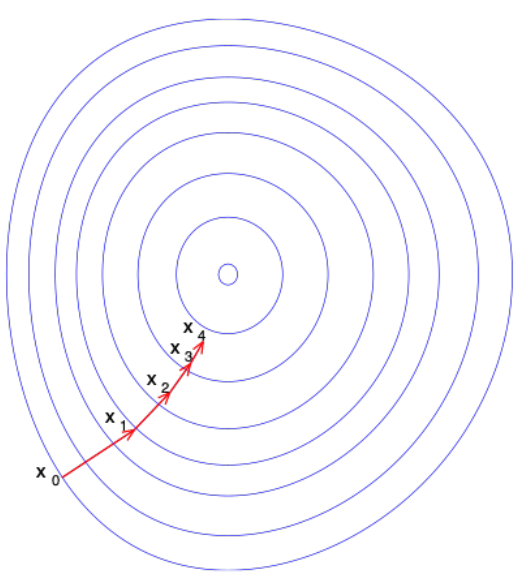


Figure 3.9: Steepest gradient descent in a favourable loss landscape. Source: Wikipedia

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. 3.8. 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. 3.9.

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.

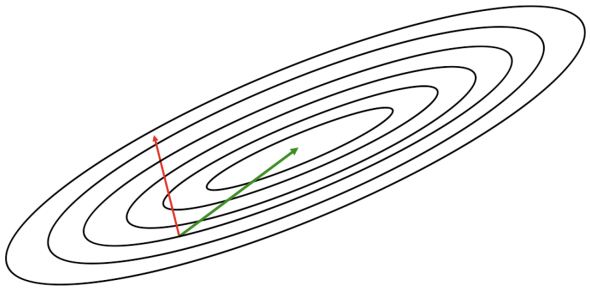


Figure 3.10: An elongated minimum may lead to slow convergence of steepest gradient descent, because the gradient does not really point to the minimum (left).

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. 3.9. 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. 3.11, where due to zigzagging the algorithm almost follows a contour in the landscape, which means it is converging extremely slowly.

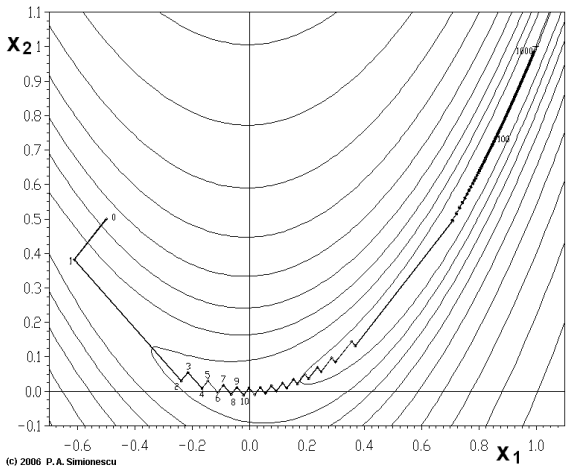


Figure 3.11: Very slow convergence of steepest gradient descent, due to zigzagging (source Wikipedia).

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:

$$\Delta \boldsymbol{w}_t = -\lambda \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} + \alpha \Delta \boldsymbol{w}_{t-1} \tag{3.12}$$

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

4 Logistic Regression

4.1 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* [1]. 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.

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

Bayes' law gives an answer, since:

$$p(\mathcal{C}_1 \mid x) = \frac{p(x \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(x \mid \mathcal{C}_1)p(\mathcal{C}_1) + p(x \mid \mathcal{C}_2)p(\mathcal{C}_2)} \tag{4.1}$$

This can be rearranged to:

$$p(\mathcal{C}_1 \mid x) = \frac{1}{1 + \exp(-a)}, \tag{4.2}$$

with

$$a = \ln \frac{p(x \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(x \mid \mathcal{C}_2)p(\mathcal{C}_2)} \tag{4.3}$$

In Eq. 4.2, we see that the sigmoid function emerges quite naturally.

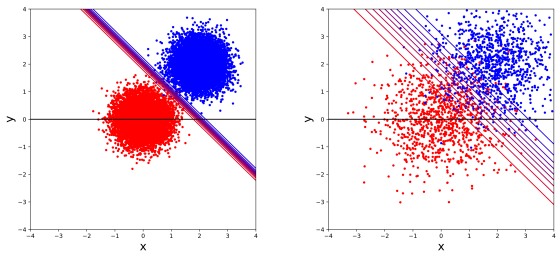


Figure 4.12: Two stochastic processes each generate data points that are Gaussian distributed. The red dots belong to class $\mathcal{C}_1$, the blue dots to class $\mathcal{C}_2$. Isolines for the probability $p(\mathcal{C}_1) = 0.1 \cdots 0.9$ (probability for a point being 'red') are given, calculated using Eq. 4.7. When we restrict x_2 to 0 (black horizontal line), a one dimensional sigmoid emerges.

As an example, consider Gaussian class-conditional functions:

$$p\left(\mathbf{x} \mid \mathcal{C}_k\right) = \frac{1}{(2\pi)^{D/2}} \frac{1}{|\mathbf{\Sigma}|^{1/2}} \exp\left\{-\frac{1}{2}\left(\mathbf{x}-\boldsymbol{\mu}_k\right)^{\mathrm{T}} \mathbf{\Sigma}^{-1}\left(\mathbf{x}-\boldsymbol{\mu}_k\right)\right\} \quad (4.4)$$

A two-dimensional two class example is given in Fig. 4.12, 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. 4.2 and 4.3 we find:

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + w_0) \quad (4.5)$$

with:

$$\begin{aligned} \mathbf{w} &= \mathbf{\Sigma}^{-1}\left(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\right) \\ w_0 &= -\frac{1}{2}\boldsymbol{\mu}_1^{\mathrm{T}}\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_1 + \frac{1}{2}\boldsymbol{\mu}_2^{\mathrm{T}}\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_2 + \ln\frac{p\left(\mathcal{C}_1\right)}{p\left(\mathcal{C}_2\right)} \end{aligned} \quad (4.6)$$

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:

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \sigma(-\frac{2}{k}(x_1 - x_2 - 2)) = \frac{1}{1 + e^{-\frac{2}{k}(x_1 + x_2 - 2)}} \quad (4.7)$$

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. 3.6). 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. 4.12, 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. 4.6 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 [2]. 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.

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

Exercise: Use Eq. 4.5 and 4.6 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. 4.6 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:

$$\boldsymbol{\mu}_1 = \frac{1}{N_1} \sum_{i=1}^{N} t_i \boldsymbol{x}_i, \tag{4.8}$$

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

$$\boldsymbol{\mu}_2 = \frac{1}{N_2} \sum_{i=1}^{N} (1 - t_i) \boldsymbol{x}_i, \tag{4.9}$$

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:

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

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

We can now use Eq. 4.6 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.

4.1.1 *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. 4.3.

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.

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

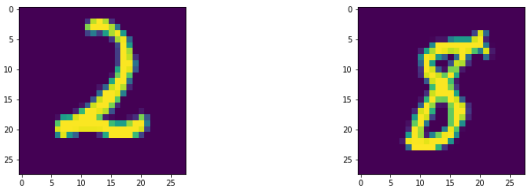


Figure 4.13: Two numerals, handwritten by humans; they are part of the MNIST dataset.

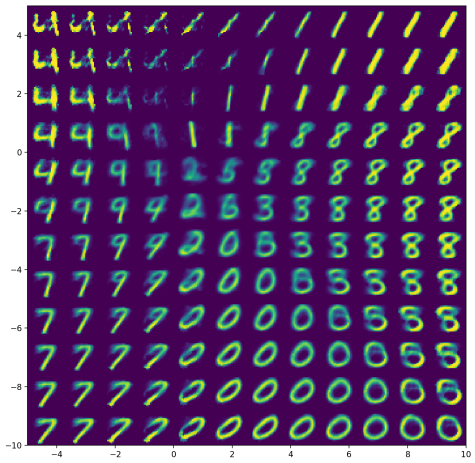


Figure 4.14: Images that have been sampled from a generative model of human handwriting.

like interpolations of real people. A simpler example is shown in Fig. 4.13 where a number of handwritten numerals are shown together with one that has been synthetically generated (Fig 4.14).

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 dimension 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.

4.2 *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. 4.15: 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.

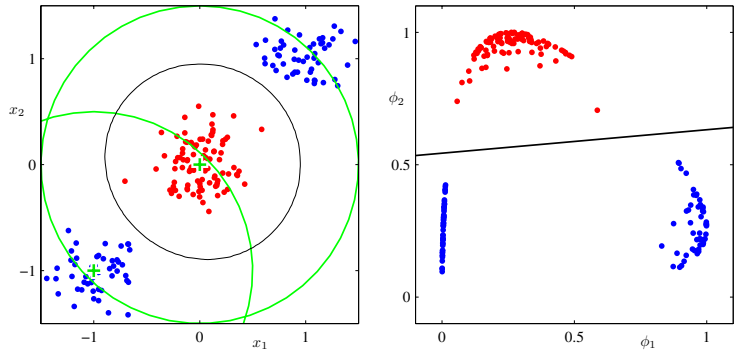


Figure 4.15: A well chosen non linear transformation of the data can often reduce in a much simpler classification problem. Two 'Gaussian' basis functions are shown, their centres represented by green crosses, and contours by green circles in the left hand plot. The right hand plot shows the data in feature space (ϕ_1, ϕ_2). Here the problem is linearly separable. This figure is Fig. 4.12 from Bishop (2006).

How does one find these basis functions? Often they suggest themselves: the transformation in Fig. 4.15 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* in Lesson **??**.

4.3 *Logistic Regression: Discriminative Modelling*

In Sec. 3.3 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. 3.9). This leads to elegant expressions for learning rules but not always to efficient learning.

In Sec. 4.1 we presented a generative model for labelled data that provides a good explanation for why the sigmoid function often emerges in practice, but the approach presented there requires us to learn the parameters for two Gaussian distributions before we can settle on reasonable weights. Sometimes this is helpful, as it provides good insight into how the data could have been generated, but for the purpose of just determining a good set of weights it may be overkill.

A more direct approach for determining the weights directly from the data is desirable. Again, we model the data with a logistic regression function, i.e. we assume that somehow the generative model described earlier underlies the data generation process, but we do not intend to model the data generation process itself. We take the assumption of a logistic function to model probabilities for granted and intend to find weights such that they determine

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

with

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

produces the best classification possible.

To this end, we define a likelihood function. Let the dataset be $\{\boldsymbol{\phi}_i, t_i\}$ with $\boldsymbol{\phi}_i = \boldsymbol{\phi}(\boldsymbol{x}_i)$ 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:

$$p(\boldsymbol{t} \mid \boldsymbol{w}) = \Pi_{i=1}^{N} y_i^{t_i} (1 - y_i)^{1 - t_i}, \tag{4.10}$$

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

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

$$\mathcal{E}(\boldsymbol{w}) = -\ln p(\boldsymbol{t} \mid \boldsymbol{w}) = \sum_{i=1}^{N} t_i \ln y_i + (1 - t_i) \ln(1 - y_i) \tag{4.11}$$

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:

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{w}} = \sum_{i=1}^{N} (y_i - t_i) \boldsymbol{\phi}_i \tag{4.12}$$

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:

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

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. 3.5 and 4.11. It is worthwhile commenting on the difference. Let us recall the definition of the Kullback-Leibler divergence from Unit 1.

$$\mathrm{KL}(p \mid\mid q) = -\int p(\boldsymbol{x}) \ln q(\boldsymbol{x}) d\boldsymbol{x} - (-\int p(\boldsymbol{x}) \ln p(\boldsymbol{x}) d\boldsymbol{x}) \tag{4.13}$$

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. 4.11 as the cross entropy. Eq. 4.12 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. 4.13 is a only function of the true distribution, which we use as reference.

4.4 *Iteratively Reweighted Least Squares*

In general, stochastic gradient descent is not used in logistic regression, but a second order method called *iterative reweighted least squares*. Second order methods are far more efficient for logistic regression than stochastic gradient descent, despite the overheard of not just having to calculate the gradient, but also the *Hessian*, a matrix containing the values of all second order derivatives at some point. We will sketch the method in one dimension. It is important to know about this method, as it is widely used an built in many data

analysis packages. Unlike steepest gradient descent rules, we do not suggest that you will implement them yourself. But there is value in understanding the underlying ideas, in particular that one can use second order derivatives.

The Newton-Raphson is a method for finding minima of a function. Suppose that we choose a point x_0, which is in the neighbourhood of some point x^* which is the minimum of some function $f(x)$. We want to create a sequence $\{x_k\}$, which converges to x^*. The method achieves this by fitting a second order polynomial, a parabola, to the current value of x_k, and then finding the value of x which is the minimum for that parabola. This will be our new value for x_{k+1}. A Taylor approximation up to second order gives the desired second order polynomial as a function of t:

$$f(t) \approx f(x_k) + (t - x_k)f'(x_k) + \frac{1}{2}(t - x_k)^2 f''(x_k)$$

It will reach a minimum for

$$t - x_k = -\frac{f'(x_k)}{f''(x_k)}$$

This gives:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This presupposes that the parabola actually has a minimum. This is only the case if our starting value is actually close enough to a minimum.

The method generalises straightforwardly to higher dimensions. It then gives an iterative way for minimising the loss function:

$$\boldsymbol{w}^{(new)} = \boldsymbol{w}^{(old)} - \boldsymbol{H}^{-1}\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} \tag{4.14}$$

Here $\boldsymbol{H}$ is the so-called Hessian matrix, whose elements comprise the second derivatives of $\mathcal{L}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$:

$$\boldsymbol{H} = \begin{pmatrix} \frac{\partial^2 \mathcal{L}}{\partial w_1^2} & \frac{\partial^2 \mathcal{L}}{\partial w_1 \partial w_2} & \cdots & \frac{\partial \mathcal{L}}{\partial w_1 \partial w_N} \\ \frac{\partial^2 \mathcal{L}}{\partial w_2 \partial w_1} & \cdots & \cdots & \frac{\partial \mathcal{L}}{\partial w_1 \partial w_N} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_N \partial w_1} & \cdots & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_N^2} \end{pmatrix}$$

In matrix vector notation we often use the notation:

$$\boldsymbol{H} = \nabla\nabla\mathcal{L}(\boldsymbol{w}),$$

to indicate that the Hessian is a matrix constructed from second order derivatives.

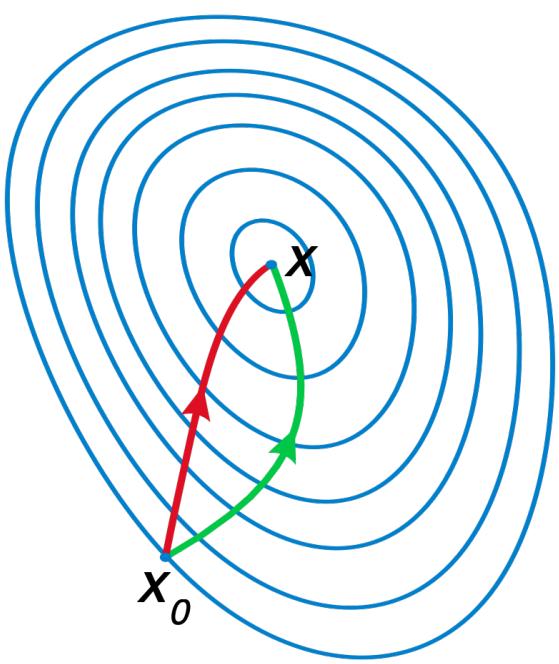


Figure 4.16: Newton-Raphson uses local information about the curvature that is code in the Hessian. It is able to follow a shorted path to the minimum (green) than steepest gradient descent (red). Source: Wikipedia.

In general, Newton-Raphson requires far fewer steps than stochastic gradient descent, as it follows the local curvature as well as the gradient (Fig. 4.16). It can go astray when the starting point is close to a saddle point instead of a minimum. For logistic regression we will show that there are no saddle points, just a global minimum and therefore second order methods are the method of choice for logistic regression. Their application also provides us with a numerical estimate of the Hessian in the minimum, which sets us up nicely for an approximate Bayesian logistic regression (Sec. 4.5).

Let us apply this method to two cases, linear and logistic regression. We are already familiar with the loss function of linear regression (Unit 1), and we have an explicit formula for linear regression: the *ordinary least square* regression formula. It is therefore interesting to see how the method of Newton-Raphson works out in this case.

Remember that in linear regression we wanted to minimise the loss function:

$$\mathcal{L} = \frac{1}{2}\sum_{i=1}^{N}\left\{t_i - \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}_i)\right\}^2, \tag{4.15}$$

where $\boldsymbol{x}_i, t_i$ is the dataset, $\boldsymbol{\phi}$ the fixed basis functions that we used as regression functions, e.g. polynomials. It is a relatively straightforward calculation to establish that:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \sum_{i=1}^{N}(\boldsymbol{w}^T\boldsymbol{\phi}_i - t_i)\boldsymbol{\phi}_i$$

and

$$\boldsymbol{H} = \nabla\nabla\mathcal{L} = \sum_{i=1}^{N} \boldsymbol{\phi}_i \boldsymbol{\phi}_i^T$$

Now, remember that in Unit 1 we introduced the design matrix $\boldsymbol{\Phi}$, where $\boldsymbol{\Phi}$ is an $N \times M$ matrix whose i-th row is given by $\boldsymbol{\phi}_i^T$. We can concisely summarise our results by:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \boldsymbol{\Phi}^T \boldsymbol{\Phi} \boldsymbol{w} - \boldsymbol{\Phi}^T \boldsymbol{t},$$

and

$$\boldsymbol{H} = \nabla\nabla\mathcal{L} = \boldsymbol{\Phi}^T \boldsymbol{\Phi}$$

The Newton-Raphson equations now take the form:

$$\begin{aligned} \boldsymbol{w}^{(new)} &= \boldsymbol{w}^{(old)} - (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \left\{ \boldsymbol{\Phi}^T \boldsymbol{\Phi} \boldsymbol{w}^{(old)} - \boldsymbol{\Phi}^T \boldsymbol{t} \right\} \\ &= (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{t} \end{aligned} \tag{4.16}$$

In other words, the new weights *are* the ordinary least square solution (OLS; see Unit 1). This should not be a surprise: Eq. 4.15 is a quadratic equation in the weights, so finding its minimum actually is the desired solution.

We can now play the same game with the loss function that we derived for the logistic regression model, Eq. 4.12. We find:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \sum_{i=1}^{N} (y_i - t_i) \boldsymbol{\phi}_i = \boldsymbol{\Phi}^T (\boldsymbol{y} - \boldsymbol{t})$$

and

$$\boldsymbol{H} = \sum_{i=1}^{N} y_i (1 - y_i) \boldsymbol{\phi}_i \boldsymbol{\phi}_i^T = \boldsymbol{\Phi}^T \boldsymbol{R} \boldsymbol{\Phi}, \tag{4.17}$$

where we introduce the $N \times N$ diagonal matrix with components $R_{ii} = y_i(1 - y_i)$. Here, it is important to remember that

$$y_i = \sigma(\boldsymbol{w}^T x_i)$$

with σ the logistic function. This implies that $0 < y_i < 1$, and since $\boldsymbol{R}$ is a diagonal matrix, it follows that $\boldsymbol{u}^T \boldsymbol{H} \boldsymbol{u} > 0$ for any vector $\boldsymbol{u}$. The Hessian is *positive definite*. It follows that the error function Eq. 4.12 is a concave function of $\boldsymbol{w}$ and has a unique minimum.

So $\boldsymbol{R}$. It is a function of the weights, which corresponds to the loss function no longer being quadratic. We can still apply the Newton-Raphson scheme, but cannot hope to arrive at the minimum after one step. Instead, now we need to treat it as an iterative algorithm. Using the results for the gradient and Hessian, we find:

$$\begin{aligned} \boldsymbol{w}^{(new)} &= \boldsymbol{w}^(old) - (\boldsymbol{\Phi}^T \boldsymbol{R} \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T (\boldsymbol{y} - \boldsymbol{t}) \\ &= (\boldsymbol{\Phi}^T \boldsymbol{R} \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \boldsymbol{R} \boldsymbol{z}, \end{aligned} \tag{4.18}$$

where

$$z = \Phi w^{(old)} - R^{-1}(y - t).$$

Eq. 4.18 has the form of a ordinary least squares solution. For that reason, it is known as *iterative reweighted least squares* (IRLS). Its extension to multi class logistic regression is straightforward [3] (Section 4.3.4), but see the remark in Sec.**??** on computational demands.

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

4.5 Bayesian Logistic Regression

In Bayesian logistic regression, we would like to consider a prior probability distribution over the weights, which we then adapt using Bayes' rule. In order to do that we need the likelihood of the data, and we have that available. For the two class problem considered in the previous section such a likelihood is available, and as usual we consider its (negative) logarithm. This leads to the cross entropy expression 4.11:

$$E(w) = -\ln p(t \mid w) = -\sum_{i=1}^{N} \{t_i \ln y_i + (1 - t_i) \ln(1 - y_i)\} \qquad (4.19)$$

where $y_i = \sigma(w^T \phi_i)$ is a *non linear* function of the weights (remember σ stands for sigmoid).

Now, remember what happened in *linear regression*. We multiplied a Gaussian prior, with a Gaussian likelihood. The logarithm of these Gaussians are quadratic functions in the weights. When we multiply prior and likelihood, we have to add their log likelihoods which are both quadratic expressions in the weights, which result in a log likelihood for the posterior which again is quadratic in the weights. In other words, the posterior is also a Gaussian, and its parameters can be determined simply by adding the logarithm of the likelihood function and the logarithm of the prior.

If we try to follow that recipe here, we get stuck immediately. Adding a quadratic function of w to the right hand side of Eq. 4.19 neither leads to a quadratic expression, nor something resembling the functional form of the cross entropy, but to a mix of the two. This has its use, as we will see when we will discuss the Laplace Approximation in the next section. But it does mean that the posterior cannot be expressed as a Gaussian. This means that the posterior cannot neatly be presented as a parameterised distribution whose parameters can be calculated in a neat formula. Even if you were able to find a closed analytical expression for the posterior, in prediction you would have to marginalise the weights, which would lead to nasty multi-dimensional integrals.

What to do? There are essentially three options:

1. Monte Carlo simulation. This amounts to writing down a Gaussian prior and use Monte Carlo sampling to approximate the posterior distribution. Monte Carlo techniques will be discussed in Unit 4

2. Variational approaches. In variational approaches the posterior distribution is modelled as a mixture of known probability functions, e.g. mixtures of Gaussian. This simplifies the problem somewhat in that mixture coefficients must be found that approximate the posterior distribution. This approach is discussed in Unit 5.

3. The Laplace approximation. This approximates the posterior distribution by a Gaussian. We will discuss it below.

4.5.1 *The Laplace Approximation*

The Laplace approximation consist in approximating a distribution with a Gaussian. The assumption is that the distribution is unimodal, that we know its maximum (mode) and its second order derivatives (Hessian). Conveniently, we know the expression for the Hessian is given by Eq. 4.17. The Laplace approximation consists of setting the posterior distribution equal to a Gaussian with the Hessian as precision matrix in the maximum found by the numerical method. To do this, start with a Gaussian prior:

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

and then maximise the likelihood function:

$$\ln p(\boldsymbol{w} \mid \boldsymbol{t}) = -\frac{1}{2}(\boldsymbol{w}-\boldsymbol{m}_0)^T \boldsymbol{S}_0^{-1}(\boldsymbol{w}-\boldsymbol{m}_0) + \sum_{i=1}^{N} t_i \ln y_i + (1-t_i)\ln(1-y_i),$$

with $y_i = \sigma(\boldsymbol{w}^T \boldsymbol{\phi}_i)$.

The resulting maximum of the likelihood, which can be found using bt the numerical methods we already described is called the *maximum a posteriori*. The corresponding weights are $\boldsymbol{w}_{\text{MAP}}$. The covariance is given by the inverse of the Hessian of the negative log likelihood:

$$\boldsymbol{S}_N = \boldsymbol{S}_0^{-1} + \sum_{i=1}^{N} y_i(1-y_i)\boldsymbol{\phi}_i \boldsymbol{\phi}_i^T$$

In the notebook *Bayesian Logistic Regression* we will demonstrate the practical use of some of the techniques discussed in this section.

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.

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.

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.

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.

MARC DE KAMPS

MACHINE LEARNING - UNIT 3 (PART 2) (v1.0)

UNIVERSITY OF LEEDS

Contents

List of Figures

List of Tables

0.1 Multi-class Logistic Regression

So far, we have discussed two-class logistic regression. This is often used to model a binary outcome, but often we want to consider the case where there are multiple possible outcomes, of which only one will be realised. The iris dataset can serve as a model here: any iris in the dataset belongs to one and only one class. The generative model introduced in Sec. **??** can be extended. In a similar manner one can construct the case for three Gaussians and for a given data point 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:

$$p(\mathcal{C}_k \mid \boldsymbol{\phi}) = y_k(\phi) = \frac{\exp a_k}{\sum_j \exp a_j}, \tag{1}$$

with

$$a_k = \boldsymbol{w}_k^T \phi.$$

Eq. 1 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. 1 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.

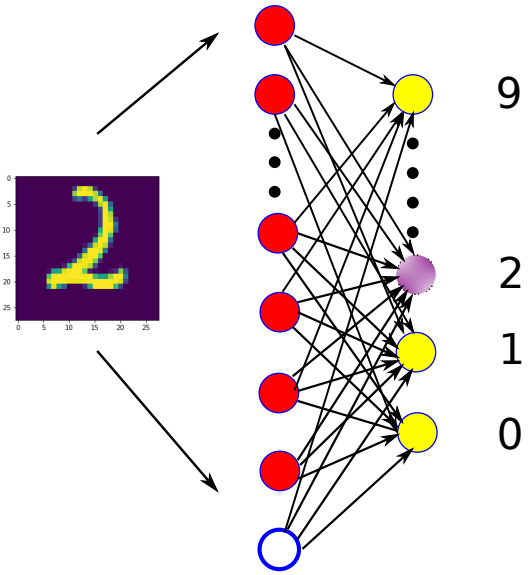


Figure 1: An example of multi-class logistic regression. There are 785 input variables: $28 \times 28 = 784$ pixels and a bias node. There are ten classes. The desired output is a 1-over-10 ('one hot') representation of the number that the regressor believes the image represents. For the regressor the 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. 1 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:

$$\frac{\partial y_k}{\partial a_j} = y_k(\delta_{kj} - y_j), \tag{2}$$

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 '0'. 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.

In *Activity: Logistic Regression* you will investigate wether the iris dataset can be classified using logistic regression.

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

0.1.1 *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.

1 Multilayer-Perceptrons

1.1 Introduction

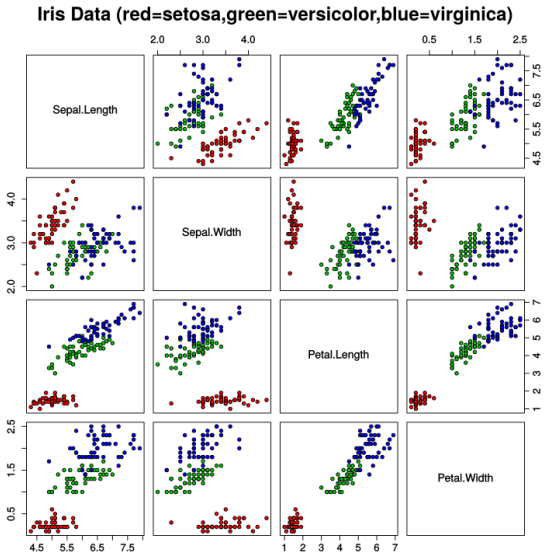


Figure 1.2: The Iris dataset represented, wastefully, as a set of scatter plots between the four dimensions of each flower.

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. 1.2. 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.

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

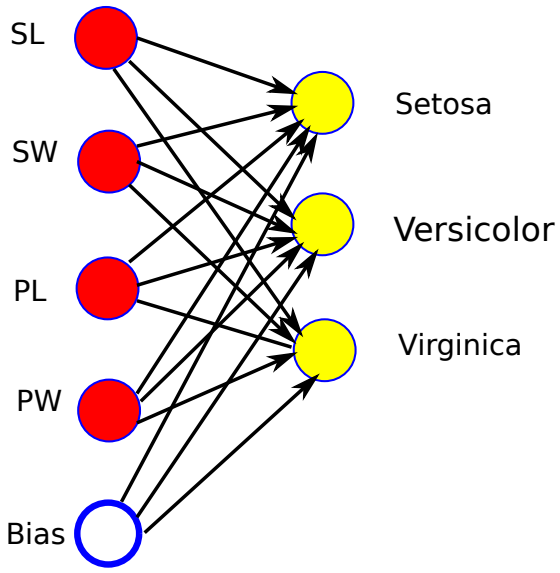


Figure 1.3: A two-layer neural network built from three perceptrons, run in parallel. It has four input nodes: *petal length*, *petal width*, *sepal length*, *sepal width* and a single bias node, whose value is always 1. Each perceptron takes a yes/no decision about one of the three classes: setosa, 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.

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. 1.3. The network can be expressed concisely as follows:

$$o_i = f(\sum_{j=1}^{4} w_{ij} x_j), i = 1, 2, 3 \tag{1.1}$$

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. 1.2 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. 1.2, 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.

1.2 *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:

$$\boldsymbol{o} = f(\boldsymbol{W}\boldsymbol{h}) = f(\boldsymbol{W}g(\boldsymbol{V}\boldsymbol{i})) \tag{1.2}$$

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 hidden nodes, and O output nodes. $\boldsymbol{V}$ is a $H \times I$ matrix, and $\boldsymbol{W}$ is

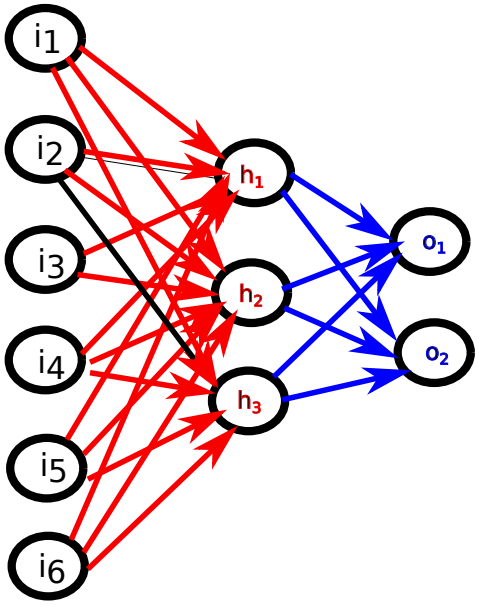


Figure 1.4: A fully connected network. The red connections can be organised in 3×6 matrix $\boldsymbol{V}$, the blue connections in $1\ 2 \times 3$ matrix $\boldsymbol{W}$.

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:

$$o_i = f(\sum_j w_{ij} g(\sum_k v_{jk} i_k)) = f(\sum_j w_{ij} h_j) \tag{1.3}$$

Eq. 1.3 is a very fundamental equation. It describes how multi-layer 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:

$$\mathcal{L} = \sum_i \frac{1}{2}(\boldsymbol{o}_i - \boldsymbol{d}_i)^2 \tag{1.4}$$

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. 1.4 runs over *data points*, whereas the summations in Eq. 1.3 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.

1.3 *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:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \sum_i (\boldsymbol{o}^{(i)} - \boldsymbol{d}^{(i)}) \frac{\partial o^{(i)}}{\partial \boldsymbol{w}} \tag{1.5}$$

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:

$$\frac{\partial \mathcal{L}}{\partial w_{pq}} = \sum_k (o_k - d_k) \frac{\partial o_k}{\partial w_{pq}} \tag{1.6}$$

Since,

$$\frac{\partial o_k}{\partial w_{pq}} = f'(\sum_j w_{kl} h_l) \frac{\partial \sum_l w_{il} h_l}{\partial w_{pq}} = f'(\sum_j w_{kl} h_l) \sum_l \delta_k^p \delta_l^q h_l \tag{1.7}$$

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:

$$\frac{\partial o_k}{\partial w_{pq}} = f'(\sum_j w_{kj} h_j) \delta_i^p h_q \tag{1.8}$$

Substituting this back into Eq. 1.6 gives:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{pq}} &= (o_p - d_p) f'(\sum_l w_{kl} h_l) h_q \\ &= \Delta_p^W h_q, \end{aligned} \tag{1.9}$$

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:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{pq}} &= \sum_k (o_k - d_k) f'(\sum w_{kl} h_l) w_{kp} g'(\sum_m v_{pm} i_m) \\ &= \sum_k \Delta^w w_{kp} g'(\sum_m v_{pm} i_m) i_q \\ &= \Delta_p^W i_q, \end{aligned} \tag{1.10}$$

Here, we introduced:

$$\Delta^V = \sum_k \Delta_k^W w_{kp} g'(\sum_m v_{pm} i_m) \tag{1.11}$$

Eq. 1.11 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. 1.5 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.

1.4 The Role of the Hidden Layer: A Heuristic Explanation

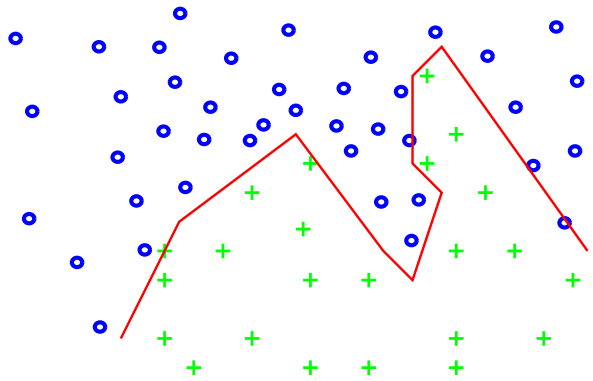


Figure 1.5: A multilayer perceptron can be naively thought of as using higher layers to combine lower level decision boundaries, which are linear, into more complex non-linear ones.

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. 1.5. 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. 1.3 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.

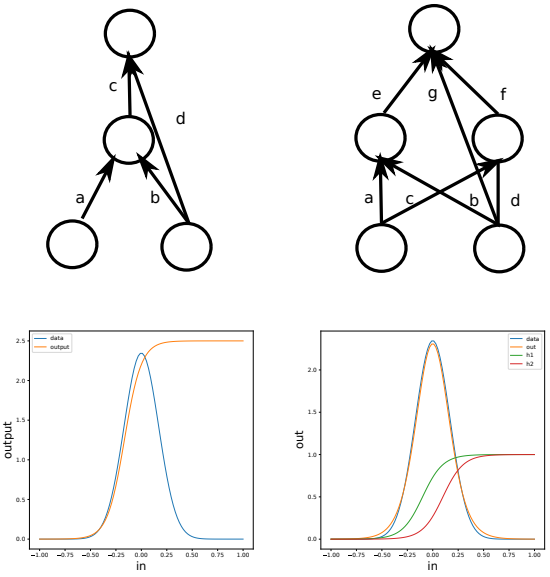


Figure 1.6: Two of the simplest neural network architectures conceivable. The one with one node is only able to capture one trend in the data. At least two nodes are necessary to capture data with a peak. Observe that the two hidden nodes respond in a similar way, but their responses combined captures the data quite well.

1.5 *Interpretation of the Backpropagation Algorithm*

Equation 1.3 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. 1.11 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. 1.10. 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 1.11 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. 1.11. 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.

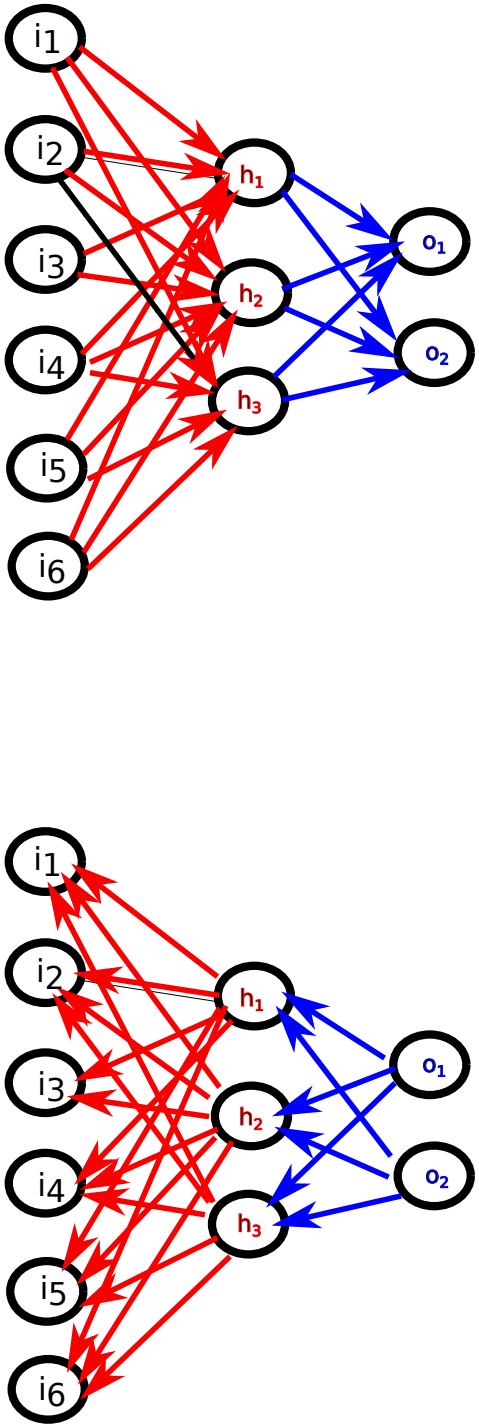


Figure 1.7: The update rule for the forward network is $o = f(\boldsymbol{W}g(\boldsymbol{V}\boldsymbol{i}))$. The update rule for the reverse network is $i = g(\boldsymbol{V}^T f(\boldsymbol{W}^T \boldsymbol{o}))$.

Bibliography

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

Classification with Decision Trees

In this unit we will learn about the general classification framework. In particular, we will cover an important and pervasive technique in classification, namely decision trees. We will also look at methods of evaluating a classification technique.

Learning outcomes:

After completing this lesson you should be able to:

- understand the deduction and induction process for decision trees.
- understand the CART algorithm
- splits of trees, using impurity measures
- use cross validation and GRID method to optimise prediction model hyper parameters
- evaluate a general classification technique using extensive set of performance metrics that depend on confusion matrices.

In a classification scenario, we have a supervised learning problem with labels. The labels themselves are a categorical data, i.e. they have a set of confined values, be it numerical or categorical that we can enumerate through. We call these values, the classes, and our task is to take an input data point and place it in the area or under one of the classes. This type of task is pervasive in all aspects of data mining, and we will find ourselves facing it in one form or the other in several settings. One distinction we need to make here is between classification and clustering. While classification uses predefined classes with names, clustering is unsupervised learning, in which data points are placed in clusters based on their intrinsic properties and the similarities between them.

There are numerous techniques for classification, but in this unit we will focus on a simple and widely used technique called Decision Trees. In later units of this module and in other modules such Machine Learning, Deep Learning and Text Analytics, you will come across several other techniques, building on the ideas that we develop here and in later units. Ok let us get started…

Figure (1) shows a schematic representation of a classification model. As it can be seen, the classification model maps an input x with an output y. the input is a set of attributes for one record and the output is the predicted class of the record. The record can be any object or entity represented in our dataset as one record.

Figure (11.): A Schematic Illustration of classification.

In the next few lessons we cover a common classification technique. Namely we will cover decision trees. We will see also their strengths and limitations and when they are most suitable for which type of problems. We will demonstrate important concepts along the side of the technique such as applying data preparation and performance measure.

Many problems are actually classification problem in disguise and our mission as data scientist is to spot them and be able to engineer the associated data if necessary to make it suitable for the types of techniques that we want to apply.

1 DECISION TREES

In this lesson we will see how to build a decision tree using CART induction algorithm. We also gain an insight into using impurity measures, including Gini impurity, entropy, and misclassification error. You will use these techniques to come up with a powerful model that will have the ability to classify records which we don't have the label for. We call these unseen records or unlabelled records.

Learning outcomes:

After completing this lesson you should be able to:

- understand the intricate details of the induction process
- explain how the impurity measures work in combination with the splitting procedure
- understand the differences between splitting a continuous variable, and splitting categorical variables.
- carry out deduction using a decision tree and induction to build the tree.

Let us assume that we have the following binary tree structure:

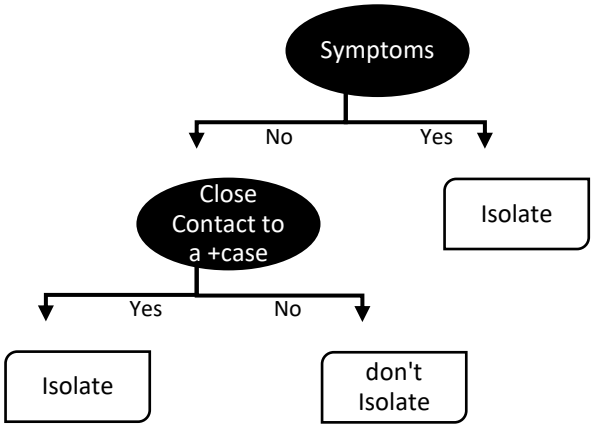


Figure (1.1): A binary tree structure.

In this diagram you can see that we have used leaves (round edged rectangles) and nodes (ovals). The nodes represent the conditions that need to be checked, while the leaves represent a decision to isolate or not to isolate. This is a binary tree since each condition has two results only (either yes or no). In other words each node can have two children only representing the two possible results of the condition that the node represents. The tree structure represents a binary decision tree. Note that DTs do not necessarily need to be binary, each node can have any number of children. However, a condition of a tree structure is for each node to have one and only one parent (if not then it is just a graph- a tree is special type of a graph). This condition helps the tree to satisfy several guarantees that simplify the inference and its inception process. Ok, so you might be asking now, what do you mean inference? We talk about it in the next section.

1.1 DECISION TREES INFERENCE

An **inference** is the process of using a model (such as a DT) in order to **infer** what is the class (label in general) of a given record (case, or data point). The process of creating a DT is called **training the tree**. We will see an algorithm that shows us to build a DT automatically form the data. But first we will have a look at how we can infer the class of a case from the DT.

Let us assume that we have been given the following new case and we want our DT to tell the concerned person whether to isolate or not.

Name	Symptoms	Close Contact to a positive case	Isolate
Jasmin	No	Yes	?

The DT then checks first if the person has symptoms

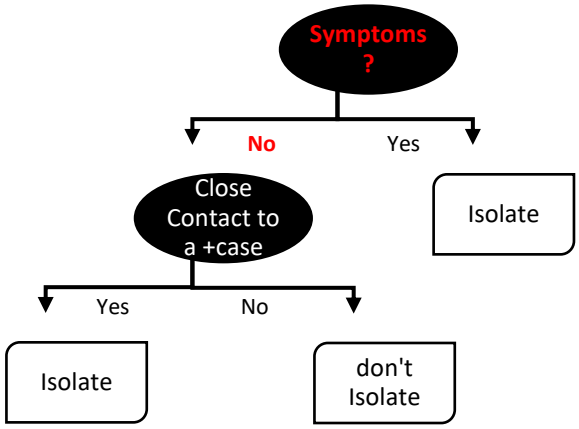


Figure (1.2): Inference in decision tree, following left hand side branch of level 0.

Since the person has no symptoms, the inference process will go to the next node to check if they have been in contact with a positive case recently:

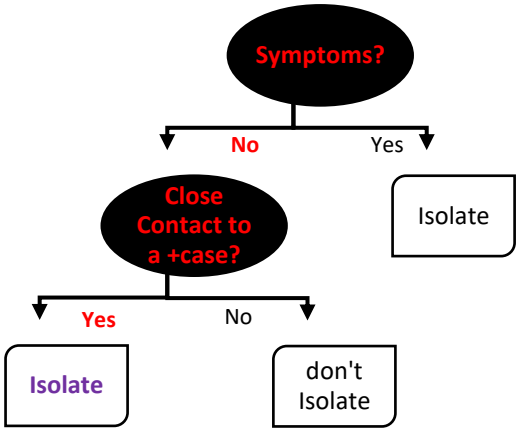


Figure (1.3): Inference in decision tree, following left hand side branch of level 1.

Name	Symptoms	Close Contact to a positive case	Isolate
Jasmin	No	Yes	Yes

So as we can see the path that has been taken by the tree is specified with a red colour, while the decision is in pink. Note that the same process can be performed live by an interactive system that is directly interacting with a user. However we will not be concerned with this ability in this unit, in fact the majority of what we cover in our module will assume that the

data has been already collected unless otherwise stated (for example if the data is sequential or coming from a time series). Let's look at another example:

Name	Symptoms	Close Contact to a positive case	Isolate
Joe	Yes	No	?

The inference process for the DT will look like the following:

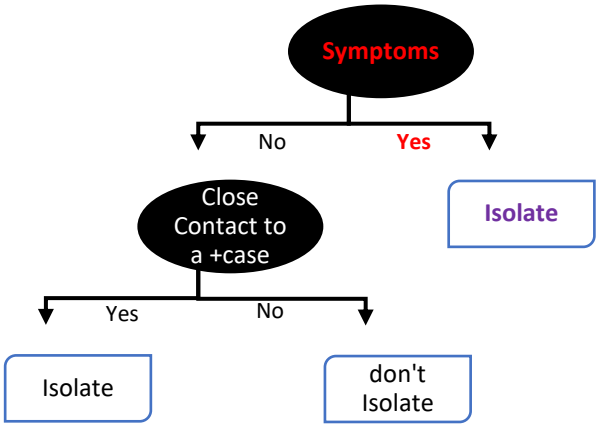


Figure (1.4): Inference in decision tree, following right hand side branch of level 0.

Name	Symptoms	Close Contact to a positive case	Isolate
Joe	Yes	No	Yes

As we can see the DT inference did not use all the available data checks since it can reach a conclusion without having to check for the second condition.

Those conditions constitute the features or the attribute for our data. The cases are the records, while decision is the label or the class of the case.

1.1.1 Example of an Invalid DT

The following structure is not valid decision tree since we have several possibilities of the same 'Has a Job' condition:

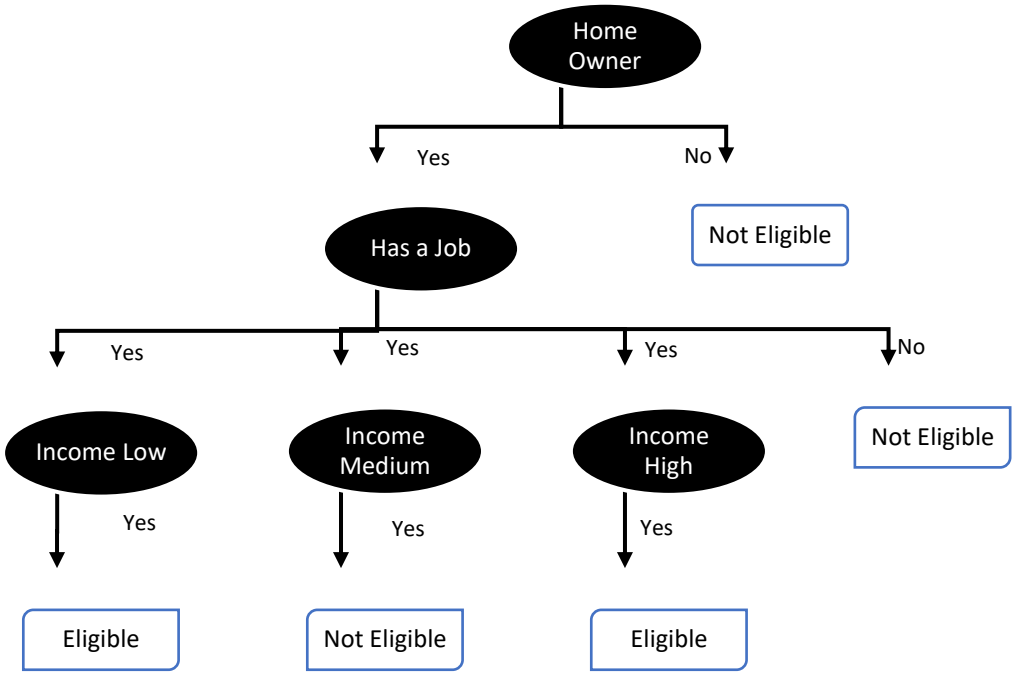


Figure (1.5): Invalid decision tree for loan eligibility. This decision tree is invalid due to multiple branches of the same value for 'Has a Job' feature.

We can alter it to make a valid decision tree as follows:

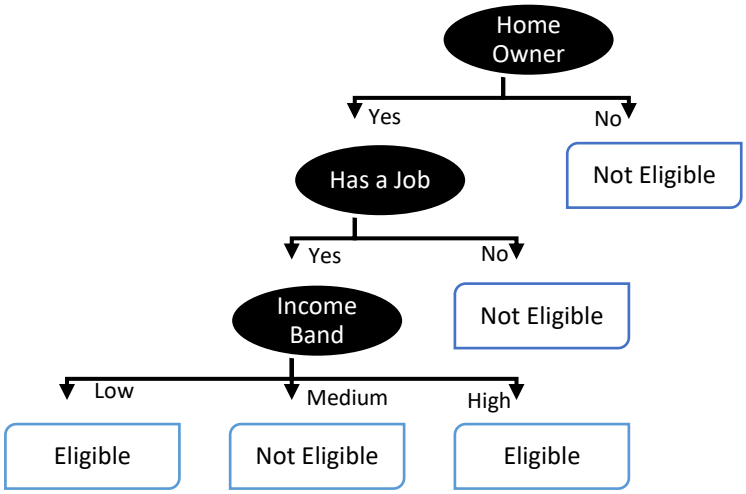


Figure (1.62.): A valid decision tree for loan eligibility.

So we can see that the graph becomes a valid DT by creating a new condition (attribute); the band of the income of the mortgage applicant which has three possible cases (band1, band2 and band3). So this DT is not binary. This is fine and we can conduct inference on this tree as before. Note that the tree is not binary but the **class** is binary {eligible, not eligible}. Given the following case:

Name	Home Owner	Has a Job	Income Band	Eligible
Emma	Yes	Yes	Band1	?

The decision tree path will be shown in red all at once (but bear in mind that it will be conducted in stages as we showed earlier)

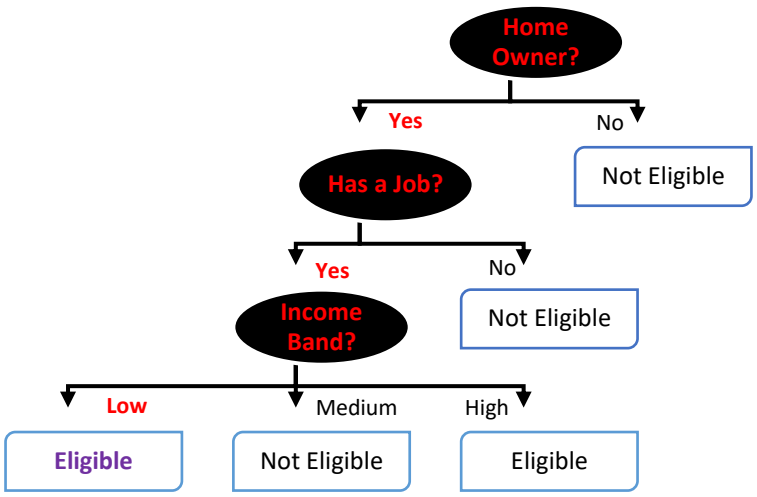


Figure (1.7): Loan eligibility decision tree, showing the inference path

Given that both cases of High and Low Bands are eligible then we can further simplify the tree as follows:

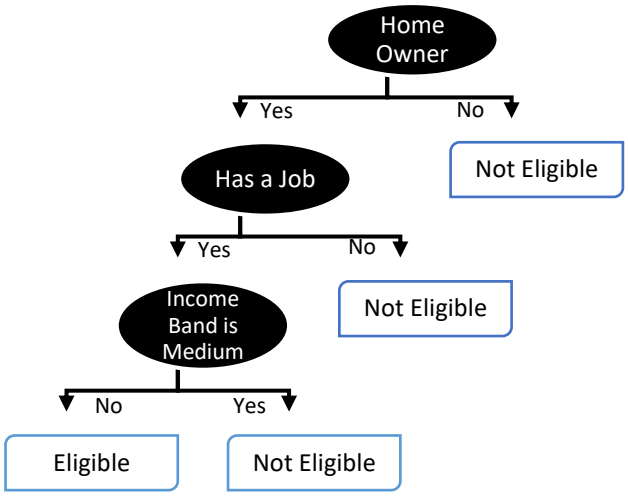


Figure (1.8): A better decision tree structure for loan eligibility.

1.1.2 EXERCISE:

Given the following cases, show the path that the inference process will take on the above DT:

Name	Home Owner	Has a Job	Income Band	Eligible
Jeff	Yes	No	Band1	?

Name	Home Owner	Has a Job	Income Band	Eligible
Steve	Yes	Yes	Band2	?

Name	Home Owner	Has a Job	Income Band	Eligible
Mitch	No	Yes	Band1	?

2 DT TRAINING (INDUCTION)

In this section, we cover the decision tree building algorithm which is also known as induction. We will look here at the CART algorithm.

In the simplistic dataset below, you can see a set of records relating to phones and tablets. The dataset allows us to classify a device that is not part of the dataset to find out whether it is a phone or a tablet.

In the dataset, the possible values for the features and the class are as follows:

Screen size = {6, 7, 8}, Makes calls = {Yes, No}, Class = {Phone, Tablet}

Devise	Screen Size	Makes Calls	Class
G1	6 inches	Yes	**Phone**
S1	6 inches	Yes	**Phone**
A1	7 inches	Yes	**Phone**
G2	7 inches	No	**Tablet**
S2	7 inches	No	**Tablet**
A2	8 inches	Yes	**Tablet**

Table (1): Devices dataset

Intuitively, the 'screen size' feature does not have an easy binary decision since in this dataset we have phones that are 6 and 7 inches and we have tablets that are also 7 or 8 inches. On the other hand 'makes calls' feature is more regular, **all** phones 'makes calls' while **most** tablets do not 'make calls'. This suggests that if we were to make decisions about a devise class being a phone or a tablet, we should first look at the 'makes calls' attribute and if it is 'no' then it is a 'tablet' if it is 'yes' then it is most likely a 'phone', we would need to check its 'screen size' if it is = 8 then it is a Tablet, if it is not then it is a phone.

So how we can create an algorithm that does this type of decisions for us. We need an algorithm that can strategically pick the more promising features first and then develop the tree based on that. The algorithm that we will talk about is called CART (Classification and Regression Tree) and we will show it in action in the following steps.

2.1 STEP (1) OF CART ALGORITHM

To be able to develop this algorithm we need to measure how promising a feature (with a specific value) is as a partition for our dataset. Think about the above dataset. The 'makes calls' feature allowed us to split the data two ways with a low impurity.

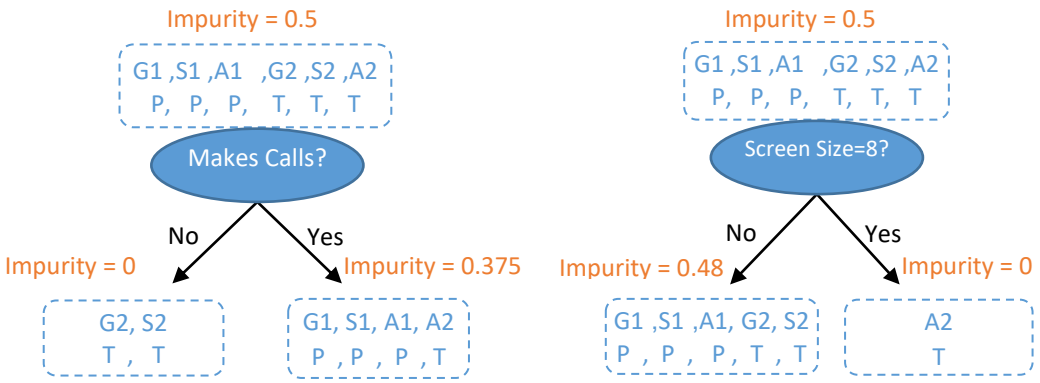


Figure (2.1): Illustration of step 1 of CART algorithm for tablet vs phone dataset. Left split based on 'makes calls' feature, rightsplit based on 'screen size=8'

To quantify the quality of each split, we use **Gini Impurity** of each **node data**. The Gini impurity describes how pure or mixed the data labels in a node. The purer the data the closer Gini is to 0, while the more mixed the data in the node the closer Gini is to 0.5. We will look at how to calculate the Gini Impurity a bit later. For now I want you to assume that you know how to calculate it.

2.1.1 Information Gain for Decision Trees

The **Information Gain of a split** refers to how much information we gain by choosing one of the possible splits. If the decision tree is binary, then each feature is a possible split. If it is not binary, then each pair of a feature with a value is represented as (feature, value), and will constitute a possible split.

If we would like to decide which of the above two splits is better, then we just calculate the information gain and we choose the one that has the highest information gain.

$$Information\ Gain\ =\ Impurity\ of\ parent - weighted\ average\ impurity\ of\ children$$

$$Info\ Gain('makes\ calls')\ = 0.5 - \left(\frac{2}{6} \times 0 + \frac{4}{6} \times 0.375\right) = 0.25$$

$$Info\ Gain('screen\ size = 8')\ = 0.5 - \left(\frac{5}{6} \times 0.48 + \frac{1}{6} \times 0\right) = 0.1$$

This shows that the first split based on the 'makes calls' feature is better since we gain form information by using it and we will intuitively move toward more pure leaves.

Please note that the term information gain is used also in Information Theory. It also uses the concept of entropy, but it refers to Kullback-Leibler Divergence which is the amount of information gained about one set of events when another related set of events are observed. The metric that we have been using in this unit, which we refer to also as information gain, is actually called mutual information

2.1.2 Gini impurity

It is time now to see how we calculate the Gini impurity. We take the probability of one label in the node and we multiply it with the probability of the other label (or sum of other labels probabilities if we have multi-class dataset). And we do that again for the second label and so on. So, to see this measure in action in a binary class dataset (like our dataset). Let us take the probability of an item being a tablet in our dataset, which we denote as $p(Tab) = \frac{3}{6} = 0.5$. This is because we have 6 items in total 3 of them are tablets. The same applies for the phones where we have $p(Pho) = \frac{3}{6} = 0.5$. Hence, the Gini Impurity of the dataset before any split is:

$$Gini\ Impurity(set) = p(Tab)[1 - p(Tab)] + p(Pho)[1 - p(Pho)]$$
$$= p(Tab) + p(Pho) - [p^2(Tab) + p^2(Pho)]$$

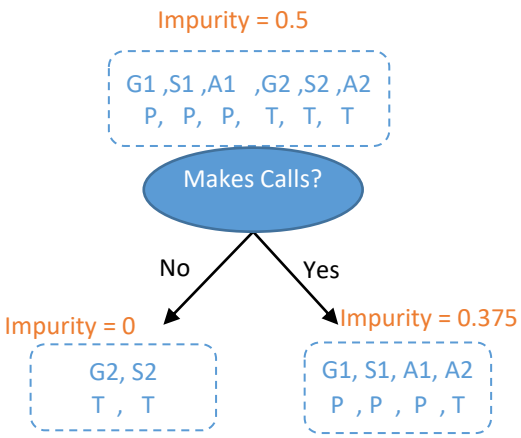
However we have $p(Tab) + p(Pho) = 1$, therefore:

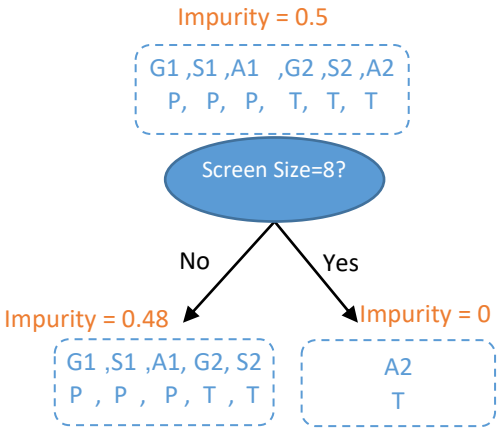
$$Gini\ Impurity(set) = 1 - [p^2(Tab) + p^2(Pho)]$$

Below we will see the calculations of the Gini Impurity for the subsets of the nodes (the items that belong to each node). We start always from the whole dataset and then the data will be distributed based on the type of question that we ask in the node. The summary of how we evaluate the two splits is in Figure (11).

Before split	$Gini\ Impurity\left(\begin{matrix}G_1S_1A_1G_2S_2A_2\\P\ P\ P\ T\ T\ T\end{matrix}\right) = 1 - \left[(\frac{3}{6})^2 + (\frac{3}{6})^2\right] = 0.5$	
After split – Left (No)	Impurity for children of 'makes calls' $Gini\ Impurity\left(\begin{matrix}G_2S_2\\T\ T\end{matrix}\right) = 1 - \left[(\frac{2}{2})^2 + (0)^2\right] = 0$	Impurity of children of 'screen size=8' $Gini\ Impurity\left(\begin{matrix}G_1S_1A_1G_2S_2\\P\ P\ P\ T\ T\end{matrix}\right) = 1 - \left[\left(\frac{3}{5}\right)^2 + \left(\frac{2}{5}\right)^2\right] = 0.48$
After split – Right (Yes)	$Gini\ Impurity\left(\begin{matrix}G_1S_1A_1A_2\\P\ P\ P\ T\end{matrix}\right) = 1 - \left[(\frac{1}{4})^2 + (\frac{3}{4})^2\right] = 0.375$	$Gini\ Impurity\left(\begin{matrix}A_2\\T\end{matrix}\right) = 1 - \left[(\frac{1}{1})^2 + (0)^2\right] = 0$
	$Info\ Gain('makes\ calls') = 0.5 - \left(\frac{2}{6} \times 0 + \frac{4}{6} \times 0.375\right) = 0.25$	$Info\ Gain('screen\ size = 8') = 0.5 - \left(\frac{5}{6} \times 0.48 + \frac{1}{6} \times 0\right) = 0.1$

$$Impurity\left(\begin{matrix}G_1S_1A_1G_2S_2A_2\\P\ P\ P\ T\ T\ T\end{matrix}\right) = 1 - \left[(\frac{3}{6})^2 + (\frac{3}{6})^2\right] = 0.5$$





$$Impurity\left(\begin{matrix}G_2S_2\\T\ T\end{matrix}\right) = 1 - \left[(\frac{2}{2})^2 + (0)^2\right] = 0$$
$$Impurity\left(\begin{matrix}G_1S_1A_1A_2\\P\ P\ P\ T\end{matrix}\right) = 1 - \left[(\frac{1}{4})^2 + (\frac{3}{4})^2\right] = 0.375$$
$$Gain('makes\ calls') = 0.5 - \left(\frac{2}{6} \times 0 + \frac{4}{6} \times 0.375\right) = 0.25$$

$$Impurity\left(\begin{matrix}G_1S_1A_1G_2S_2\\P\ P\ P\ T\ T\end{matrix}\right) = 1 - \left[\left(\frac{3}{5}\right)^2 + \left(\frac{2}{5}\right)^2\right] = 0.48$$
$$Impurity\left(\begin{matrix}A_2\\T\end{matrix}\right) = 1 - \left[(\frac{1}{1})^2 + (0)^2\right] = 0$$
$$Gain('screen\ size = 8') = 0.5 - \left(\frac{5}{6} \times 0.48 + \frac{1}{6} \times 0\right) = 0.1$$

Figure (2.2):Illustration of step 1 of CART algorithm for tablet vs phone dataset, showing information gain calculations.(left) split based on 'makes calls' feature, (right): split based on 'screen size=8'. Both with the full information gain calculations needed to decide which split to choose. In this case the left wins.

The algorithm will go ahead and calculate the information gain for another two splits possibilities, these are 'screen size=6' and 'screen size=7'. We have not shown these, so if you'd like to give it a try yourself you can do the calculations now. The results should be in favour of 'makes calls' split.

> Note: The Gini impurity deals with classes in a binary manner (one versus the rest). So, if we have multi-class dataset then we would simply enumerate through the different classes and deal with each label in the node as the target class and the rest as misclassification. For example, if we have say 3 classes, {'Tablet', 'Phone', 'Portable PC'} the Gini impurity will be:
>
> $$Gini\ Impurity(node\ set) = p(Tab)[1-p(Tab)] + p(Pho)[1-p(Pho)] + p(PC)[1-p(PC)].$$
>
> By noting that $p(Tab) + p(Pho) + p(PC) = 1$, we get
>
> $$Gini\ Impurity(node\ set) = 1 - [p^2(Tab) + p^2(Pho) + p^2(PC)]$$
>
> In the general case if we have I classes each with a probability $p(i)$ in the concerned node set then
>
> $$Gini\ Impurity(node\ set) = 1 - \sum_{c=1}^{I} p^2(i)$$

Note: Tan et al (2020) use Entropy and a slightly different algorithm for building the tree called Hunt's Algorithm, here we use the CART algorithm which is widely used for DT.

2.2 Step (2) of CART Algorithm

Next, the CART algorithm will convert the branch on the left of the 'makes calls' into a leaf since it's a pure node (all of its data point are of class 'Tablet'). The right hand side node is a mixture of 3 'Phones' and a 'Tablet'.

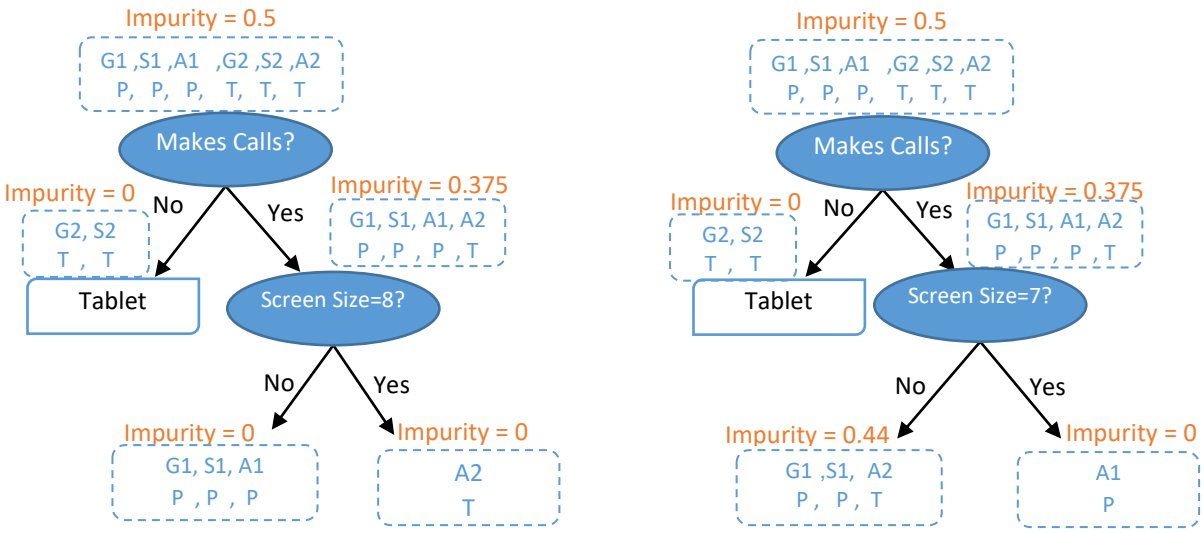
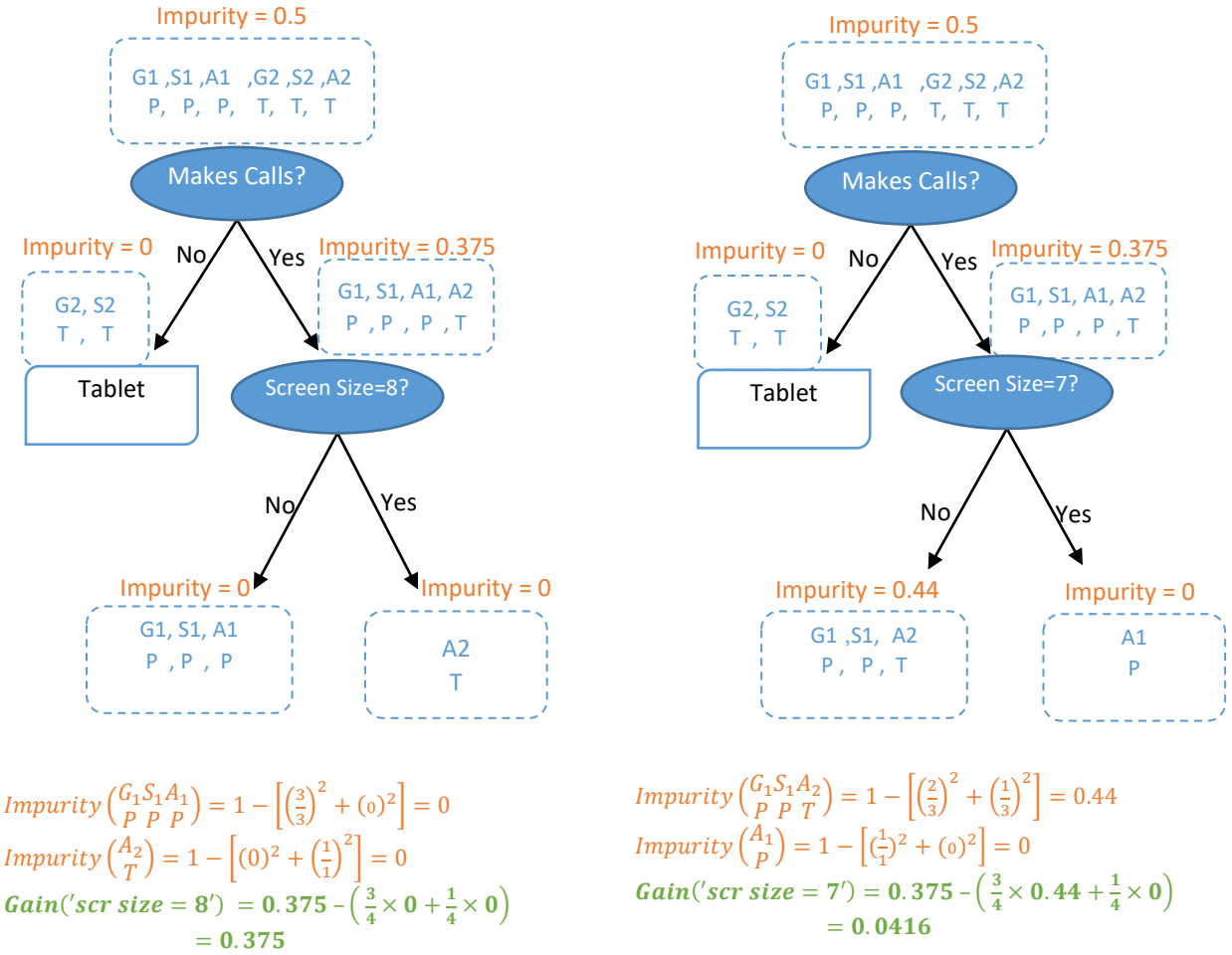


Figure (2.4): Illustration of step 2 of CART algorithm for tablet vs phone dataset. Left split based on 'screen size=8' feature, right split based on 'screen size=7'.

After we have chosen the 'makes calls' split where we have exhausted its different possibilities (the yes and no values), we move to the next feature 'screen size' (which happens to be the last feature that we have in our simple dataset). Since we said that there are only three values that this feature can take {6, 7, 8} we have three splits that can be done based on this feature. The algorithm will evaluate each split and we will choose the best one.

Before split	$Gini\ Impurity\left(\begin{matrix}G_1S_1A_1A_2\\P\ P\ P\ T\end{matrix}\right) = 1-\left[\left(\frac{3}{4}\right)^2+\left(\frac{1}{4}\right)^2\right] = 0.375$	
After split	Impurity of children of 'screen size=8'	Impurity of children of 'screen size=7'
Left (No)	$Gini\ Impurity\left(\begin{matrix}G_1S_1A_1\\P\ P\ P\end{matrix}\right) = 1-\left[\left(\frac{3}{3}\right)^2+(0)^2\right] = 0$	$Gini\ Impurity\left(\begin{matrix}G_1S_1A_2\\P\ P\ T\end{matrix}\right) = 1-\left[\left(\frac{2}{3}\right)^2+\left(\frac{1}{3}\right)^2\right] = 0.44$
Right (Yes)	$Gini\ Impurity\left(\begin{matrix}A_2\\T\end{matrix}\right) = 1-\left[(0)^2+\left(\frac{1}{1}\right)^2\right] = 0$	$Gini\ Impurity\left(\begin{matrix}A_1\\P\end{matrix}\right) = 1-\left[\left(\frac{1}{1}\right)^2+(0)^2\right] = 0$
	$Info\ Gain('screen\ size = 8') = 0.375 - \left(\frac{3}{4}\times 0 + \frac{1}{4}\times 0\right) = 0.375$	$Info\ Gain('screen\ size = 7') = 0.375 - \left(\frac{3}{4}\times 0.44 + \frac{1}{4}\times 0\right) = 0.0416$

If you'd like to try it yourself now, you can calculate the information gain for 'screen size=6'.The results should be in favour of 'screen size=8'. The final results are summarised in Figure (13)



$Impurity\left(\begin{matrix}G_1S_1A_1\\P\ P\ P\end{matrix}\right) = 1-\left[\left(\frac{3}{3}\right)^2+(0)^2\right] = 0$

$Impurity\left(\begin{matrix}A_2\\T\end{matrix}\right) = 1-\left[(0)^2+\left(\frac{1}{1}\right)^2\right] = 0$

$Gain('scr\ size = 8') = 0.375 - \left(\frac{3}{4}\times 0 + \frac{1}{4}\times 0\right) = 0.375$

$Impurity\left(\begin{matrix}G_1S_1A_2\\P\ P\ T\end{matrix}\right) = 1-\left[\left(\frac{2}{3}\right)^2+\left(\frac{1}{3}\right)^2\right] = 0.44$

$Impurity\left(\begin{matrix}A_1\\P\end{matrix}\right) = 1-\left[\left(\frac{1}{1}\right)^2+(0)^2\right] = 0$

$Gain('scr\ size = 7') = 0.375 - \left(\frac{3}{4}\times 0.44 + \frac{1}{4}\times 0\right) = 0.0416$

Figure (2.5): Illustration of step 2 of CART algorithm for tablet vs phone dataset (left) split based on 'screen size=8' feature,

Figure 2.6 Illustration of step 2 of CART algorithm for tablet vs phone dataset (right): split based on 'screen size=7'.

Both with the full information gain calculations needed to decide which split to choose. In this case the left wins.

Based on the above step, the algorithm will reach the following form:

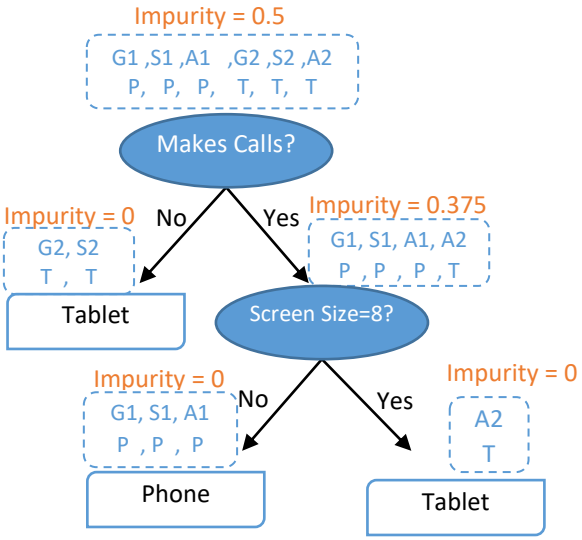


Figure (2.7): Final Step of CART algorithm tree induction (training).

At this stage the algorithm stops since all lower levels nodes are pure and produces the following final tree which can be used for inference as we did earlier in the previous section.

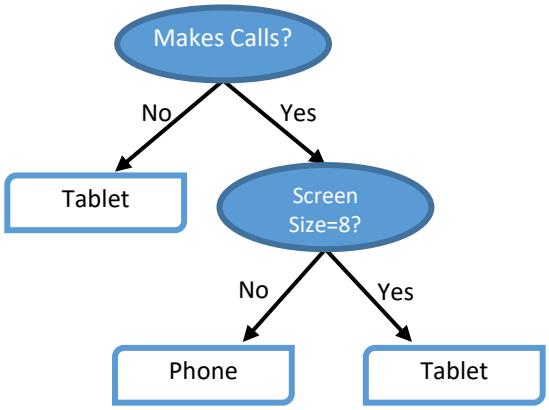


Figure (2.8): Final Tree structure after CART algorithm finished training for phone vs tablet dataset.

Exercise: mimic a DT inference to deduce the class of the following devices

Device	Screen Size	Makes Calls	Class
M1	7 inches	Yes	?
K1	8 inches	Yes	?

2.3 CART ALGORITHM

To summarise the CART algorithm does the following:

Algorithms 1: Build A Decision Tree (aka DT Induction)

Grow (*Data*, *Attributes*):
    If (number of rows |*Data*| < *h*): # *h is a Stopping Condition*
        Create (a *leaf* that has the dominant *label* of the *Data*) # *label* represents the class
        Return the *leaf* with its *label*
    Else:
        AttributesGain= **InfoGain** (Attributes) # *for possible Splits of Data based on Attributes)*
        attribute = **Max**(*AttributesGain*) # *best condition that yields max InfoGain*
        bsNode = **Create** (*attribute*) # *create a node that represents the best split*
        For each *value* of *attribute:*
            Data = **Split**(*attribute*, *value*) # *return the data based on the split*
            Child = **Grow** (*Data*, *Attributes*) # *recursive function call itself*
            bsNode = **Add** (*Child* to *bsNode)* and name the *edge(node* $\overline{value}$ *child)* # *label the edge for inference*
    Return *bsNode*

The above box shows the pseudocode for a decision tree induction algorithm. The algorithm works by expanding the tree using the best split attribute that yields the best information gain. E is a set of data inside a node and F is the set of attributes that we can use to split the data E.

DT Video1 for presentation part1

2.3.1 Discretising Continuous Variables

Given the following dataset, we want to build a decision tree that can predict whether or not a borrower is going to default on their debt. This type of decision is important for banks to decide upon the eligibility of customers to be lent money. While our dataset is simple, the ideas can be easily expanded into a fully developed scenario for an actual bank.

ID	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125	No
2	No	Married	100	No
3	No	Single	70	No
4	Yes	Married	120	No
5	No	Divorced	95	Yes
6	No	Married	60	No
7	Yes	Divorced	150	No
8	No	Single	85	Yes
9	No	Married	75	No
10	No	Single	90	Yes

Table (6): Borrowers dataset

- So far we have only dealt with categorical features. However, as you will see in the dataset above, the 'Annual Income' feature is not a categorical but a continuous feature. To be able to build a decision tree for the above dataset we can discretise the 'Annual Income' feature (in the next section we will see how to deal with it without discretisation). One way to discretise this feature is to partition the space values into a limited set of intervals and replace any value in the dataset by the interval that it falls into. We can give the intervals names to reflect them as categorical values of the continuous feature. The intervals can be evenly distributed or can have other distributions such as a normal distribution; this depends on the underlying distribution of the feature itself. If we think that all values are equally likely then a uniform distribution is suitable and we just divide the possible values into a set of equal ranges. For our dataset we can assume that Annual Income ranges are as shown in the following table and graph:

Annual Income Ranges £K	Annual Income Bands
[18, 48[	Basic
[48, 78[	Intermediary
[78, 108[	Advanced
[108, 138[	High
[138, 168[	Top
[168, [	Exec.

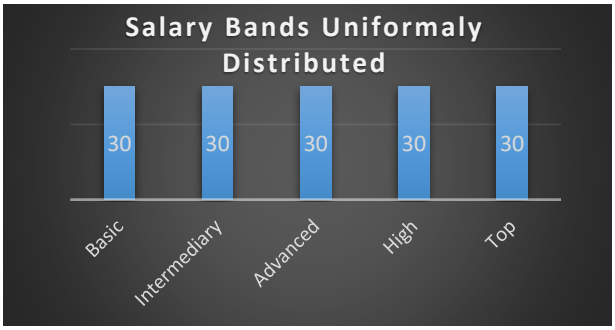


Figure (2.9): Annual Income bands uniformly distributed, note that the width of all the ranges are £30k.

- However, if we think that we might have values in the middle more than on the sides, then maybe we want to have more fine ranges in the middle and more coarse ranges on the side. We can partition the values into a set of ranges according with the a variable interval width range that is inversely normally distributed.

Annual Income Ranges £K	Annual Income Category
[18, 48[	Basic
[48, 66[	Intermediary L
[66, 78[	Intermediary H
[78, 86[	Advanced L
[86, 93[	Advanced M
[93, 100[	Advanced H
[100, 108[	Advanced T
[108, 112[	High L
[112, 130[	High H
[130, 160[	Top
[160, [	Exec.

Figure (2.10): Annual Income categories with an inverted normal distributed

- Another way to discretise is by looking into the domain of the model and how it is used. For example in the UK salaries are categorised according to tax bands as follows:

Annual Income Ranges £K	Annual Income Increment £K	Annual Income Category
54,900	2,300	Basic
57,700	2,800	Intermediary L
61,000	3,300	Intermediary H
65,000	4,000	Advanced L
70,200	5,200	Advanced M
76,800	6,600	Advanced H
86,000	9,200	Advanced T
98,600	12,600	High L
121,000	22,400	High H
175,000	54,000	Exec.

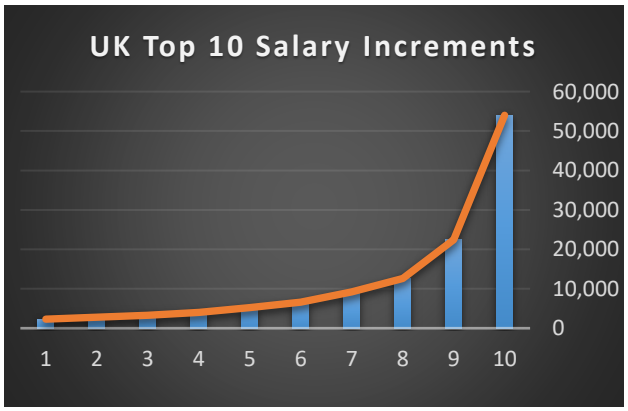


Figure (2.11): Top 10 UK actual annual income in 2018, the increments have reversed Pareto distribution.

As can be see the increments take a long tailed (skewed) distribution that is not a Gaussian, but more of a reversed Pareto distribution. This is not surprising as the Pareto distribution has historically been used to describe wealth in society. The 80-29 Pareto principle is related to this distribution but is precisely realised when the alpha value is 1.16. It takes the form:

$$\Pr(X > x) = \begin{cases} 1 - \left(\frac{x_{min}}{x}\right)^{\alpha} \; when \; x \geq x_{min} \\ 1 \; when \; x < x_{min} \end{cases}$$

Note that the categories' names {'Basic, 'Intermediary L', …, 'Exec.'} are arbitrary and could be changed to any values that suit the usage of the model (L, M, H, T stands for Low, Medium, High and Top, respectively).

2.4 SPLIT FOR CONTINUOUS VARIABLES

In the examples we have used so far we have seen how to split for features with categorical values. For 'Screen Size' we restricted the possibilities into 3 values, making it effectively categorical. We also discretised the 'Annual Income' by dealing with 3 ranges of salaries called 'bands', also effectively yielding it as categorical. However, discretisation is not always possible and can restrict the generalisation ability of our models. What we want to be able to do is to allow a continuous variable (such as 'Annual Income') to take any value for continuous variables, and we decide from the data what would be the best value to split the data according to. To understand the approach let us look at a tangible example. To deal with a split of continuous features we simply look into its values inside the available dataset. Remember we have in theory an infinite number of values so we cannot try them all!

1. We need to sort the dataset according to this feature, and we take the split values to be **in-between** the feature values in the dataset.

ID	Home Owner	Marital Status	Annual Income	Defaulted Borrower	Possible Splits for Annual Income
6	No	Married	60	No	
3	No	Single	70	No	65
9	No	Married	75	No	72.5
8	No	Single	85	Yes	80
10	No	Single	90	Yes	87.5
5	No	Divorced	95	Yes	92.5
2	No	Married	100	No	97.5
4	Yes	Married	120	No	110
1	Yes	Single	125	No	122.5
7	Yes	Divorced	150	No	172.5

Table (6): Borrowers dataset with possible splits for the Annual Income feature

2 We take the in-between values instead of the values themselves because we do not want to make any of the dataset records a boundary case. We do not need to worry about the first and last values since they cannot be a split condition otherwise they yield the feature ineffective- all data is greater than the first value and smaller than the last values. So if we have N records in our dataset (N=10 in the Borrowers dataset), we try N-1 in-between splits. See Table 6 above for the possible Splits for Annual Income after sorting the dataset according to 'Annual Income'.

3Then we now try to split according to each in-between value, and we calculate the Gini index and information gain for the results. We compare between all the information gain of the different split and we take the split that maximises the information gain. Note that all the calculations that we talked already about in the previous section applies. Since the original data Gini is not going to vary, we can simply take the split that minimises the Gini index since Information Gain = Gini for parent – Gini for the split. See table 7 below for the different Information Gain and Gini Index calculations.

Defaulted Borrower		No		No		No		Yes		Yes		Yes		No		No		No	
Annual Income(Sorted)		60		70		75		85		90		95		100		120		125	
Split Condition	All Data	<=65	>65	<=72.5	>72.5	<=80	>80	<=87.5	>87.5	<=92.5	>92.5	<=97.5	>97.5	<=110	>110	<=122.5	>122.5	<=172.5	>172.5
Yes (staisfy abv cond)	3	0	3	0	3	0	3	1	2	2	1	3	0	3	0	3	0	3	0
pr(Yes)	2/7	0	1/3	0	3/8	0	3/7	1/4	1/3	2/5	1/5	1/2	0	3/7	0	3/8	0	1/3	0
No (¬staisfy abv cond)	7	1	6	2	5	3	4	3	4	3	4	3	4	4	3	5	2	6	1
pr(No)	2/3	1	2/3	1	5/8	1	4/7	3/4	2/3	3/5	4/5	1/2	1	4/7	1	5/8	1	2/3	1
weight	1	0	8/9	1/5	4/5	2/7	2/3	2/5	3/5	1/2	1/2	3/5	2/5	2/3	2/7	4/5	1/5	8/9	0
Gini impurity	3/7	0	4/9	0	1/2	0	1/2	3/8	4/9	1/2	1/3	1/2	0	1/2	0	1/2	0	4/9	0
Gini Index	0.42	0.40		0.38		0.34		0.42		0.40		0.30		0.34		0.38		0.40	
Information Gain		0.02		0.05		0.08		0.00		0.02		0.12		0.08		0.05		0.02	
Final Criteria for Splitting		<=97.5										>97.5							

Table (7): Borrowers dataset with information gain calculations for possible splits for the Annual Income feature. The datasheet with all the formulas is available as an Excel file here.

Figure(19) shows the advantage of a test condition for a continuous attributes, the branching of the tree is much simpler and will lead to a more elegant and less cultured and easy to interpret tree.

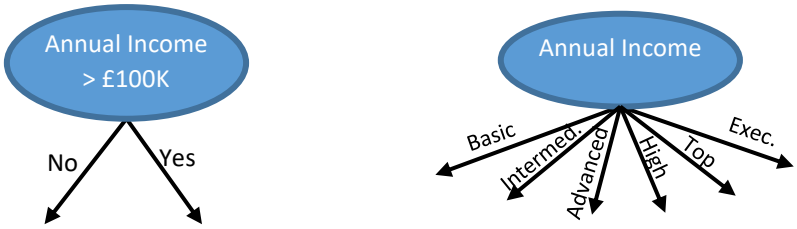


Figure 2.123.: Test condition for a continuous attribute

DT Video2 for presentation part2

2.5 Other Types of Impurity Measurements

There are other impurity measures such as the entropy or the misclassification error. Figure 2.13 below shows the behaviour of these three impurity measures as per two probabilities of two classes. We only show one probability on the x axis because the other is just the complement of p, i.e. 1-p. As you can see, when both probabilities of the two classes are close to 0.5 the impurity is maximal. When either is close to the 1 (the other 1-p would be close to 0) the impurity is minimised. The figure shows that the max of the Gini and misclassification error is 0.5 while the max for the entropy is 1.

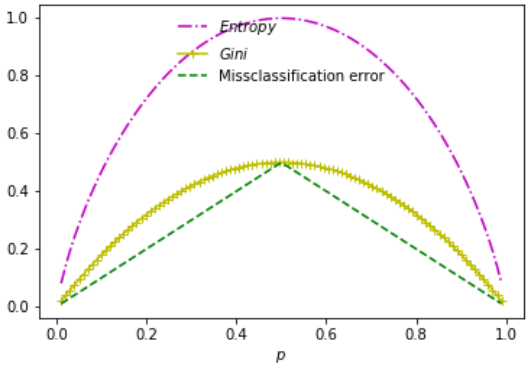


Figure4.2.13: Comparison of different impurity measures

In fact, these are all valid and you can use any. Albeit an important element of decision tree induction which must be used, changing the impurity measure between these three measures has a limited effect on the tree structure. In fact, although they vary in range, they produce consistent decision trees. What matters more in the context of decision trees is the use of

pruning and pre-pruning. Therefore we will only briefly discuss them here, but you should try to familiarise yourself with these other types of impurity measures from the Tan et al (2020), particularly entropy which has applications in a wide range of disciplines.

The entropy is a measure of chaos in a system. It is also used as a measure of information- in fact, information theory depends heavily on it. In this lesson however, we will concentrate on it as a measure of chaos or surprise. If the set of events or items have a probability peak, i.e. a subset of those items have high probability, then the system is less chaotic and the entropy is small. On the other hand, if the events or items have similar probabilities, without a clear winner, then the system is harder to predict and its chaos or entropy is maximal.

This is reflected in figure 20 above, where we can see that when the two classes have a probability of 0.5 (remember if p=0.5 then 1-p=0.5) then the entropy is maximal =1. It fades away when one of the classes has high probability and the higher the probability the lower the entropy until it reaches 0, when the probability of either classes is 1 (same for one of the classes probability is close to 0 the other would be close to 1). There is always symmetry in all of those impurity when dealing with a binary class problem, but when it is multi-class this is not guaranteed. Below we contrast Gini and entropy to gain understanding of both.

For Class 1 with probability p, we want to make sure that:

1- When the *probability* p is low, the $Entropy$ is low. Hence, we simply include p in $Entropy$ formula.
At the same time
2- When the *probability* p is high, the $Entropy$ is low. Hence, we include the term $-\log p$ in the $Entropy$ formula.
Note that $\log p \leq 0$ because $p \leq 1$. Hence $-\log p \geq 0$.
Note that $-\log p$ is monotonically decreasing function.
Note also that the base of $\log$ is normally 2 but any can do as long as we are consistent.
The behaviour of $-\log p$ for class C1 and $-\log(p')$ for class C2 can be seen below. When the probability increases $-\log p$ decreases but it is still positive (to be precise it is non-negative). Note that $-\log(p')$ is monotonically increasing function with respect to p and is non-negative as well.

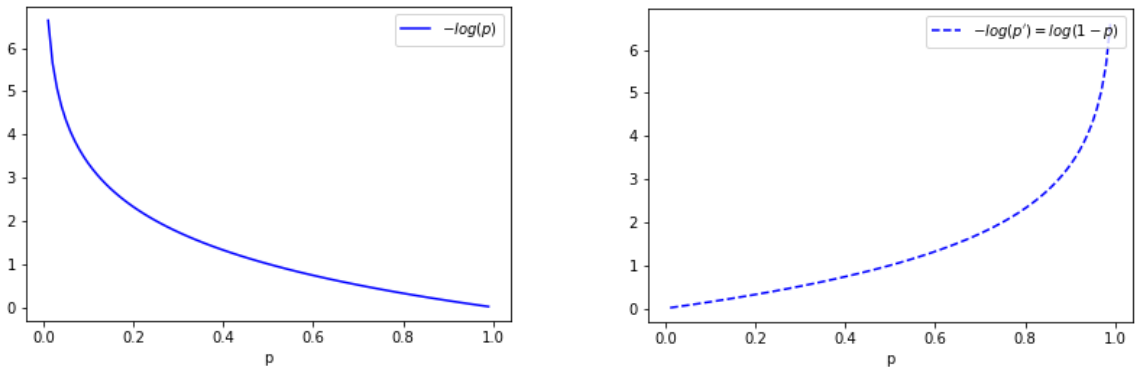


Figure 2.145.: Behaviour of the term $-p \log p$ which is the entropy for class C1. Note that C1 has a probability p and the figure shows how the entropy of C1 is varying with the probability p.

To take into account both of the points above, the entropy for class C1 will be written as $-p\log p$, which has a behaviour that is described in the left hand side of figure 22 below. In addition, since we have two classes then we also need a similar term for the second class C2. Given that C2 has a probability $p' = 1 - p$, its entropy is $(1-p)\log(1-p)$. The behaviour of this term is shown in the right hand side of the figure below.

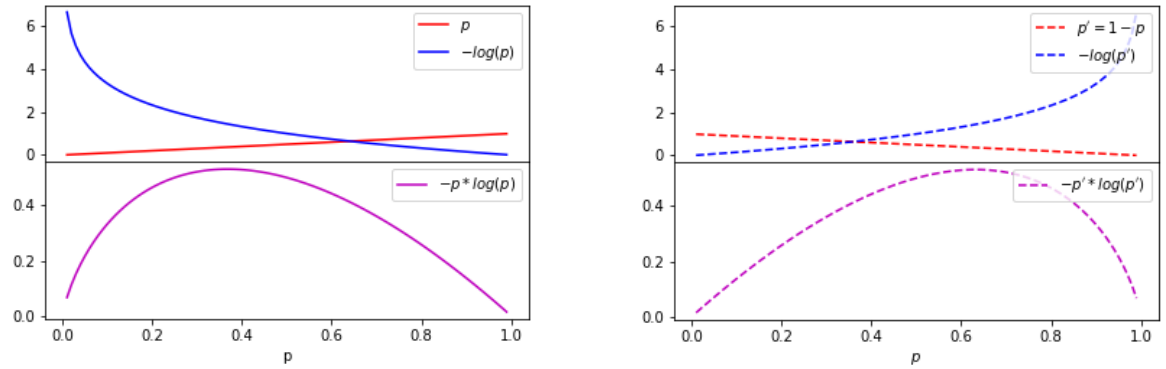


Figure 2.15 6.: Left: The entropy for class C1= $-p\log p$. C1 has a probability p, the figure shows how the entropy of C1 varies with the probability p. **Right**: The entropy for class C2= $-p'\log p'$. C2 has a probability $p' = 1 - p$, the figure shows how the entropy of C2 varies with the probability p.

We can finally define the entropy as

$$Entropy = -p\log p - p'\log p'$$

$$Entropy = -p\log p - (1-p)\log(1-p)$$

Its behaviour is shown figure 23, below

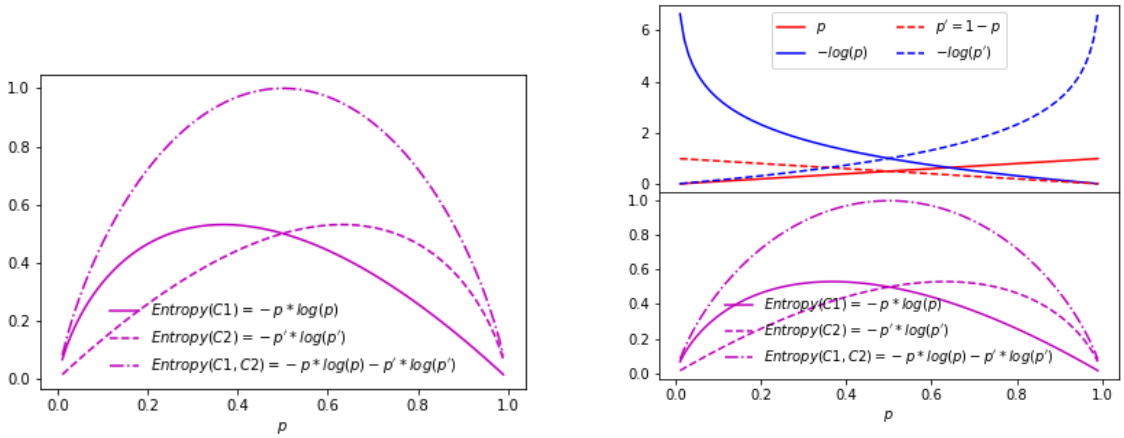


Figure 2.167.: Left: The entropy of both classes C1 and C2 who have probabilities p and $p' = 1 - p$, respectively. Right: The different components of the entropy fit together.

Note that we are talking about two classes (events) not two probability distributions. In the case of two probability distributions we use cross-entropy which is outside the scope of this discussion. In general if we have more than K classes, then

$$Entropy = \sum_{i=1}^{K} p_i \log p_i$$

2.5.1 Comparison of the Entropy with Gini index

The same ideas applies for the Gini index, but it is less complex

1- When the *probability* p is low, the $Gini$ is low. Hence, we simply include p in $Gini$ formula.
At the same time
2- When the *probability* p is high, the $Gini$ is low. Hence, we include the term $1-p$ in the $Gini$ formula.
To take into account both of the points above, the Gini index should include the term $p(1-p)$. Its behaviour is shown in figure 24 below.

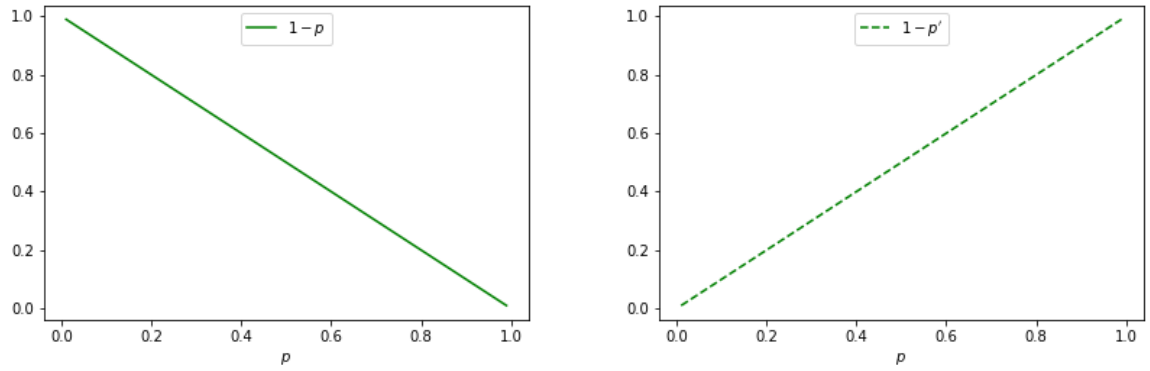


Figure 2.178.:The behaviour of the term $(1-p)$ with respect to class C1 which has probability p.

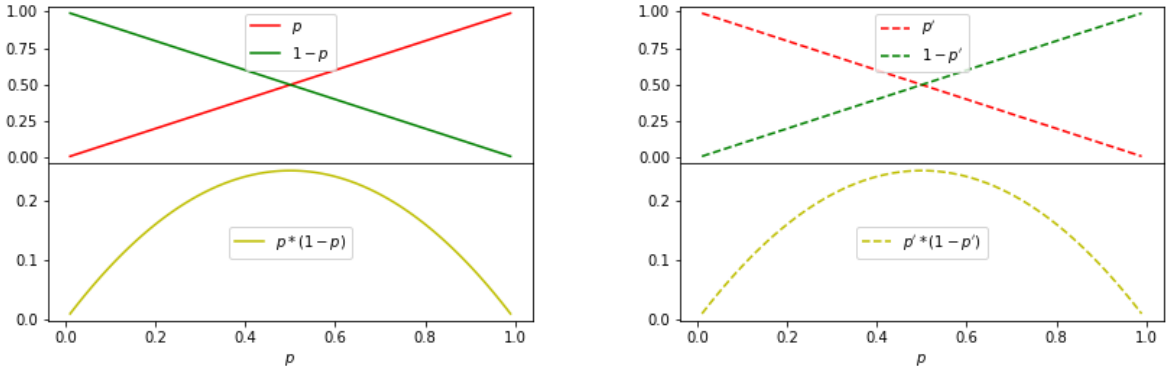


Figure 2.189.: left: The Gini impurity for class C1 $= p(1-p)$, C1 has probability p. Note that the term $1-p$ replaces the $-\log p$ in the entropy and it is easier to calculate.

Note that $1-p$ happens to be the probability of class C2 but it is not what is meant her, this becomes clearer when we consider a multi-class situation where the term $(1-p)$ is still used to calculate the impurity of C1 but the probability of C2 is likely to be different due to the involvement of other classes. This coincidence makes the left and right hand sides identical for the binary classes problems. Note that the term has a max of 0.5*0.5=0.25.

In addition, since we have two classes then we need also similar term for the second class. Given that its probability is p'

$$Gini = p(1-p) + p'(1-p')$$

In the case of Gini impurity it is helpful to realise that $p + p' = 1$ hence

$$Gini = p(1-p) + p'(1-p') = (p+p') - \left(p^2 + p'^2\right) = 1 - \left(p^2 + p'^2\right)$$

Its behaviour is shown the figure below

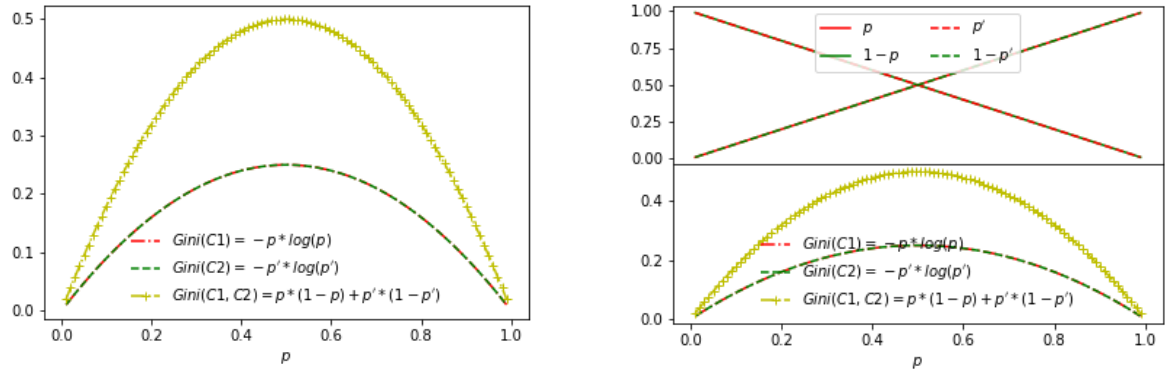


Figure (10.): left: The Gini impurity for two classes C1 and C2 with probabilities p and $p' = 1 - p$ respectively. Note that the Gini impurity has a max of 0.25+0.25=0.5. Right: The different components of the Gini impurity fit together.

In general if we have more than K classes, then

$$Gini = \sum_{i=1}^{K} p_i(1 - p_i) = 1 - \sum_{i=1}^{K} p_i^2$$

Figure 2.20 below summarises all of the terms included in both the entropy and Gini and as we have said earlier, both produce consistent trees and have a similar behaviour albeit having different ranges.

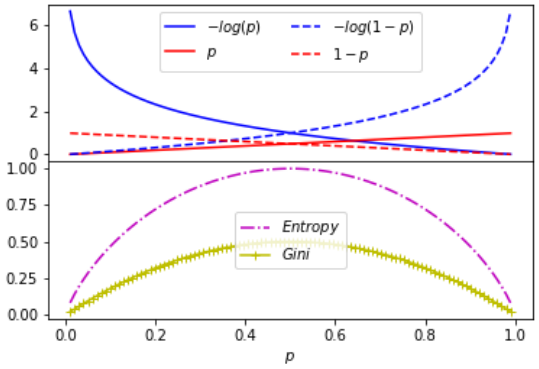


Figure 2.2011.:The behaviour of the entropy Gini with respect to both class 1 which has probability p and class 2 which has probability $1 - p$.

Note that the colours are representative of the terms involved in the calculation of both measure. The Gini is represented as red since on p and 1-p are involved in its calculations, while the Entropy is represented as magenta since all the four terms in blue and red are involved in its calculations (red + blue=magenta).

Finally the classification error is give as

$$Classification\ error = 1 - \max(p_i)$$

The behaviour of all of the three impurity measures have been already shown in Figure 2.13

2.5.2 Exercise:

See the following Jupyter notebook that implement and visualise the above impurity metrics.

DT Video3 for presentation part3

3 DT TRAINING ON A VERTEBRATE DATASET

Now let's take a look at an example of a simple classification task. Let us assume that we have the following data:

Name	Gives Birth	Warm-blooded	Aquatic Creature	Aerial Creature	Has Legs	Hibernates	Animal Class
human	1	1	0	0	1	0	mammals
python	0	0	0	0	0	1	reptiles
salmon	0	0	1	0	0	0	fishes
whale	1	1	1	0	0	0	mammals
frog	0	0	1	0	1	1	amphibians
komodo	0	0	0	0	1	0	reptiles
bat	1	1	0	1	1	1	mammals
pigeon	0	1	0	1	1	0	birds
cat	1	1	0	0	1	0	mammals
leopard shark	1	0	1	0	0	0	fishes
turtle	0	0	1	0	1	0	reptiles
penguin	0	1	1	0	1	0	birds
porcupine	1	1	0	0	1	1	mammals
eel	0	0	1	0	0	0	fishes
salamander	0	0	1	0	1	1	amphibians

Table (2): Animals class dataset

The data is taken from Tan et at (2020). This data shows the attributes (features) of different vertebrates, the attribute Class is the label and it shows the classification of the vertebrate, there is 5 classes {mammals, reptiles, fishes, amphibians or birds}. We have 6 features (excluding the label) these are {Warm-blooded, Gives Birth, Aquatic Creature, Aerial Creature, Has Legs and Hibernates}. Our dataset consists of 16 records each with its values of 0 or 1. So our features are all binary (true/false or yes/no) hence they are categorical (but these can be treated as numerical if necessary). Our task is to build a decision tree classifier that is able to model the relationship of the vertebrate and its class. These relationship are studies in biology, so we want to teach our tree a biological lesson and we want to tell us later in the future if we give a vertebrate whether it is a mammal, reptile etc.

Exercise: Think of ways to build a decision tree similar to what we had earlier and that is able to infer the class of a given vertebrate.

Step 1: What do you expect the decision tree will look like? Try to come up with a DT based on your own analysis of the dataset.

Step 2: Make an attempt to create a DT as per CART algorithm for the given dataset. Don't worry about doing this perfectly at this stage, as we will go through this together later.

Step 3: compare your outcomes from step 1 and 2. Reflect on what you have done differently from the CART algorithm

Perhaps you realised from the previous exercise that it is possible to solve the problem just using a priori domain knowledge. If we already know how to tell the difference between mammals and non-mammals, it may seem like a waste of time to involve a decision tree in the process. But if we approach this simple-seeming problem as we would a more complex one, we lay the groundwork for solving much more advanced problems. In fact, we will see later that we can automate the process by creating decision tree learning algorithms that are capable of structuring a tree to be used for inferencing

You will solve the full classification problem involving all of the vertebrate classes (mammals, reptiles, fish, amphibians etc) at the end of the section. First however, we will simplify this dataset to make the problem a binary classification task, the two classes being {mammal, non-mammal}, so our new dataset is as follows:

Name	Gives Birth	Warm-blooded	Aquatic Creature	Aerial Creature	Has Legs	Hibernates	Mammality Class
human	1	1	0	0	1	0	mammal
python	0	0	0	0	0	1	non-mammal
salmon	0	0	1	0	0	0	non-mammal
whale	1	1	1	0	0	0	mammal
frog	0	0	1	0	1	1	non-mammal
komodo	0	0	0	0	1	0	non-mammal
bat	1	1	0	1	1	1	mammal
pigeon	0	1	0	1	1	0	non-mammal
cat	1	1	0	0	1	0	mammal
leopard shark	1	0	1	0	0	0	non-mammal
turtle	0	0	1	0	1	0	non-mammal
penguin	0	1	1	0	1	0	non-mammal
porcupine	1	1	0	0	1	1	mammal
eel	0	0	1	0	0	0	non-mammal
salamander	0	0	1	0	1	1	non-mammal

Table (3): Mammals class dataset

Name	Gives Birth	Warm-blooded	Aquatic Creature	Aerial Creature	Has Legs	Hibernates	Mammality Class Binary
human	1	1	0	0	1	0	1
python	0	0	0	0	0	1	0
salmon	0	0	1	0	0	0	0
whale	1	1	1	0	0	0	1
frog	0	0	1	0	1	1	0
komodo	0	0	0	0	1	0	0
bat	1	1	0	1	1	1	1
pigeon	0	1	0	1	1	0	0
cat	1	1	0	0	1	0	1
leopard shark	1	0	1	0	0	0	0
turtle	0	0	1	0	1	0	0
penguin	0	1	1	0	1	0	0
porcupine	1	1	0	0	1	1	1
eel	0	0	1	0	0	0	0
salamander	0	0	1	0	1	1	0

Table (4): Binary class mammalians dataset

Note that we can represent the new 'Mammalian Class' using {0, 1} or {yes, no} .

For the purposes of this lesson, we are stating explicitly what the class is so that these instructions don't seem confusing. However normally, we would just use {0, 1}. If the dataset is large, it is almost always the case that we use {0, 1} as a compact way of describing binary features and binary labels. For the output to be expressive we can convert the {0, 1} values when we need to visualise the data or display a model classification results into more expressive representations. Our mission is now to build a decision tree that **automatically** learns how to classify a vertebrate into a mammal or non-mammal classes. We will not tell the tree what the relationship is between the features and the label, we want it to learn by itself. To show how the tree learns this dataset we will show the steps that a decision tree learning algorithm will do to build the tree.

Let us start by utilising the attribute 'Gives Birth' to split the data. Using RapidMiner we can build a quick model that helps us to do so. You will watch a video at the end of this section on how to easily build a decision tree model in RapidMiner, and you can do the same exercise in a Python Jupyter notebook. But first let us see the resultant tree when we split by 'Gives Birth':

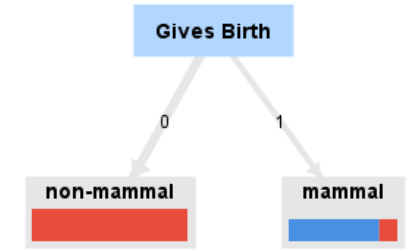


Figure (12.) Decision Tree Graph for mammalian dataset with the data is split according to 'Gives Birth' features, the graph is exported from RapidMiner.

This figure exported from RapidMiner combines the tree structure into the data that is distributed between the nodes. In RapidMiner the condition is represented as a rectangle without a colour bar while the leaves are with colour bars. The colour represents the class, red is 'non-mammal' and blue is 'mammal'. The thickness of the colour represents the data percentage, in our dataset we have $10/15 \approx 67\%$ of the animals are 'non-mammals' and $5/15 \approx 33\%$ are 'mammals'. Here in the above decision tree graph we have 60% of the data in the left and 40% of the data in the right node. The left hand side node is pure red ('non-mammals') and the right hand side is mainly blue ('mammals') with some red.

The figure shows that if we split according to the 'Gives Birth' feature **only**, setting aside all other features, then this split gives us a pure leaf for the left hand side branch of the tree. This means that the learning algorithm discovered that **all** 100% of animals that do not 'Gives Birth' are definitely 'non-mammals' (note that in the dataset 60% of the animals that are 'non-mammals' and do not 'Gives Birth'). It also shows that we have a mixture of mammals and non-mammals in the right hand leaf node with the majority being mammals. This means that we can further split the right hand side node using another feature.

Ok so now let us move to the next feature and we continue to split our data in our tree with feature 'Warm-blooded'. If we do so we get the following graph:

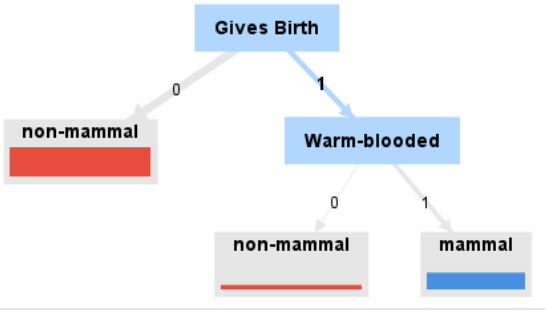


Figure (13.) Decision Tree Graph for mammalian dataset with two features splits ('Gives Birth' and 'Warm-blooded'), exported from RapidMiner.

This shows that we can actually decide on whether an animal is a mammal using only the two features shown above, and all other features are not needed to do so. We can discard all other features and just use the 'Gives Birth' and 'Warm-blooded' features. Later we will see that we cannot do so if we want to know finer classification for the animal such as 'reptile' or 'fish'.

Name	Gives Birth	Warm-blooded	Mammality Class
human	1	1	mammal
python	0	0	non-mammal
salmon	0	0	non-mammal
whale	1	1	mammal
frog	0	0	non-mammal
komodo	0	0	non-mammal
bat	1	1	mammal

pigeon	0	1	non-mammal
cat	1	1	mammal
leopard shark	1	0	non-mammal
turtle	0	0	non-mammal
penguin	0	1	non-mammal
porcupine	1	1	mammal
eel	0	0	non-mammal
salamander	0	0	non-mammal

Table (5): Mammals class dataset with necessary and sufficient features.

3.1.1 Exercise

Please watch this video to see how we can easily build a decision tree model in RapidMiner, the resultant model can be imported from here.

3.1.2 Exercise

Pick another feature to start with, and try the process again. What happens?. Hint: compare between the information gain when we start the split by 'Gives Birth' and when we start the split by 'Warm-blooded'.

3.1.3 Exercise

Build a decision tree for the original dataset including all of the original features. This is a good opportunity to try using RapidMiner if you want to.

3.1.4 Exercise

Take away all the features except for 'Gives Birth' and 'Warm Blooded' from the original dataset. Now attempt to build a decision tree from this new dataset. Are the leaves pure? Can you estimate the accuracy of the decisions that will be made by this tree?

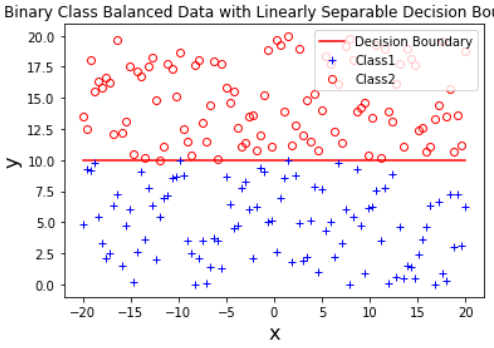
3.2 Validity of Decision Trees

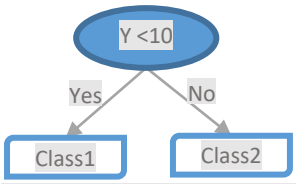
Complex structures that are called decision trees may not satisfy the definition of an automatic decision tree due to horizontal and cyclic paths. See the following example of a DT for risk assessment by one of the NHS groups in the north of England, which it violates the definition in one horizontal connection. For a more complex example of more violation of the automated DT see this example for Cattle testing. It should be noted that it is possible to reproduce these structures to be a valid DT or to validate DT based on a raw dataset, see this article for example.

3.3 Decision Boundaries of Decision Trees and the Limitations of Decision Trees

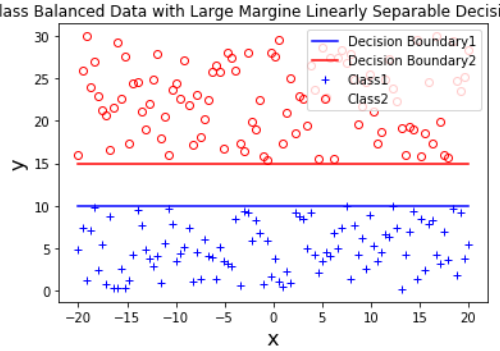
So far we have seen how decision trees work and how they are inducted (trained). Later we will see how to measure their performance. But first we would like to study further their inner properties. In particular we would like to see what type of decision boundaries they constitute. The idea of decision boundaries is central to classification and not unique to decision trees. It will reappear in other types of classification techniques we will study later, such as the perceptron and k-nearest neighbour (k-NN) algorithms. One way to understand the decision boundaries of a classifier is by plotting a dataset in 2d or 3d, note however that the discussion extends to any space dimension not just 2d but it would be harder to visualise it. Let us start with a simple decision tree with its decision boundary.

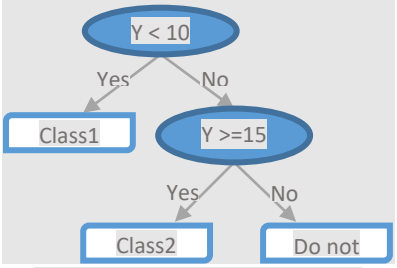
Binary Class Balanced Data with Linearly Separable Decision Boundary



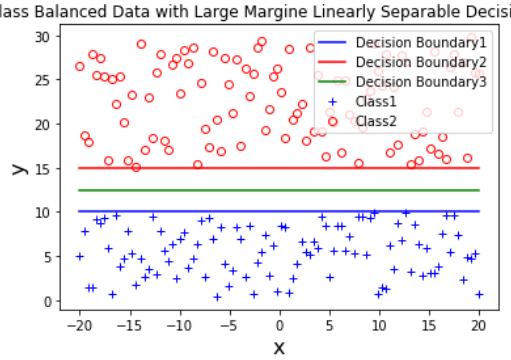


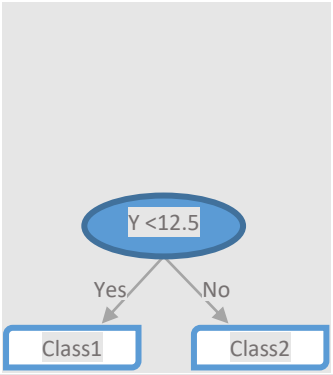
Binary Class Balanced Data with Large Margine Linearly Separable Decision Boundaries



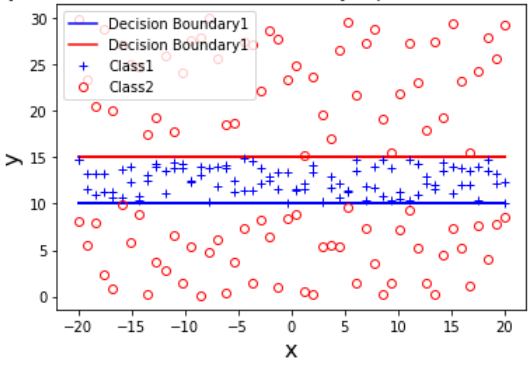


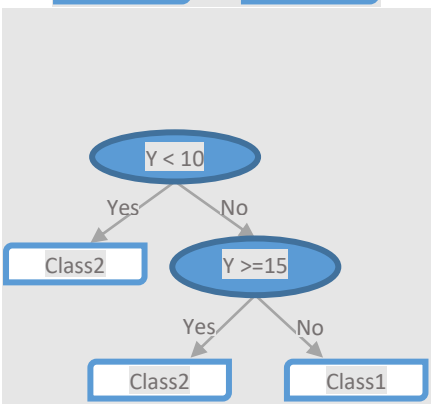
Binary Class Balanced Data with Large Margine Linearly Separable Decision Boundaries





Binary Class Balanced Data with non-Linearly Separable Decision Boundaries





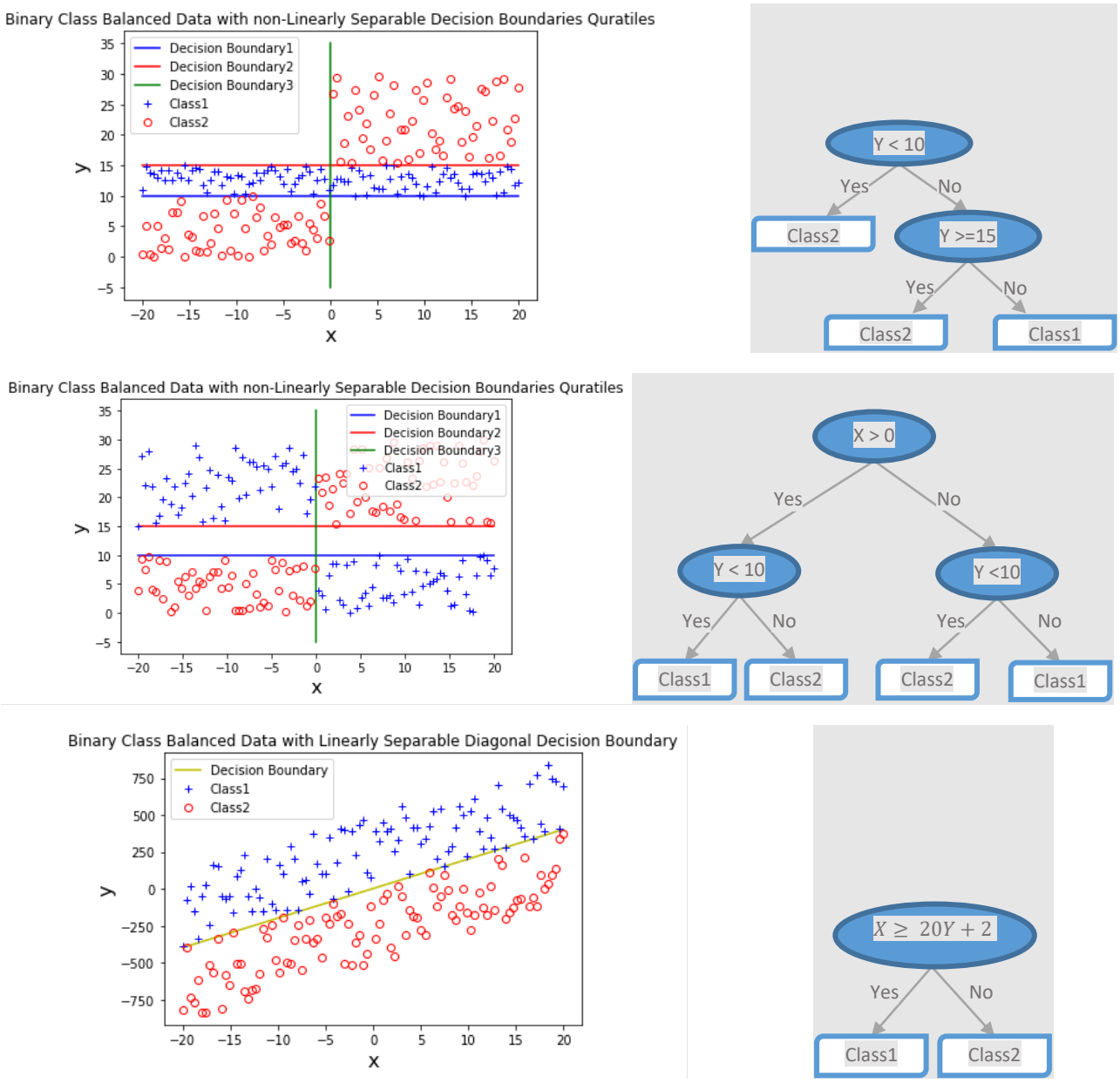
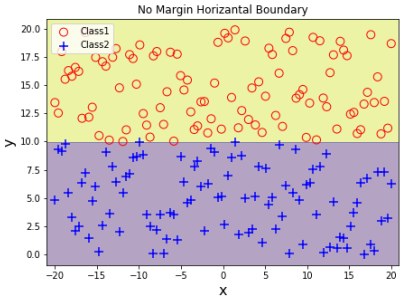


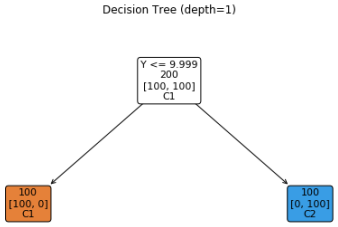
Figure (14.) (left) Linearly and non-linearly separable classes data with their virtual decision boundaries. (Right) expected corresponding decision trees.

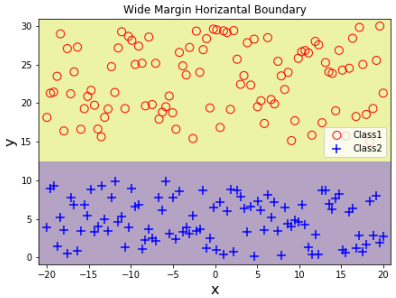
In the above cases, we showed the data and the decision boundary on the left-hand side and on the right-hand side we showed the corresponding expected decision trees that can express the data concisely. In classification such decision boundaries are crucial in two ways: they help us understand the nature of the data, and they help us to assign a suitable technique to the problem in hand. It is not always possible to represent the data in two-dimensional space, in fact it is rarely the case. However, even when we move to higher space dimension, a similar argument can hold.

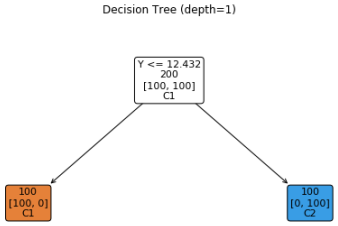
The boundaries are assumed when we constructed the datasets. The boundaries do not exist separately form the dataset, instead they are inferred from the dataset. In the decision trees, the nodes correspond to the condition as usual. These conditions that help us decide the classes of the dataset create their own boundaries. In the first case, any point above the boundary is of class 1 and any point below the boundary is from class 2. In the second case, we have a large margin decision boundaries and we can express the tree in several ways, one of them is shown. In the third case, the tree is simplified to reflect a midpoint margin. In the fourth we sandwiched class1 between two parts of class 2. The fifth a sandwiched class 1 but class 2 is separated in two different quarters. The sixth, we distribute the two classes into four crossed quarters. In the seventh, the data is distributed above and below a diagonal line.

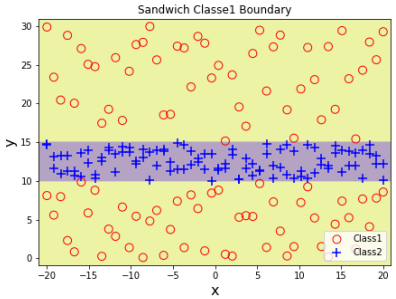
So let us see if we hand in the generated data to the CART decision tree induction algorithm, would it be able to recognise the decision boundaries of the data and would it be able to build corresponding trees as per our expectations. Below are the results.

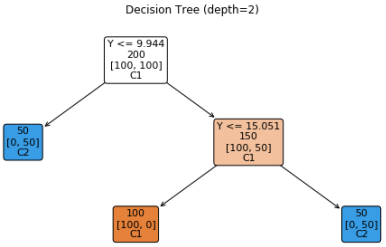


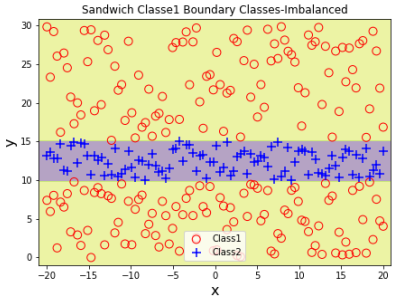


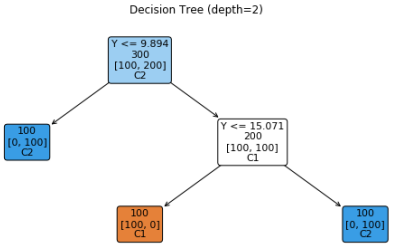












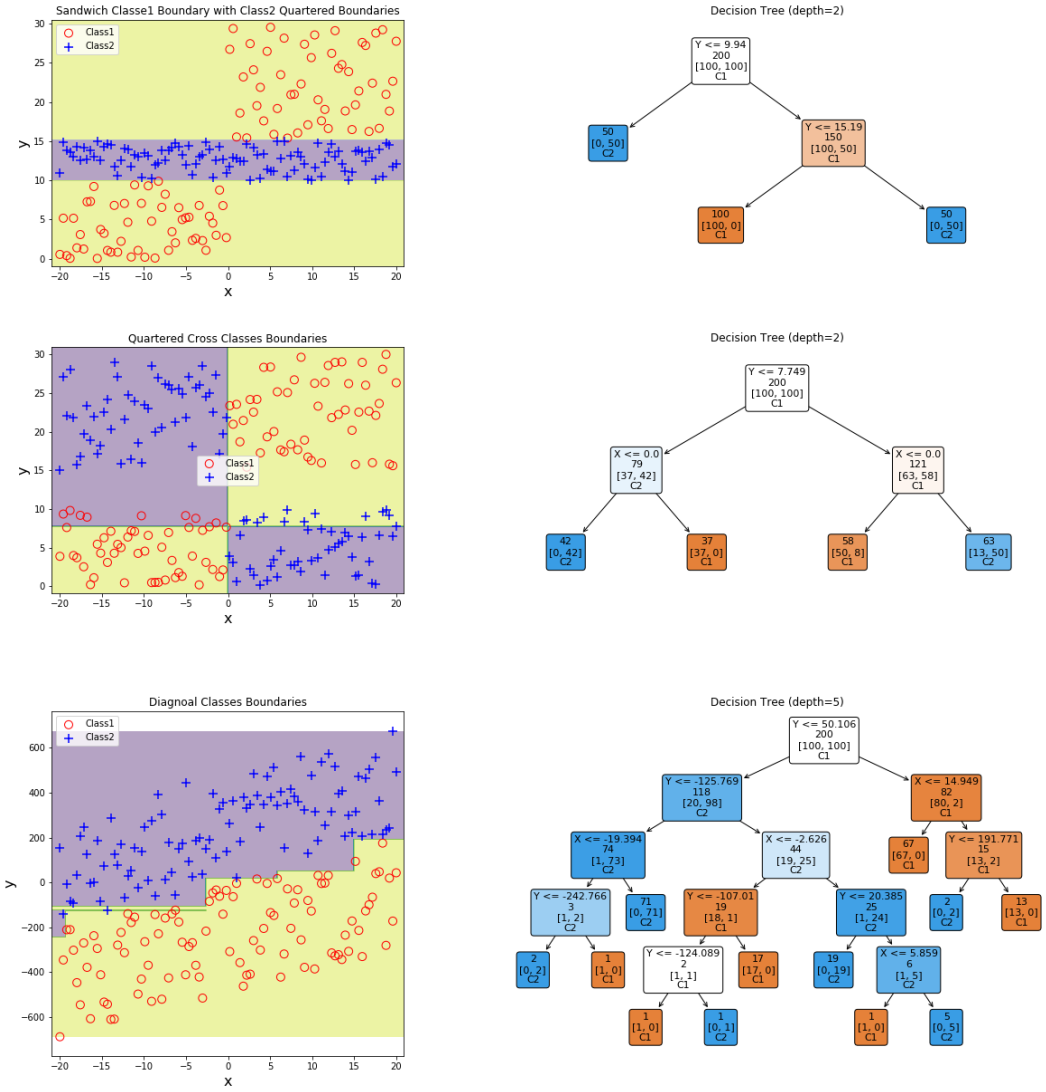


Figure (15.) (left) Linearly and non-linearly separable classes data with the decision boundaries of the corresponding decision tree which are shown to the right. (Right) actual corresponding decision trees.

Note how the algorithm struggled with the last case as the classes and its corresponding boundary becomes non-linearly separable. Linearly separable classes are those classes that we separate by just a line. Non-linearly separable classes are those that need more than one line to separate them or those that need another more complex shape to separate them, whether the shape is regular such as a circle or hyperbola or non-regular such as a convoluted curve. Here the boundaries are inferred from the decision tree itself, unlike the previous set of figures where the boundaries were assumed when we constructed the datasets. As we can see, the DT struggles the most when the data is diagonal. This is because the CART deals with one feature at a time in its conditions. Obviously, there are ways to work around this issue. The most obvious is to allow the DT to deal with two features inside its conditions. This would add to the complexity of the algorithms, and the problem becomes extenuated when we consider hundreds of features. If we are to consider all possible combinations of even 20 features this would amount to checking $2^{20} = 1{,}048{,}576$ combinations. If each one has 10 possible values we are talking about $10^{20} = 100{,}000{,}000{,}000{,}000{,}000{,}000$ which is clearly problematic. So the DT might not be the best in dealing with these cases. In fact, it is not great at dealing with numerical data in general. Please note that the decision boundary idea is quite powerful and we will utilise it in other techniques more centrally. These better suited techniques include the perceptron, multi-layer perceptron and nearest neighbours classifiers.

3.3.1 Exercise

In the following Jupyter notebook exercise you will be able to visualise the decision boundaries of a decision trees.

3.4 Lesson Summary

In this lesson we have covered decision trees, an important and pervasive technique for classification. We have looked at the CART algorithm and seen examples of its inner mechanism. We have also discussed limitations and decision boundaries of DT. DT is quite a powerful technique in terms of interpretation and can provide an excellent tool to convey and explain decisions made by it. However it employs a local search strategy when it comes to growing its branches and it has rectilinear decision boundaries. Although there are ways to mitigate these limitations, DT might not be the best technique when we deal with numerical data because it is discretised by nature.

4 Measuring the Performance of a Classification Model (2)

Learning outcomes:

After completing this lesson you should be able to:

- understand the essential metrics needed to evaluate different classification models
- distinguish between metrics that are based on actual classes, and metrics that are concerned with predicted classes.
- understand the holistic metrics for classification

In this section we will study how we can measure how accurate our classification model is. The concepts and ideas will be applied to a decision tree because this is the only classification technique we have covered so far, but the concepts and ideas are equally applicable regardless of the technique. Indeed, we will be utilising these metrics in later sections and units directly without explanation.

Measuring the effectiveness of a model is an essential skill, since we will often face problems that we can solve in several techniques. To objectively choose among them we need to compare their respective predictive models' capabilities and pick the one that suits best our problem. There is also the issue of picking the right **metric** for the problem in hand. Therefore, we will cover several metrics and we will learn which suits a specific problem, both in terms of the dataset structure and in terms of the aim of the analysis that we intend to perform in order to solve the problem in hand.

We will start by classification of a binary class problem, and we will generalise the measures for multi-class problems. However, please bear in mind that some metrics do not directly extend to a multi-class problem, and we will point this out whenever it is the case. But before seeing these metrics, we need to talk about why in the first place we might have imperfect performance.

4.1 Measuring the performance of a Binary-Class Model

For a binary class problem, we have two classes that an instant is either belongs to class C1 or to class C2 but cannot belong to both at the same time. In fact, the question can be posed as whether an instant belongs to a one class of concern or not. The classification would be effectively stating yes or 1 or + if the instant belongs to the main class of concern or stating no or 0 or – if the instant does not belong to the main class (which implicitly means it belongs to the complement of the class). We just have to be consistent in our approach. So in this context it is a binary choice, and it is useful to represent one of the classes as positive + and the other as negative –. Now, our classifier (our DT) mission is to predict whether an instant is of class + or class –. Therefore, we contrast what the classifier has **predicted** and what was the **actual class** of all instances to measure how good our classifier is. This can be done for the training set, the validation set or the testing set, all of which we have answers for (i.e. we know the classes for these sets). When we contrast the prediction against the actual class of each instant we have four possibilities:

1- The actual class is + and the predicted class is + (True Positive-TP)
2- The actual class is – while the predicted class is + (False Positive-FP)

3- The actual class is + while the predicted class is – (False Negative-FN)
4- The actual class is – and the predicted class is – (True Negative-TN)

These 4 cases can be better summarised in the figure below. This is called the confusion matrix because the red boxes represent the cases confused by the prediction model, while the black boxes represent the cases where the predictions of the model are aligned with the reality. The aim of any model is to reduce the cases in the red boxes and make them as close to 0 as possible. We can actually do a lot with these counts, and below we show several metrics that can be defined based on them.

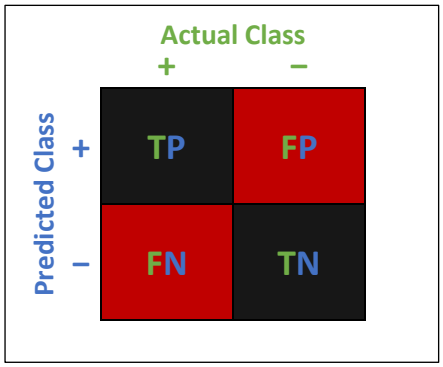


Figure 4.1: Confusion matrix for binary classification model

Be mindful that some sources present the confusion matrix using the transpose of the above matrix (with actual classes on the left and the predicted classes on top) as in the figure below. This will not change anything but the presentation. RapidMiner and Weka for example use the first form, while Tan et al (2020) use the second form. We will adopt the first form as it is more common. Note that FP is called **type I error,** while FN is called **type II error**, accordingly it makes more sense to present the matrix in the first form.

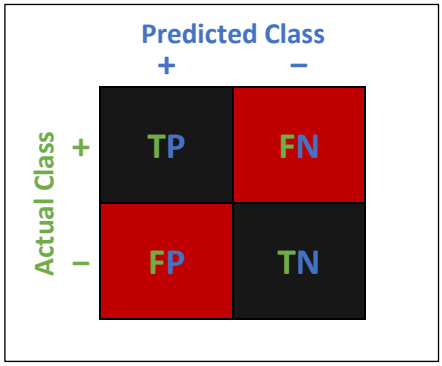


Figure 4.2: Confusion matrix for binary classification model presented differently.

Note that the total number of instances is the sum of all of the numbers in the boxes of the confusion matrix:

$$N = TP + TN + FP + FN$$

this is regardless of the distribution of the correctly and incorrectly classified instances.

We define the Accuracy of a classifier as the rate of correctly classified instances out of the total number of instances:

$$Accuracy = (TP \quad TN)/N$$

On the other hand, we define the Error rate as the rate of the incorrectly classified instances out of the total number of instances:

$$\boldsymbol{Error\ rate} = (\mathbf{FP} + \mathbf{FN})/\boldsymbol{N}$$

All classifiers try to increase its accuracy or equivalently reduce its error rate. However, in some special cases these metrics do not reflect how good the model is. For example, if the classes are not balanced, the accuracy can be misleading. We will study these cases and more suitable measures for them in unit 4.

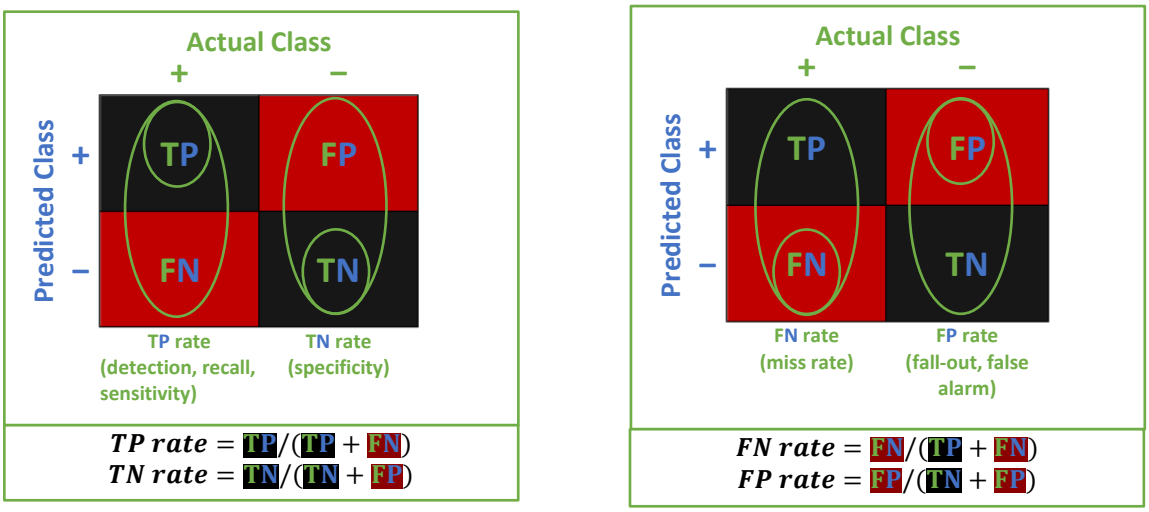


Figure 4.3: Metrics that are related to actual classes (common names used). Left: positive actual classes related metrics. Right: negative actual classes related metrics.

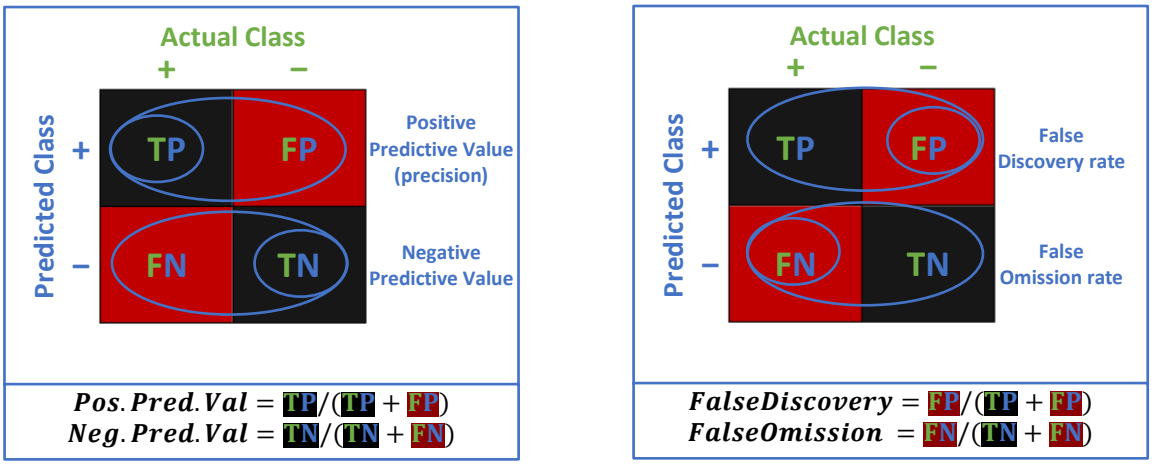


Figure 4.4: Metrics that are related to predicted classes (common names used). Left: positive predicted classes' related metrics. Right: negative predicted classes' related metrics.

As it can be seen, all possible ways of taking the rate of either of the four values in the confusion matrix is covered and has its own properties. Of particular interest is the **precision** and the **recall** (aka positive prediction value and true positive rate, respectively). When we talk about precision, we are referring to the precision of the **prediction** of our classifier with respect to the positive class. The recall, on the other hand, measures how good our classifier in detecting the positive **actual** cases is.

If we are less concerned with false positives then we can use the hit rate (aka recall) to choose between two models. If we are diagnosing cancer for example, then misdiagnosing people as having cancer is preferred over missing those who actually have cancer (given that there would be further checking to confirm the positive cases). In this case if we have to choose between two models with the same accuracy, but with different hit rates then we choose the one with the higher hit rate. We note however that these names are a bit confusing and some simplification and uniformity is needed in order to reveal their intrinsic properties and the relationship between each other. Therefore, we propose to rename them for consistency and uniformity as in the below figures. Note that these are our own naming and some coincide with the metrics names in the literature and some do not.

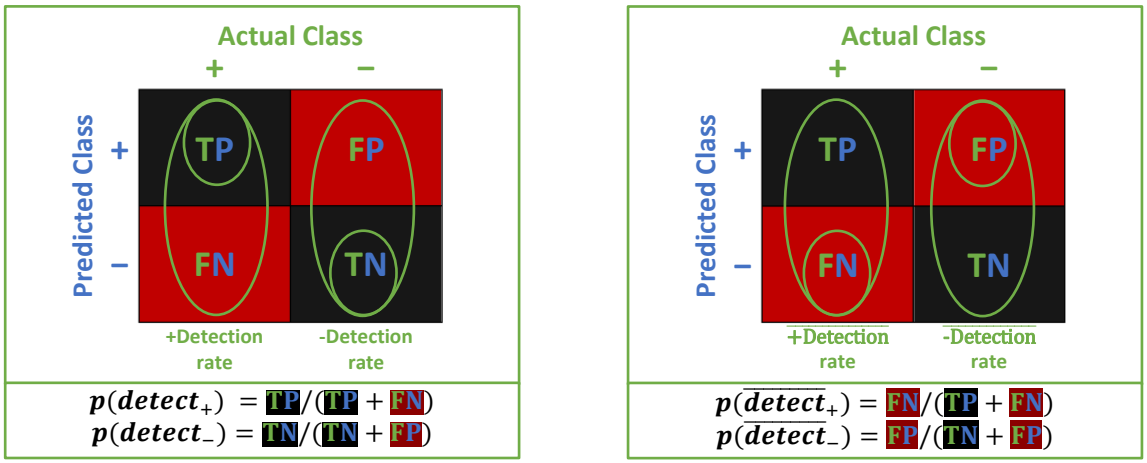


Figure 4.5: Detective Metrics related to actual classes (new suggested names used for consistency. Left: performance metrics. Right: error metrics.

The bar on top represents a complement of an event in a probabilistic sense. The true positive detection rate is denoted as $p(detect_+)$ and it represents the probability of correctly detecting positive instances by the model. Similarly, the true negative detection rate is denoted as $p(detect_-)$. It represents the probability of correctly detecting negative instances by the model.

On the other hand, the false positive detection rate is denoted $p(\overline{detect}_+)$ and it represents the probability of incorrectly detecting positive instances by the model. While, the false negative detection rate is denoted $p(\overline{detect}_-)$ and represents the probability of incorrectly detecting negative instances by the model.

We can now easily verify that:

$$p(detect_+) + p(\overline{detect}_+) = 1$$

$$p(detect_-) + p(\overline{detect}_-) = 1$$

The names are meant to reflect the inner relationship between the different metrics. To see why, we first note that $\neg\, TP = FN, \neg\, TN = FP$, where we use $\neg$ to denote the logical not. Now, if we negate both sides of the positive detection rate equation: $\neg(Detect_+ = TP/(TP + FN))$ we get $\neg Detect_+ = FN/(FN + TP) = \overline{Detect}_+$. Similarly, if we negate both sides of the negative detection rate equation: $\neg(Detect_- = TN/(FP + TN))$ we get $\neg Detect_- = FP/(TN + FP) = \overline{Detect}_-$. In other words, $\overline{Detect}_+$ $\overline{Detect}_-$ represents the model inability to detect the positive and negative instances, respectively.

Similar argument is used for the prediction related metrics. The true positive prediction value is denoted as $p(predict_+)$ and represents the probability of correctly predicting positive instances by the model.

The true negative prediction value is denoted as $p(predict_-)$ and represents the probability of correctly predicting negative instances by the model. On the other hand, the false positive prediction value is denoted $p(\overline{predict}_+)$, it

represents the probability of incorrectly predicting positive instances by the model. While, the false negative prediction value is denoted $p(\overline{predict}_-)$, it represents the probability of incorrectly predicting negative instances by the model.

We can now easily verify that:

$$p(\boldsymbol{predict}_+) + p(\overline{\boldsymbol{predict}}_+) = 1$$

$$p(\boldsymbol{predict}_-) + p(\overline{\boldsymbol{predict}}_-) = 1$$

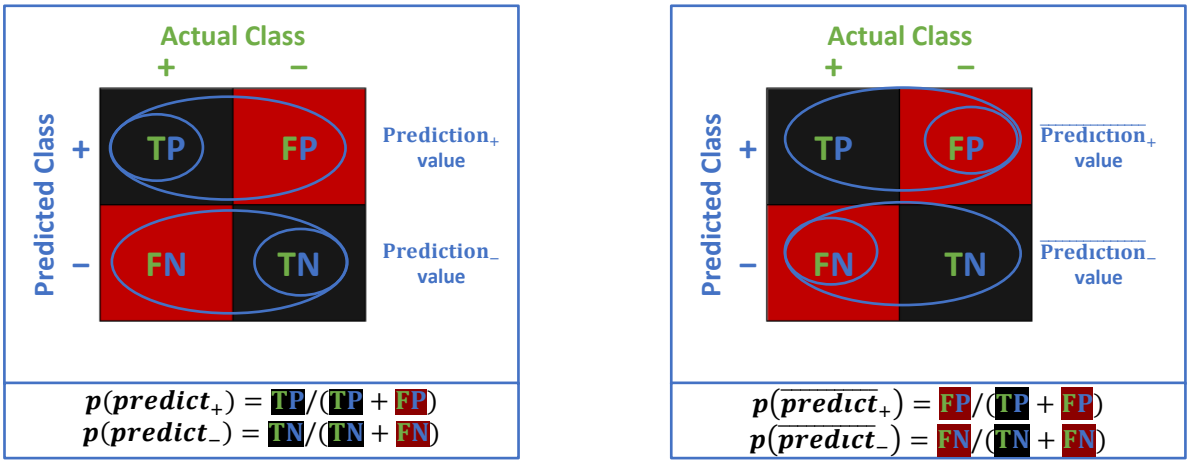


Figure 4.6: Predictive Metrics related to predicted classes (new suggested names used for consistency). **Left**: performance related predictive metrics. **Right**: error related predictive metrics.

Negation on the prediction metrics yields similar but not quite the same relationship as for the detection metrics. This is because if we negate both sides of the positive prediction value equation: $\neg(Predict_+ = TP/(TP + FP))$ we get $\neg Predict_+ = FN/(FN + TN) = \overline{Predict}_-$.

Similarly, if we negate both sides of the negative prediction value equation: $\neg(Predict_- = TN/(TN + FN))$ we get $\neg Predict_- = FP/(FP + TP) = \overline{Predict}_+$. Note that the negation here changed also the prediction metric form positive to negative.

Note that we used capital initial for the metrics to express them as a rate, while we uses small letter when we place them in the context of probabilities, so for example $\overline{Predict}_- = p(\overline{Predict}_-)$, $Predict_+ = p(predict_+)$ and so on.

4.1.1 Which Type of Metric is More Important?

Note that predictive metrics are concerned with the model ability to predict or guess the correct class of the instances, while detective metrics are concerned with the model ability to detect or recognise the correct class of the instances. In particular, detective metrics are discriminative ones by nature since we can interpret it as if we are giving the model a set of positive only (or negative only) instances without revealing the class to the model and ask the model if it can tell us which one of these are positive (or which ones are negative). The answer should be all positive (or all negative) and the more the model diverges from this answer the less discriminative it is. On the other hand, predictive metrics are speculative by nature since we can interpret them as if we are giving the model a set of mixed class instances and we ask it to come up with a guess or prediction on the class. So, predictive metrics may appear (due to its name) to pertain more for a prediction problem. However this is not the case; both the detective and predictive metrics give different insights of the quality of the classification model. The question is, can we come up with metrics that encompasses both types of measures? The answer is yes.

4.1.2 Holistic Metrics

Holistic Metrics are those metrics that look at both horizontal and vertical views and involve both positive and negative classes. These are better metrics for model comparison of most problems when we are concerned with an overall good performance without a particular preference of guessing ability or discrimination ability of the model. Among these holistic metrics are the F1 score and Matthew Correlation Coefficient. The F1 score is defined as the harmonic mean of the precision and recall. Harmonic means differs from the usual arithmetic mean. The harmonic mean for n numbers x_i is defined as the reciprocal of the arithmetic mean of the reciprocals of the given numbers: $\left(\frac{\sum_{i=1}^{k} x_i^{-1}}{k}\right)^{-1}$. Therefore, for two numbers it is defined as $\left(\frac{1/x_1 + 1/x_2}{2}\right)^{-1} = \frac{2\,x_1 x_2}{x_1 + x_2}$. Therefore, F1 score is given as

$$\boldsymbol{F1\ score} = \frac{2\,p(\boldsymbol{predict}_+)\,p(\boldsymbol{detect}_+)}{p(\boldsymbol{predict}_+) + p(\boldsymbol{detect}_+)} = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$

Note that $\boldsymbol{Accuracy} = (\mathrm{TP} \quad \mathrm{TN})/\boldsymbol{N} = (\mathrm{TP} \quad \mathrm{TN})/(\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN})$. If we compare this with the F1 score formula, we realise that we can obtain F1 score directly from the accuracy formula, by replacing the term TN with TP. In other words, we can view the F1 score from another perspective as being a measure of overall accuracy for the true positive predicted cases only (no true negative).

The Matthew correlation coefficient (MCC) is defined as

$$\boldsymbol{MCC} = \frac{\mathrm{TN}.\,\mathrm{TP} - \mathrm{FP}.\,\mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})}}$$

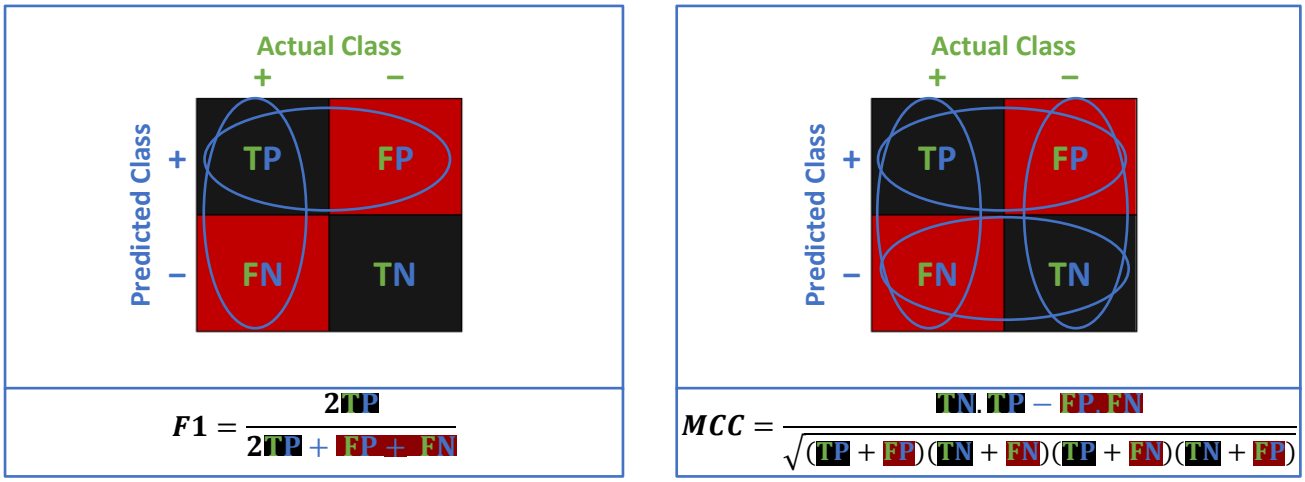
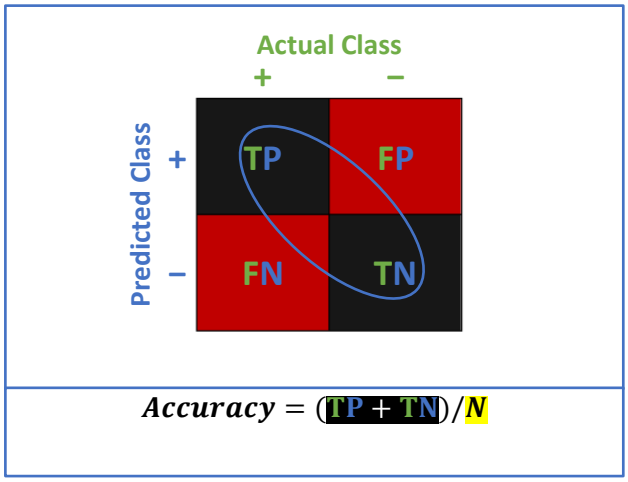


Figure 4.7: Holistic Metrics that are comprehensively involving both predicted and actual classes. Left: F1 score. Right: Matthew Correlation Coefficient. Both are suitable for any binary class problem even when the classes' counts are imbalanced. (i.e. when the number of instances form one class – normally the positive class – are much smaller than number of instances from the second class – normally the negative class).



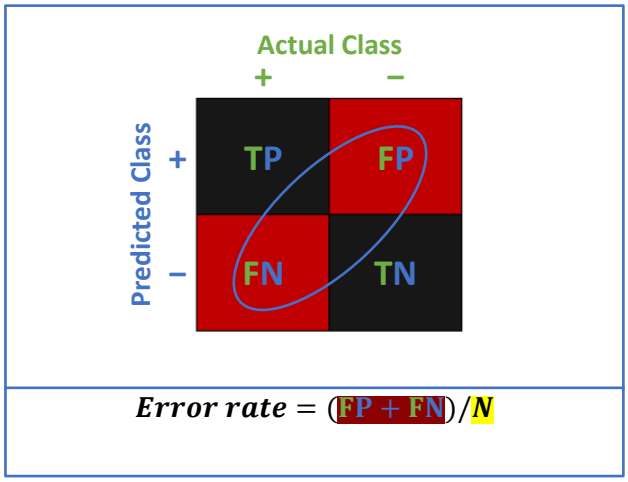


Figure 4.8: Holistic Metrics that are comprehensively involving both predicted and actual classes. Left: Accuracy. Right: Error rate. Both performs poorly when the classes count is imbalanced (i.e. when the number of instances form one class – normally the positive – are much smaller than number of instances from the second class – normally the negative class). Nevertheless, these are the basic metrics that several algorithms use.

The range of MCC is between -1 and 1. -1 represents total disagreement between the model predictions and the actual classes, 1 represents total agreement between the predicted and actual classes and 0 means no correlation, i.e. the model is not better than a random guess.

In terms of comparison, we can meaningfully compare as follows:

Model Performance Metrics		**Model Error Metrics**	
Positive Class	**Negative Class**	**Positive**	**Negative**
$p(detect_+)$	$p(detect\_)$	$p(\overline{detect_+})$=1-$p(detect_+)$	$p(\overline{detect\_})$=1-$p(detect\_)$
$p(predict_+)$	$p(predict\_)$	$p(\overline{predict_+})$=1-$p(predict_+)$	$p(\overline{predict\_})$=1-$p(predict\_)$
Holistic		**Holistic**	
Accuracy		$Error\ rate = 1 - Accuracy$	
F1 score		$\overline{F1\ score} = 1 - F1\ score$	
MCC		$\overline{MCC} = 1 - MCC$	

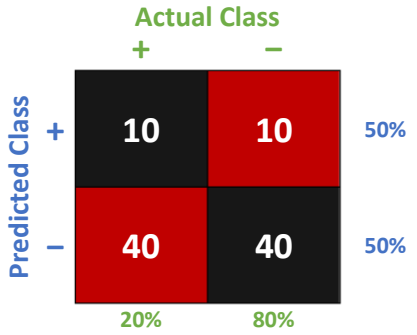
As can be seen we either compare using the left-hand side for performance or the right-hand side for errors. We do not need to use both, and we must be aware not to mingle the left with right when we compare different models' performance.

4.1.3 Examples of Metrics for a Classifier

Ok let us now have a look at some examples. Let us assume that we trained a decision tree classifier and it gave us the confusion matrix that can be seen below. We have stated all the metrics that we have covered so far and we demonstrated the calculations in a separate box underneath. Later we might just state the metrics values without the calculations.

By comparing the Accuracy (or F1 score or MCC) we accordingly prefer classifier 4.

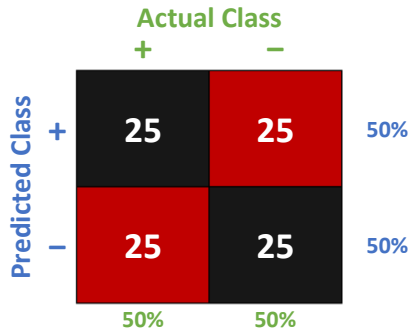
Classifier 1



n=100 Accuracy 50.0%
F1 Score 28.6%
MCC 00.0%

$$Detect_+ = \frac{10}{10+40} = 0.2 \qquad Detect_- = \frac{40}{40+10} = 0.8$$
$$Predict_+ = \frac{10}{10+10} = 0.5 \qquad Predict_- = \frac{40}{40+40} = 0.5$$
$$Accuracy = \frac{10+40}{100} = 0.5 \qquad F1 = \frac{2\times 10}{2\times 10+40+10} = 0.286$$
$$MCC = \frac{10\times 40 - 10\times 40}{\sqrt{(10+10)(10+40)(40+40)(40+10)}} = 0.0$$

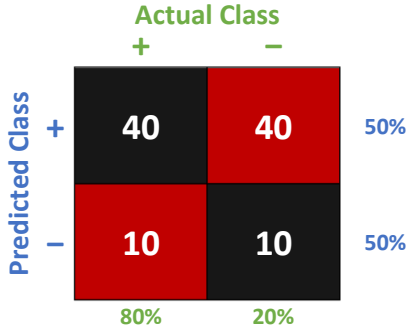
Classifier 2



n=100 Accuracy 50.0%
F1 Score 50.0%
MCC 00.0%

$$Detect_+ = \frac{25}{25+25} = 0.5 \qquad Detect_- = \frac{25}{25+25} = 0.5$$
$$Predict_+ = \frac{25}{25+25} = 0.5 \qquad Predict_- = \frac{25}{25+25} = 0.5$$
$$Accuracy = \frac{25+25}{100} = 0.5 \qquad F1 = \frac{2\times 25}{2\times 25+25+25} = 0.5$$
$$MCC = \frac{25\times 25 - 25\times 25}{\sqrt{(25+25)(25+25)(25+25)(25+25)}} = 0.0$$

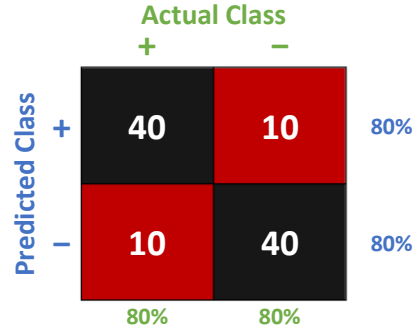
Classifier 3



n=100 Accuracy 50.0%
F1 Score 61.5%
MCC 00.0%

$$Detect_+ = \frac{40}{40+10} = 0.8 \qquad Detect_- = \frac{10}{10+40} = 0.2$$
$$Predict_+ = \frac{40}{40+40} = 0.5 \qquad Predict_- = \frac{10}{10+10} = 0.5$$
$$Accuracy = \frac{40+10}{100} = 0.5 \qquad F1 = \frac{2\times 40}{2\times 40+40+10} = 0.615$$
$$MCC = \frac{40\times 10 - 40\times 10}{\sqrt{(40+40)(40+10)(10+40)(10+10)}} = 0.0$$

Classifier 4



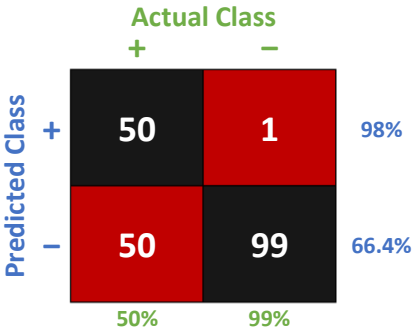
n=100 Accuracy 80.0%
F1 Score 61.5%
MCC 60.0%

$$Detect_+ = \frac{40}{40+10} = 0.8 \qquad Detect_- = \frac{40}{40+10} = 0.8$$
$$Predict_+ = \frac{40}{40+10} = 0.8 \qquad Predict_- = \frac{40}{40+10} = 0.8$$
$$Accuracy = \frac{40+40}{100} = 0.8 \qquad F1 = \frac{2\times 40}{2\times 40+40+10} = 0.615$$
$$MCC = \frac{40\times 40 - 10\times 10}{\sqrt{(40+10)(40+10)(40+10)(40+10)}} = 0.6$$

4.1.4 Examples of Metrics on Classifiers Comparison

Let us now assume that we trained a further two classifiers that produced the following two confusion matrices. Comparing the classifiers we can see how the different metrics react to the changes in the way the instances has been classified. In particular we can see that F1 score reflect a balanced estimation of the quality of the classifier, while accuracy can be quite optimistic in its estimation of the quality of the classifier. MCC is the more reserved of holsitic metrics and it tends to be more pessimistic in its estimation.

Classifier 1



Accuracy	74.5%
F1 Score	66.2%
MCC	56.2%

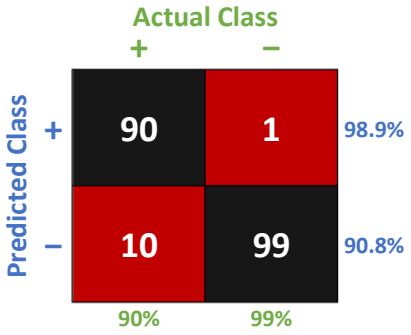
$$Detect_+ = \frac{50}{50+50} = 0.5 \qquad Detect_- = \frac{99}{99+1} = 0.99$$

$$Predict_+ = \frac{50}{50+1} = 0.98 \qquad Predict_- = \frac{99}{99+50} = 0.664$$

$$Accuracy = \frac{50+99}{200} = 0.745 \qquad F1 = \frac{2\times 50}{2\times 50+50+1} = 0.662$$

$$MCC = \frac{50\times 99 - 50\times 1}{\sqrt{(50+1)(99+50)(50+50)(99+1)}} = 0.56$$

Classifier 2



Accuracy	94.5%
F1 Score	94.2%
MCC	89.3%

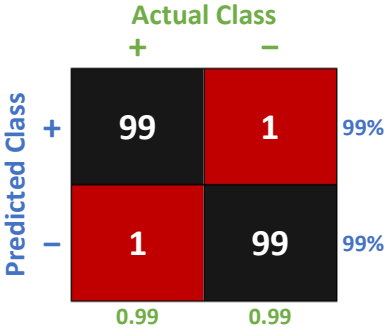
$$Detect_+ = \frac{90}{90+10} = 0.9 \qquad Detect_- = \frac{99}{99+1} = 0.99$$

$$Predict_+ = \frac{90}{90+1} = 0.989 \qquad Predict_- = \frac{99}{99+10} = 0.908$$

$$Accuracy = \frac{90+99}{200} = 0.945 \qquad F1 = \frac{2\times 90}{2\times 90+10+1} = 0.942$$

$$MCC = \frac{90\times 99 - 10\times 1}{\sqrt{(90+1)(99+10)(90+10)(99+1)}} = 0.893$$

Classifier 3



Accuracy	99%
F1 Score	99%
MCC	98%

$$Detect_+ = \frac{99}{99+1} = 0.99 \qquad Detect_- = \frac{99}{99+1} = 0.99$$

$$Predict_+ = \frac{99}{99+1} = 0.99 \qquad Predict_- = \frac{99}{99+1} = 0.99$$

$$Accuracy = \frac{99+99}{200} = 0.99 \qquad F1 = \frac{2\times 99}{2\times 99+1+1} = 0.99$$

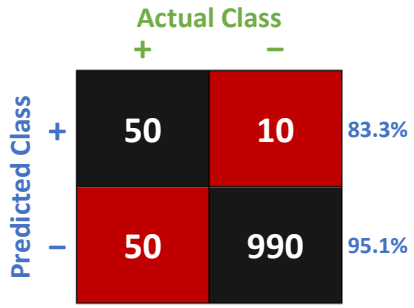
$$MCC = \frac{99\times 99 - 1\times 1}{\sqrt{(99+1)(99+1)(99+1)(99+1)}} = 0.98$$

Accordingly we prefer classifier 3 since we are comparing on the same dataset. Please refer to section 3.9 of the text book for a more detailed discussion of model comparison. In particular, the reader needs to be careful on what constitutes a statistically significant difference of two different models.

4.1.5 Classes Imbalance and Metrics

Let us now see how these metrics react to an increase in one of the classes, i.e. when the problem has imbalanced classes issue.

Classifier 1



Accuracy	94.5%
F1 Score	62.5%
MCC	62.0%

$$Detect_+ = \frac{50}{50+50} = 0.5 \qquad Detect_- = \frac{990}{990+10} = 0.99$$

$$Predict_+ = \frac{50}{50+10} = 0.833 \qquad Predict_- = \frac{990}{990+50} = 0.951$$

$$Accuracy = \frac{50+990}{1100} = 0.945 \qquad F1 = \frac{2\times 50}{2\times 50+50+10} = 0.625$$

$$MCC = \frac{50\times 990 - 50\times 10}{\sqrt{(50+10)(990+50)(50+50)(990+10)}} = 0.62$$

Classifier 2

Accuracy	98.1%
F1 Score	90.0%
MCC	89.0%

$$Detect_+ = \frac{90}{90+10} = 0.9 \qquad Detect_- = \frac{990}{990+10} = 0.99$$

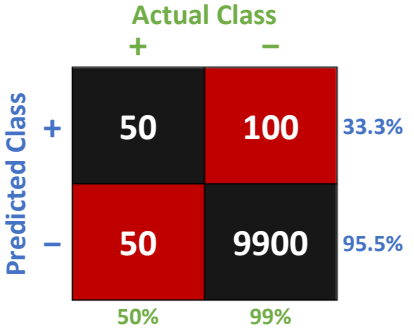
$$Predict_+ = \frac{90}{90+10} = 0.9 \qquad Predict_- = \frac{990}{990+10} = 0.99$$

$$Accuracy = \frac{90+990}{1100} = 0.981 \qquad F1 = \frac{2\times 90}{2\times 90+10+10} = 0.9$$

$$MCC = \frac{90\times 990 - 10\times 10}{\sqrt{(90+10)(990+10)(90+10)(990+10)}} = 0.89$$

Ok now let us see how the metrics react when the problem has a rare class. i.e. it is a severe imbalanced classes problem.

Classifier 1



n=10100

Accuracy	98.5%
F1 Score	40.0%
MCC	40.0%

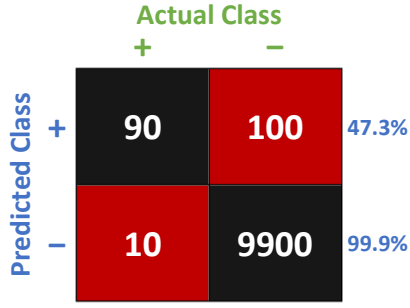
$$Detect_+ = \frac{50}{50+50} = 0.5 \qquad Detect_- = \frac{9900}{9900+100} = 0.99$$

$$Predict_+ = \frac{50}{50+100} = 0.333 \qquad Predict_- = \frac{9900}{9900+50} = 0.995$$

$$Accuracy = \frac{50+9900}{10100} = 0.985 \qquad F1 = \frac{2\times 50}{2\times 50+50+100} = 0.4$$

$$MCC = \frac{50\times 9900 - 50\times 100}{\sqrt{(50+100)(9900+50)(50+50)(9900+100)}} = 0.4$$

Classifier 2



Accuracy	98.9%
F1 Score	62.0%
MCC	64.8%

$$Detect_+ = \frac{90}{90+10} = 0.9 \qquad Detect_- = \frac{9900}{9900+100} = 0.99$$

$$Predict_+ = \frac{90}{90+100} = 0.473 \qquad Predict_- = \frac{9900}{9900+10} = 0.999$$

$$Accuracy = \frac{90+9900}{10100} = 0.989 \qquad F1 = \frac{2\times 90}{2\times 90+10+100} = 0.62$$

$$MCC = \frac{90\times 9900 - 10\times 100}{\sqrt{(90+100)(9900+10)(90+10)(9900+100)}} = 0.648$$

See the following video for a comprehensive example of a DT with different metrics.

4.2 Measuring the performance of Multi-Class Model

Generalising the accuracy from binary-class to multi-class problems is straightforward. We just have to take all the diagonal counts of the confusion matrix and then we divide by the total. Let us take a look at an example.
Below we show the confusion matrix for the IRIS dataset. For more information about the IRIS dataset read the following passage extracted from SKLearn description for the dataset.

Iris plants dataset

Data Set Characteristics:

:Number of Instances: 150 (50 in each of three classes)
:Number of Attributes: 4 numeric, predictive attributes and the class
:Attribute Information:
- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm
- class:
  - Iris-Setosa
  - Iris-Versicolour
  - Iris-Virginica

Summary Statistics:

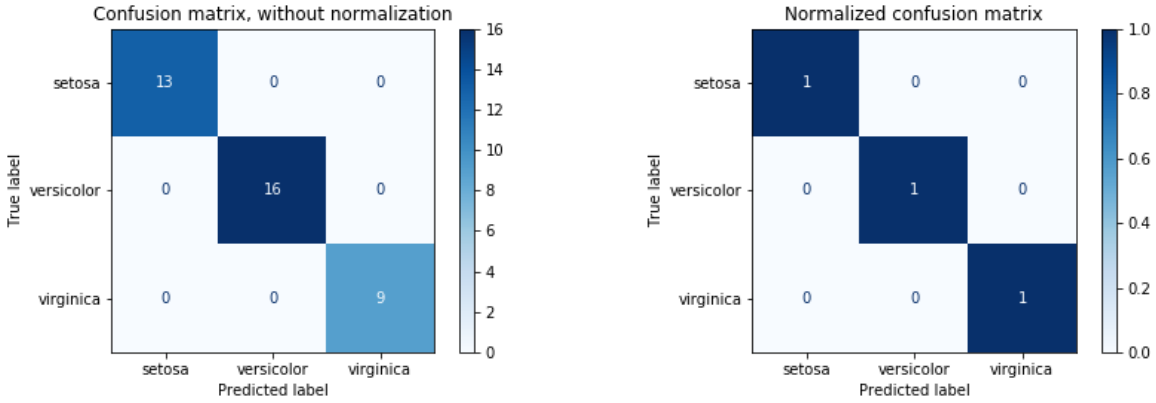
	Min	Max	Mean	SD	Class Correlation
sepal length:	4.3	7.9	5.84	0.83	0.7826
sepal width:	2.0	4.4	3.05	0.43	-0.4194
petal length:	1.0	6.9	3.76	1.76	0.9490 (high!)
petal width:	0.1	2.5	1.20	0.76	0.9565 (high!)

:Missing Attribute Values: None
:Class Distribution: 33.3% for each of 3 classes.
:Creator: R.A. Fisher
:Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
:Date: July, 1988

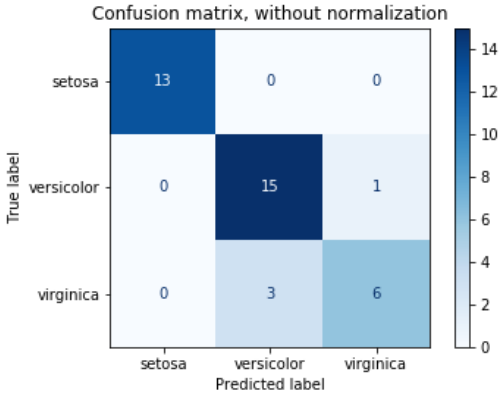
Description

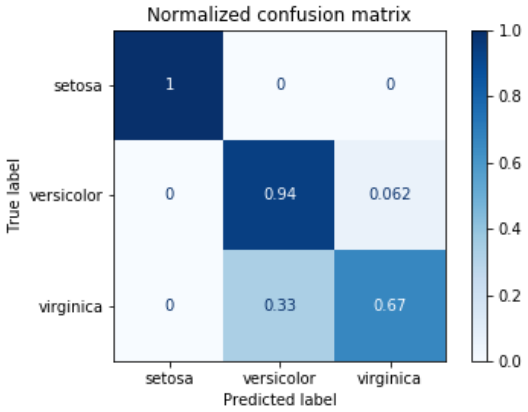
The famous Iris database, first used by Sir R.A. Fisher. The dataset is taken from Fisher's paper. Note that it's the same as in R, but not as in the UCI Machine Learning Repository, which has two wrong data points.
This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. (See Duda & Hart, for example.) The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

Note that the true labels are placed horizontally while the prediction is vertically. This is opposite to what we have used before, but as we said earlier it should not matter as long as we are vigilant about it.



Different sources uses these two formatting as well). On the right also you can see the same confusion matrix after normalisation. We normalise by dividing each entry by the sum along the *true label axis*. Below you will see an example that clarifies this.





The accuracy for a binary classification problem is given as

$$\boldsymbol{Accuracy} = \frac{1}{N}(\mathbf{TP} + \mathbf{TN})$$

The accuracy for a multi-class problem is given as

$$\boldsymbol{Accuracy} = \frac{1}{N}\sum_{i=1}^{C} \mathrm{TC}_i$$

Where TC_i represents the count of the correctly classified instances of class C_i.

So for example the accuracy of a classification model that have the above confusion matrix when applied on the iris dataset is given as

$$\boldsymbol{Accuracy} = \frac{1}{38}\left(\mathrm{TC}_{setosa} + \mathrm{TC}_{versicolor} + \mathrm{TC}_{virginica}\right) = \frac{1}{38}(13 + 15 + 6) = 0.894$$

4.2.1 Accuracy Limitation

One issue that we face with the accuracy of multi-class and binary-class is that it favours the dominance class with more instances. For example, in the iris confusion matrix the count for the virginica is 9 only while for the versicolor is 16 this means that the performance of the model on the versicolor class will overshadow and dominate its performance on the virginica. Sometimes this desirable if we want the performance measure to be consistent with the dataset class distribution. Other times, when we are more interested in a balanced score of all classes regardless of their distributions, or when we want to emphasise the importance of a rare class due to a difficulty in detecting it, then the accuracy is not suitable. In these cases, balanced scores are preferred. In the next two subsection, we will talk about the $pr(detect)$ and $p(predict)$ ***scores*** for multi-class problems (aka recall and precision ***scores,*** respectively). These are balanced scores that are more suitable to imbalanced class datasets. In fact the detect score is also known as the balanced accuracy score.

4.2.2 Holistic Metric for Multi-Class: $p(detect)$ (aka Recall Score or Balanced Accuracy Score)

Balancing out the detection scores (recall scores) of different classes can be performed in few ways. Essentially we calculate the detection rate (recall) of the model for individual classes and then we can

1- Either take the arithmetic mean of these rates and in this case each class has equal weight, even if the class has low probability.
2- Or we take a weighted average of the detections rates. This will allow us to assign different weights for the different classes, the weights must sum up to 1 and reflect the relative importance that we want to assign to each class.
    a. If we assign the class distributions to the weights for the detection rates then we go back to the usual accuracy score.

For multi-class problems, the recall or $\boldsymbol{p(detect_{class})}$ can be defined in terms of averaged sum of true instances of each class. To demonstrate how, let us look into the above confusion matrix. The recall for each class separately give us the following:

$$\boldsymbol{p(detect_{setosa})} = \frac{13}{13}\,,\ \boldsymbol{p(detect_{versicolor})} = \frac{15}{16},\ \boldsymbol{p\left(detect_{virginica}\right)} = \frac{6}{9}$$

Which yields $1.0, 0.9375$ and 0.666 for the classes 'setosa', 'versicolor', 'virginica', respectively and these are the scores that we can see on the normalised confusion matrix after rounding to 2 decimals. Now, the recall for the above model can be calculated in several ways, some of them are:

1- Macro detect score (aka recall score or balanced accuracy)

$$\boldsymbol{p(detect)} = \frac{\mathbf{1}}{\mathbf{3}}\left(\boldsymbol{p(detect_{setosa})} + \boldsymbol{p(detect_{versicolor})} + \boldsymbol{p\left(detect_{virginica}\right)}\right) = \mathbf{0.868}$$

This is just arithmetic average which will assume that all classes the same importance.

2- Weighted detect score (aka weighted recall score or weighted balanced accuracy)
For example if we decided that the relative importance of the classes are as follows:

$$w_{setosa} = 1/4, w_{versicolor} = 1/4, w_{virginica} = 1/2$$

Then the weighted detection score (or balanced accuracy score) is given as

$$\boldsymbol{p(detect)} = \frac{\mathbf{1}}{\mathbf{4}}\boldsymbol{p(detect_{setosa})} + \frac{\mathbf{1}}{\mathbf{4}}\boldsymbol{p(detect_{versicolor})} + \frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{p\left(detect_{virginica}\right)}$$

$$\boldsymbol{p(detect)} = \frac{1}{4} \times 1 + \frac{1}{4} \times 0.9375 + \frac{1}{2} \times 0.666 = \mathbf{0.8177}$$

This metric shows that the classifier that we trained has less weighted accuracy than the balanced accuracy since we emphasised the detection of the virginica more than other classes and the model did not performed that well on this particular class ($\boldsymbol{p\left(detect_{virginica}\right)} = 0.666$).

Now, let us try to use the classes' distribution as the weights for the classes' detection scores to see where this can lead us. The count that we have is $\boldsymbol{n = 13 + 16 + 9 = 38}$ for 'setosa', 'versicolor', 'virginica' classes, respectively. This is not the count of the entire dataset $N = 150$ because the above confusion matrix is for a testing set not the total dataset.

The weights for the classes are: $w_{setosa} = 13/38, w_{versicolor} = 16/38, w_{virginica} = 9/38$

Hence, the balanced accuracy is:

$$\boldsymbol{p(detect)} = \frac{\mathbf{1}}{\mathbf{38}}\left(\mathbf{13}\boldsymbol{p(detect_{setosa})} + \mathbf{16}\boldsymbol{p(detect_{versicolor})} + \mathbf{9}\boldsymbol{p\left(detect_{virginica}\right)}\right) = \mathbf{0.8947}$$

The detection probabilities are as follows:

$$\boldsymbol{p(detect_{setosa})} = \frac{13}{13}, \qquad \boldsymbol{p(detect_{versicolor})} = \frac{15}{16}, \qquad \boldsymbol{p\left(detect_{virginica}\right)} = \frac{6}{9}$$

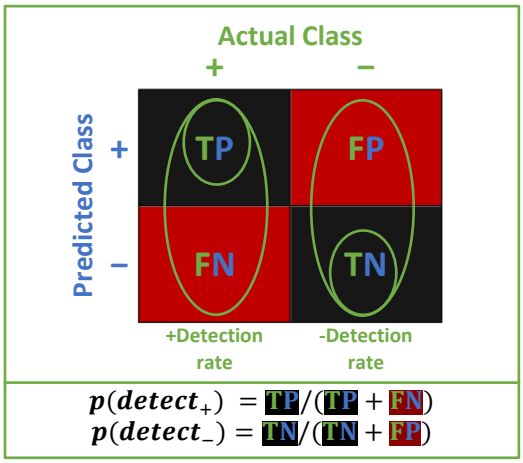
$$\boldsymbol{p(detect)} = \frac{\mathbf{1}}{\mathbf{38}}\left(\mathbf{13}\frac{\mathbf{13}}{\mathbf{13}} + \mathbf{16}\frac{\mathbf{15}}{\mathbf{16}} + \mathbf{9}\frac{\mathbf{6}}{\mathbf{9}}\right) = \frac{\mathbf{1}}{\mathbf{38}}(\mathbf{13 + 15 + 6}) = \mathbf{0.8947}$$

Which means that choosing the classes' distribution as the weights for a weighted balanced accuracy brings us back to the usual accuracy measure.

Here we would like to point out the macro balanced accuracy for binary class's problem is defined as

$$\textbf{Balanced Accuracy} = \frac{1}{2}\left(\frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{TN} + \text{FP}}\right) = \frac{1}{2}p(detect_{+}) + \frac{1}{2}p(detect\_)$$

A weighted average version can be defined as we showed earlier.



4.2.3 Exercise

Try to calculate the balanced accuracy for the above problems and compare it with the accuracy and see if makes any difference.

4.2.4 Holistic Metric for Multi-Class: $p(predict)$ (aka Precision Score)

All calculations for $pr(predict)$ (aka precision) extends naturally similar to what we did for the $pr(detect)$ (aka recall). As before we can calculate the precision for each class as follows:

$$p(predict_{setosa}) = 13/13\ ,\ p(predict_{versicolor}) = 15/18\ ,\ p(predict_{virginica}) = 6/7$$

Which yields $1,\ \ 0.8333,\ \ 0.857$ for the classes 'setosa' 'versicolor' 'virginica', respectively.

1- Macro prediction score (aka macro precision score) is given as

$$p(predict) = \frac{1}{3}\left(p(predict_{setosa}) + p(predict_{versicolor}) + p(predict_{virginica})\right) = \mathbf{0.8968}$$

2- Weighted

The count for the respective classes are $\boldsymbol{n = 13 + 16 + 9 = 38,}$ the weights for the classes are: $\boldsymbol{w_{setosa} = 13/38}$ $\boldsymbol{w_{versicolor} = 16/38, w_{virginica} = 9/38}$. Hence, the prediction score (aka the precision score is given as:

$$p(predict) = \frac{1}{38}\left(13p(predict_{setosa}) + 16p(predict_{versicolor}) + 9p(predict_{virginica})\right)$$
$$= \frac{1}{38}\left(13 \times \frac{13}{13} + 16 \times \frac{15}{18} + 9 \times \frac{6}{7}\right) = \mathbf{0.8959}$$

Note here that we cannot use the counts on the diagonal of the confusion matrix as in $\boldsymbol{pr(detect)}$ and we are not back to some other basic measures as in the detect case.

4.2.5 Holistic Metric for Multi-Class: $F1$ Score

To generalise the F1 score into multi-class problem we can

1- Either use overall predict and detect scores (recall and precision) and then we apply the harmonic mean on them as in the binary class problem. Here we can use either the macro detect and predict scores or the weighted detect and predict scores.

2- Or we can also treat the classes as one vs. the rest fashion (yielding the problem into a binary class problem) and obtain the F1 score for each class separately. And then we take the average of the F1 scores for all the classes.

Again we can use either the macro or the weighted method consistently (do not mix macro with weighted for different classes).

We will show the first method here (however note that 2 is preferred over 1, see next exercise). In the last couple of sections we saw that the overall macro detection and precision for the given examples are: $\boldsymbol{p(predict) = 0.8968,}$ $\boldsymbol{p(detect) = 0.868.}$

So, the harmonic average:

$$\boldsymbol{F1\ score = \frac{2p(predict)p(detect)}{p(predict) + p(detect)} = 0.8821}$$

So we can see that the value for the F1 score lies between the $\boldsymbol{p(predict)}$ and $\boldsymbol{p(detect)}$.

Note, there is a yet another approach that we have not shown which is the micro detect and micro predict and F score all of which yield the accuracy and hence are omitted.

Note that for binary class problems, we only take the harmonic mean of the $\boldsymbol{p(detect_+)}$ and $\boldsymbol{p(predict_+)}$ for the positive classes only. While, for the multi-class problems, we take the harmonic mean of the $\boldsymbol{p(detect)}$ and $\boldsymbol{p(predict)}$ for all the classes.

Please note also that method macro are special cases of weighted measures (macro and weighted F1 score and macro and weighted predict and detect scores) where the weights are all $= 1/C$ the number of classes). On the other hand, 1 and 2 are two different forms for F1 score and are not necessarily equivalent.

4.2.6 Exercise

Extend the F1 score, as per the second method, into a multi-class case and calculate it for the above example. See the following paper that compares the two different methods and see the following paper to see how to calculate F1 according to the second preferred method. In practice you might want to consider both (1 and 2) and compare or at least use 2.

4.3 LESSON SUMMARY

In this lesson we have covered various metrics for binary and multiclass problems to evaluate classification models in general. Please note that these metrics are not necessarily related only to decision trees. They are general metrics that can be used for any classification model. In later units we will see different classification techniques, and these metrics will also be applicable to them. Please be aware that we have used a unified and simple notation for the metrics, but these have different names in the wider community and you should be aware of these names. We denoted precision as predict, and recall as detect.

5 OVERFITTING, UNDERFITTING AND GENERALISATION

Learning outcomes:

After completing this lesson you should be able to:

- understand the root cause for overfitting and underfitting
- take steps to mitigate the problem of overfitting and underfitting
- pick a suitable cross validation technique to obtain a more accurate measure for the model performance
- employ cross validation with hyper parameter optimisation

We saw earlier that our decision tree model was perfectly capable of classifying animals according to our simple dataset, without any mistakes. There was no misclassification occurring. However, this is not always the case. The accuracy of the model was partly due to the simple and deterministic nature of the dataset that we handled and because it is based on clear-cut biological findings.

As we mentioned in unit 1, datasets are often far from perfect. This can be due to data collection deficiencies, as well as the intrinsic errors and noise of the underlying phenomena that the dataset records. This will make coming up with a perfect model not only impossible, but actually **undesirable**. For example, some labels might be incorrect, or some features inaccurately measured or captured via a device with deficiencies. In cases such as these, if our model comes up with perfect answers for all the training examples, it almost always means that it captured not only the essential relationship between the features and the class, but also the noise and errors in the data. This is **undesirable**, and can lead to serious problems later when we want to use the model to predict the class of a data point that it has **not seen** before (during training).

Using the training set to come up with a **general** relationship between the features and the class that can be utilised to later classify **unseen** examples is called **model generalisation**. This generalisation ability is crucial to the success of the model if it is to become a part of comprehensive software solution. Two main problems that can prevent a model from generalisation are **overfitting** and **underfitting**. In this context then we need to come up with metrics that are capable of capturing misclassification by the model and its generalisation ability by capturing the misclassification errors of unseen data points.

5.1 Training and Testing Sets

Because of the potential problems mentioned above, we often need to do the same performance measurement for the training set and on another separate dataset that we call the test set. In order to measure the performance for the test set we need to have the labels available in the same manner that they are in the training set. Remember, classification is a supervised learning problem so we have the answers (labels) available during training and test, but not when we actually use the model to predict unknown labels. If we do not have separate training and test sets, we can simply **partition** our dataset into two sets, with one acting as a **training set** and the other as a **test set**. Note that the term partition indicates that there is no overlap between the training and test set.

5.1.1 Proportions of Training and Testing Sets

Unless there is abundance in the data that captures a simple classification model, or there is already a dataset set aside for testing, we often want to split the original dataset into training and testing sets. The proportion or percentage of the split can be any as along as the training data captures all the different patterns the classes can have. Often we specify a 70% to 30% split for the training and testing respectively.

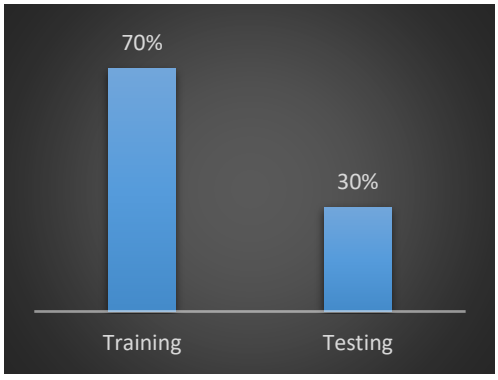


Figure 5.1: Common choice for Training and Testing proportions of the original dataset.

5.1.2 Splitting with Stratification

When we split we need to take into account the distribution of the labels in the dataset. We want to reserve the structure of the original dataset in the partitioned training and test sets. One way to guarantee this property is by **stratified sampling**. Stratification allows us to preserve the classes' distribution. So for example if we have binary classes {Class1, Class2} with the first class occupying 60% of the dataset and the second class occupying 40% of the dataset, then when we split with stratification into training and test sets, the distribution of classes {Class1, Class2} inside the training set is 60% to 40% and the same also true for the distribution of the classes inside the test set.

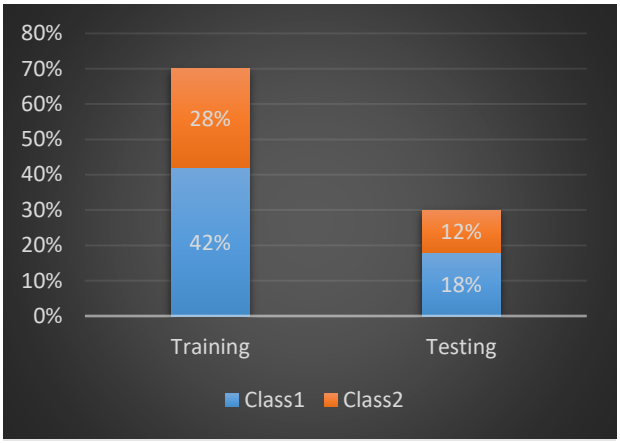


Figure 5.2 Stacked columns figure shows stratification for training 70% and testing 30% sets for two classes. Class1 and Class2 are distributed into 60%, 40% in the original dataset and the same percentage kept (through stratification) for the partitioned training and testing sets resulting in the percentages shown in the figure.

5.2 Cross Validation, Model Testing, Model Selection and Model Comparison

Model testing is the process of testing to see how good the generalisation ability of the model is. We use one of the splitting regimes mentioned above, i.e. randomised splitting or with stratification. What we would like to speak about here is how to choose a good hyper parameter for a model. This is called model selection. In model selection we are normally talking about the same classification technique (such as decision trees), but we are interested in which decision tree is best for our problem. In model comparison we are talking about different techniques (such as decision trees vs. k-nearest neighbours) and we would like to come to a conclusion about which technique is the best.

5.2.1 Cross Validation

Cross validation is an elaborate testing technique that is useful to balance out the different patterns that might exist in our data. Think about it, when we select a testing set we might be lucky and get a test set that the model is particularly good at predicting its instances, while it may not be as good for other instances. To balance out the ability of the model on different patterns that underline the instances, we can repeatedly select different testing sets and take the average of the testing results to be a more representative value for the performance of the model. Cross validation is one example of this strategy, where we partition our dataset into a number of subsets with equal instances. We hold out a subset for testing, we train on the rest of the subsets, we repeat for all subsets then we take the average. We call those subsets folds.

For example, let us assume that we have a dataset S. To perform a 3 folds cross validation on it we partition our data into 3 almost equal subsets of S_1, S_2, S_3 where we have $S = \{S_1 \cup S_2 \cup S_3\}$.

Now we train the model on subset $\{S_1 \cup S_2\}$ while we test the trained model on subset S_3 which will give a generalisation error $Err(S_3)$.

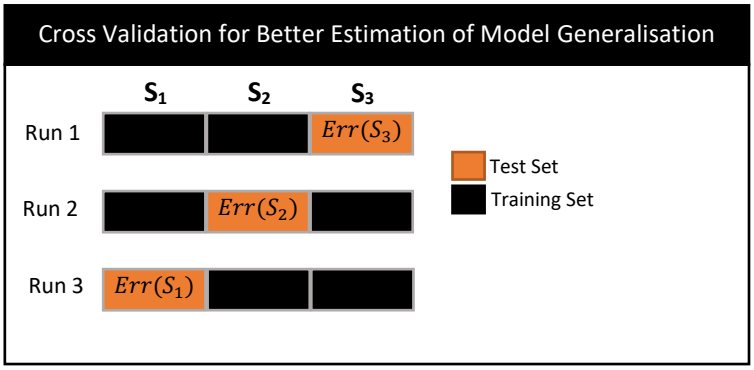
We repeat the process by training on subsets $\{S_1 \cup S_3\}$ and test on subset S_2 to obtain the generalisation error $Err(S_2)$, then we train on subsets $\{S_2 \cup S_3\}$ and test on subset S_1 to obtain the generalisation error $Err(S_1)$. We then average all the tests to obtain $Err(S)$. We can also look into the standard deviation to see how much variation is there in our dataset

with respect to our model's ability to generalise. Figure 5.3 below shows this process. For more details, please have a look at Algorithm 3.2 and 3.3 of Tan et al (2020).

Figure 5.3: 3 folds cross-validation example.

5.2.2 Model Selection

Model is selection is the process of selecting the best model for the problem at hand, from among several other possible models of similar prediction powers. For example, we can build several decision trees with different depths for the same



problem, and each tree would have its own properties and prediction strength. The depth of the DT is an example of what we call a hyper parameter. It is called so because changing this parameter fundamentally changes the properties of the model. The question would be then which one is the best for the problem in hand. We often first perform **model selection** to pick the best candidate of a set of models and then we do **model comparison** to pick a final model that corresponds to the best technique. We will talk about model comparison in a later section.

Model selection can be performed mainly in two ways. One approach is to evaluate a set of possible models by enumerating through a set of discrete or discretised hyper parameters values. We will have to then evaluate each one separately and compare between them all to arrive to the best value among these pre-defined values. This is the approach that we will take in this module.

Another approach is to sample from a set of potential models and come up with an algorithm that will help us to find a good estimation of the hyper parameters. This approach will be covered in the Machine Learning module. A third and best approach is to analytically find the best hyper parameter value by analysing the problem and coming up with automated process to give us the best model among many others (and potentially infinite number of models). This is sometimes possible with the Bayesian approach where the hyper parameters (the mean and variance) can be found by performing an expectation maximisation process to find the best hyper parameter. Then we can integrate out (marginalising) the possible hyper parameters values to come up with an estimation of the performance of the model. A method called Bayesian processes is an example of such analytical approach. However, this approach is not always possible due to the difficulties that arises with the mathematical integrals. Again, we will study this type of problem in the Machine Learning module.

5.2.3 Model *Selection* with Cross Validation

In model selection especially when we do not have enough data, we can use cross validation to aid in the process of selecting the best hyper parameter. We first start by splitting the data into folds (ex. 3 folds). Let us assume that we have hyper parameter h (ex. Tree depth) with values v_1 and v_2 (ex. Depth = 5 and Depth = 10) and we would like to know what value we should pick for our model. In this case, we can set $h = v_1$ and training the model on subsets $\{S_1 \cup S_2\}$ while we test the trained model on subset S_3 which will give a generalisation error $Err_{v1}(S_3)$.

We repeat the process by training on subsets $\{S_1 \cup S_3\}$ and test on subset S_2 to obtain the generalisation error $Err_{v1}(S_2)$, then we train on subsets $\{S_2 \cup S_3\}$ and test on subset S_1 to obtain the generalisation error $Err_{v1}(S_1)$. We then average all the tests with hyper parameter h value of v_1 to obtain Err_{v1} of our model. We repeat the same process for hyper parameter value v_2 to obtain $Err_{v2}(S)$ we then compare $Err_{v1}(S)$ and Err_{v2} and we pick the value that minimise the error. i.e. we pick the value with the least error. The figure below summarise the model selection for hyper parameter h.

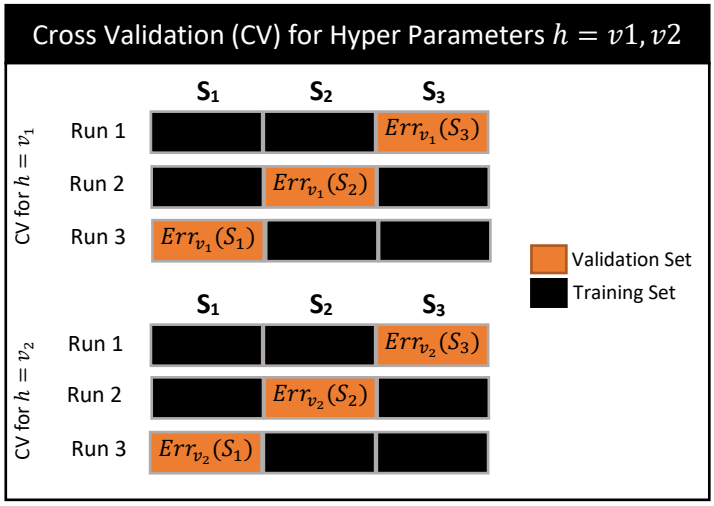


Figure 5.4: Model Selection with 3 folds cross-validation example.

5.2.4 Model *Evaluation* of Cross-Validated Selected Model with Hyper Parameters

After we have selected the best parameters for our model, we want to evaluate the performance of the model. A pitfall would be to use the averaged cross validation error as an indication for the performance of the model. This a biased estimation of the generalisation ability of our model because we have already used the validation data to select the best hyper parameters. Therefore, we need to reserve a portion of the dataset for this final evaluation of the resultant selected model. The below figure shows this complete process.

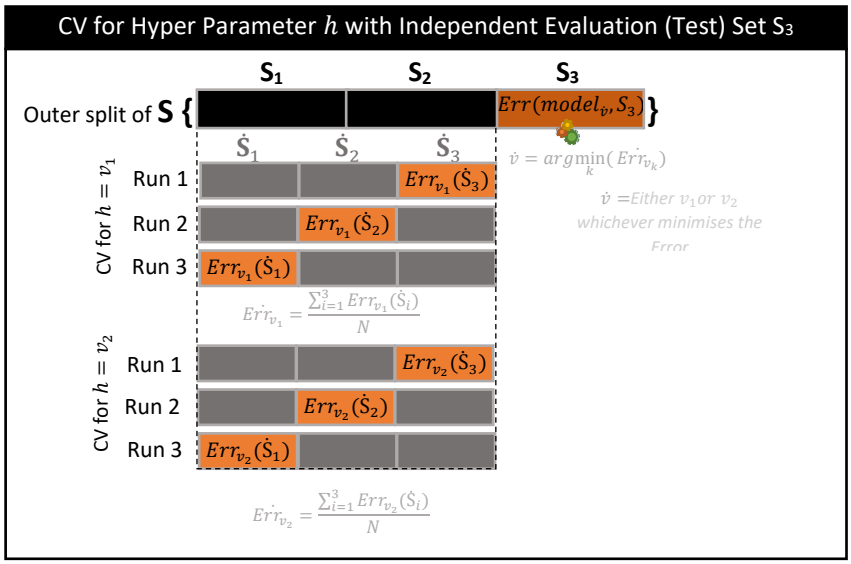


Figure (5.5): Model Selection with 3 folds cross-validation and an outer split to facilitate an independent test set for evaluating the final resultant model denoted as $model_{\dot{v}}$. Note that $\dot{v}$ takes the best value of $\{v_1, v_2\}$ that minimises the average errors $\dot{Err}_{v_1}, \dot{Err}_{v_2}$. So for example if we assume that $\dot{Err}_{v_1} > \dot{Err}_{v_2}$ then $\dot{v} = v_2$.

5.2.5 Evaluation of Cross Validation Selected Model with and Grid Search for Hyper Parameters

When we have multiple hyper parameters and we want to select the best combination of values for them, we can employ several search techniques. Here we can take two approaches. One simple approach is to perform an exhaustive search of all the possible combinations of the hyper parameters. We evaluate the models that stem from them and we select the model with the least generalisation error or highest overall accuracy. This is called grid search because each hyper parameter adds a dimension to a grid of possible values. For example if we have three hyper parameters the first with 5 values, the second with 5 values and the third with 2 values, then the number of combinations of these values is $5 \times 5 \times 2 = 100$ different combinations. So in this approach we need to evaluate all of these values to select the best model among them. There are less computationally expensive but inexact methods that we can employ to search for close to optimal values. These include hill climbing, simulated annealing some of these search methods will be covered in the Algorithms module.

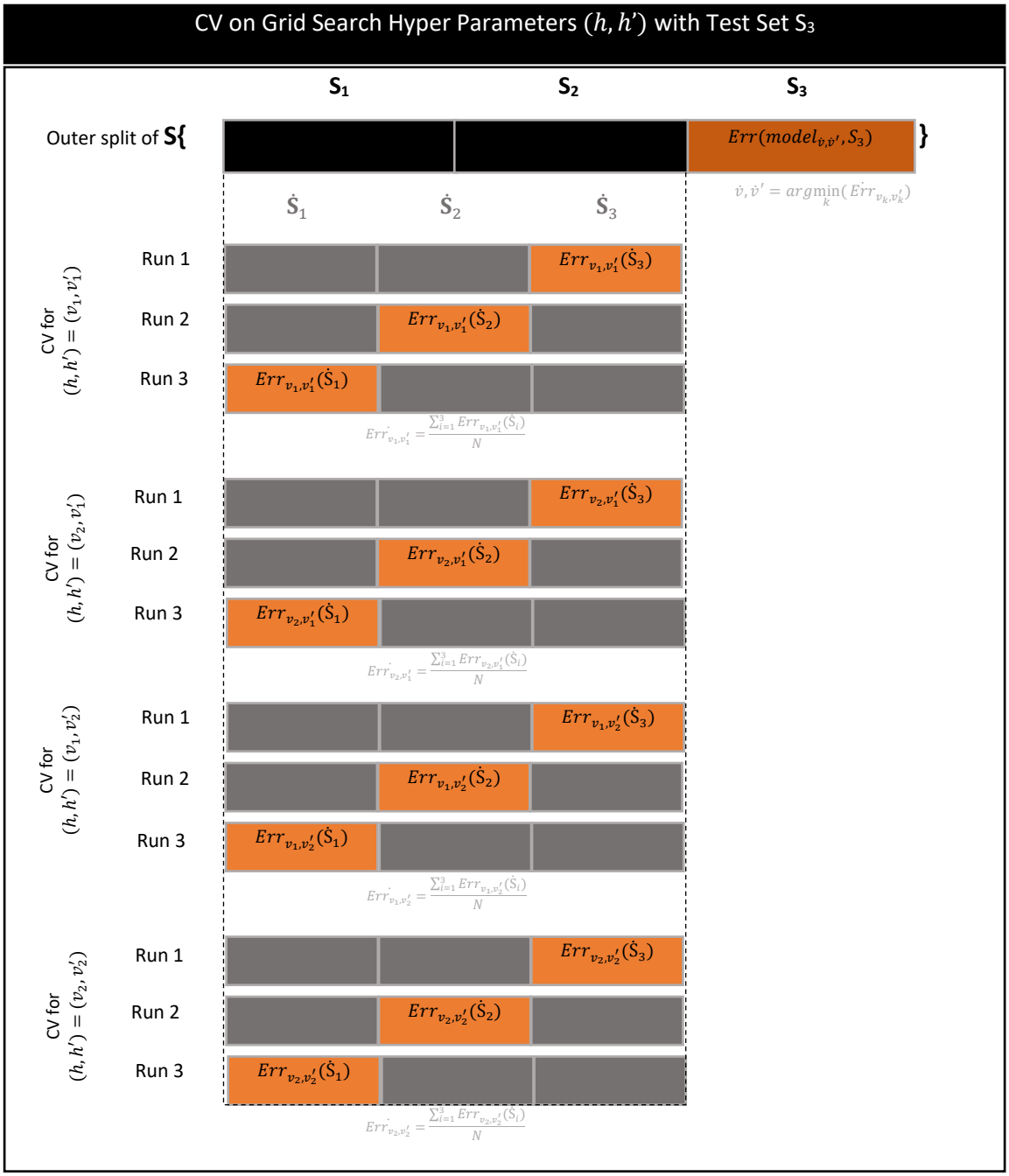


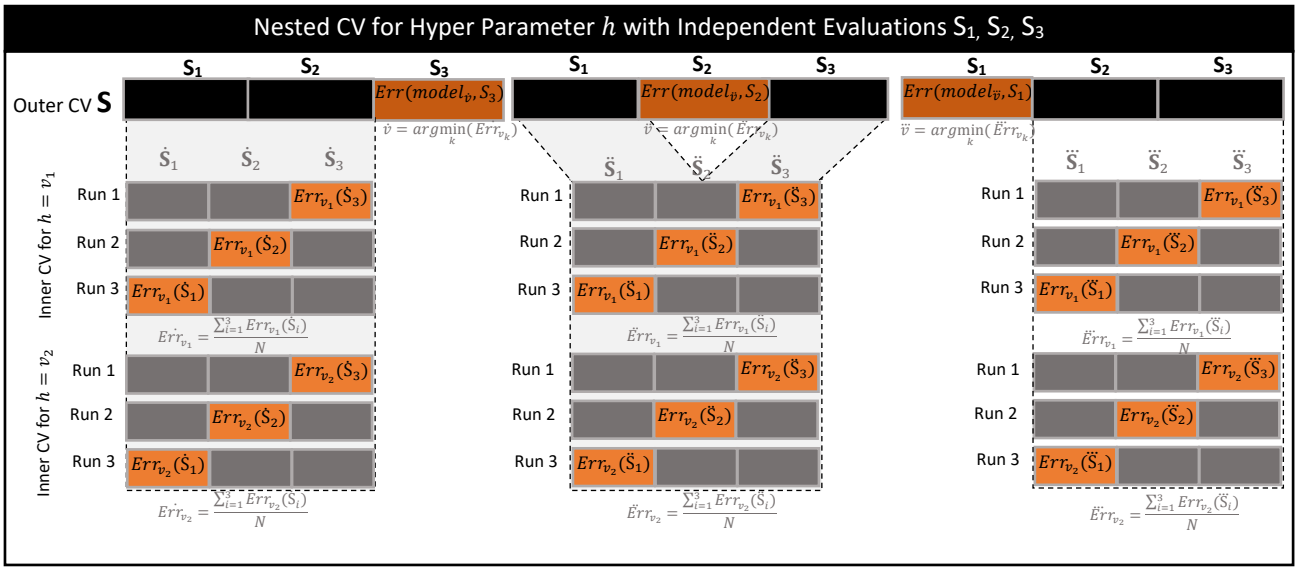
Figure (5.6): Model Selection with 3 folds cross-validation and an outer split to facilitate an independent test set for evaluating the final resultant model denoted as $model_{\dot{v}}$. Note that $\dot{v}$ takes the best value of $\{v_1, v_2\}$ that minimises the average errors $\dot{Err}_{v_1}, \dot{Err}_{v_2}$. So for example if we assume that $\dot{Err}_{v_1} > \dot{Err}_{v_2}$ then $\dot{v} = v_2$.

5.2.6 Model *Evaluation* with Nested Cross Validation and Hyper Parameters

We have shown how to evaluate a selected model on an unseen data in the previous section. However, we should note that this estimation is still a reflection of the model performance only one part of the dataset. But what if we wanted to estimate the performance on all the dataset without seeing the data? Before we show how, it should be clear in your mind that we will not be able to show an overall unbiased performance on all the available data for a particular model. We can however get an estimation by averaging the performances of all possible models that stem from the different parts of the dataset. So this section is not for model selection, it just for model evaluation.

This where nested cross validation comes to the rescue. The idea now is that we split our data into folds and we use cross validation to evaluate the performance of the model on an **unseen** dataset while we use the rest of the data to perform another inner cross validation to select the best parameters. This called nested cross validation. Note that we cannot use this to select a model (we have already done that in each inner CV) we use it just to come up with an unbiased estimate of the generalisation ability of a technique. There are no specific hyper parameters that we will get out of this procedure. Note, however that you still can use the outer splitting regime without the use of outer cross validation to get a final performance on an unseen test set for a particular selected best model hyper parameters, but bear in mind that this estimation is still a reflection on only part of the dataset. The figure below shows this procedure.

Figure (5.7): Nested Cross Validation Evaluation selected models, both inner and outer cross-validation have 3 folds.



Note that we have $3 \times 3 \times 2$ models to be trained and this can be consuming and computational expensive depending on the size of the dataset.

For simplicity of presentation, the hyper parameter h is assumed to have two different values that it can take $\{v_1, v_2\}$. Each fold of the outer CV has an inner CV process that will be executed inside it to suggest a best value for the hyper parameter h. As we saw earlier an inner CV gives its own best value for h that stems from its error comparisons. Since we have 3 –fold outer CV we get 3 values which might all be v_1 or v_2 or a mix of both (ex. v_1, v_1 and v_2). We have represented the best values of the three outer folds as $\dot{v}, \ddot{v}$ and $\dddot{v}$ (ex. $\dot{v} = v_1, \ddot{v} = v_1, \dddot{v} = v_2$). These values in turn give us 3 (possibly different) models $model_{\dot{v}}, model_{\ddot{v}}$ and $model_{\dddot{v}}$, therefore, the final results is an average of the *errors* or accuracy of the 3 different models. We can take a vote on the best value of h to produce a final model (ex. v_1) but the final averaged errors is not guaranteed to be unbiased unless we do yet another third CV process.

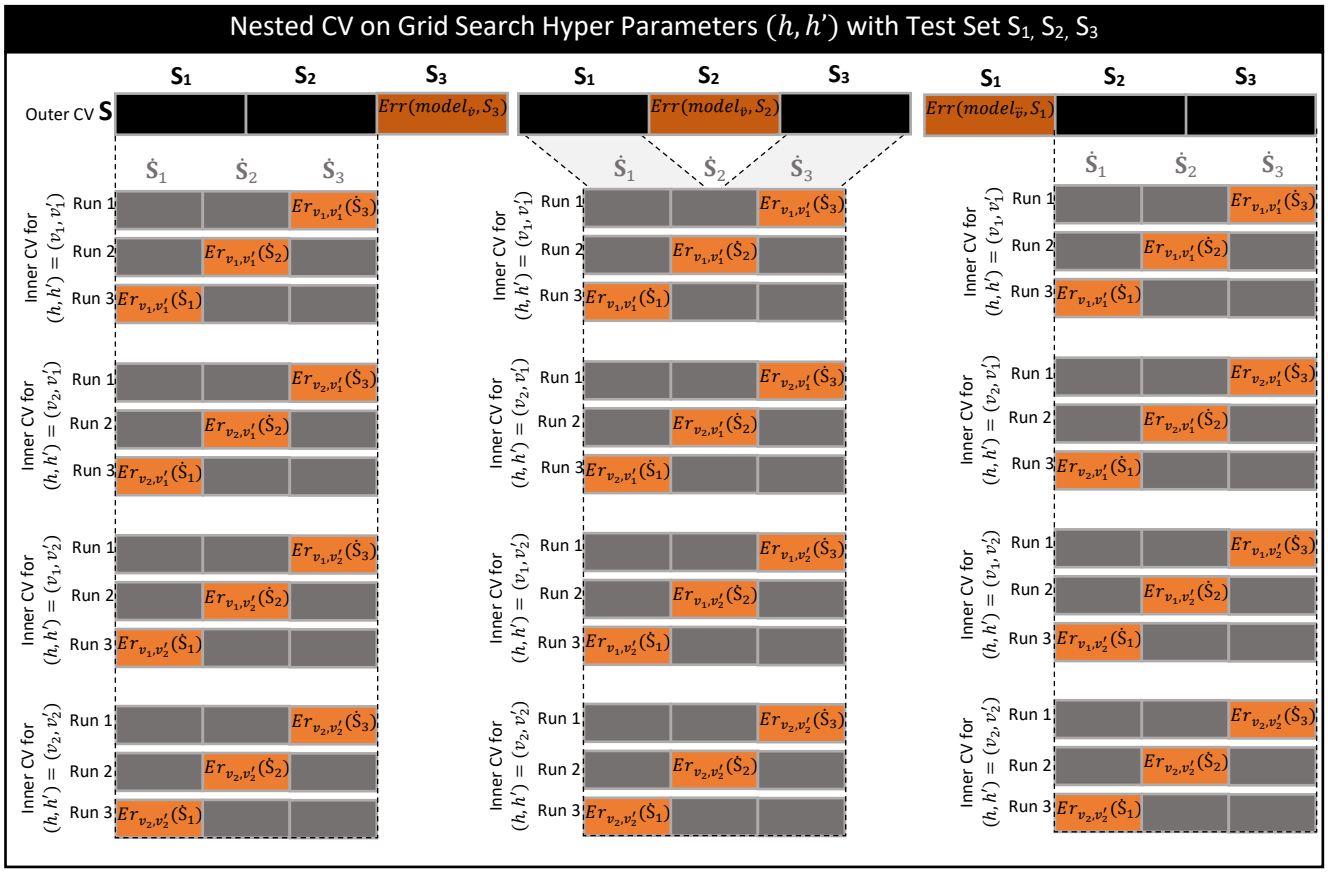


Figure (5.8): Nested cross-validation evaluation for grid-search model selection, both inner and outer cross-validation have 3 folds. Note that we have $3 \times 3 \times 4$ models to be trained and this can be consuming and computational expensive depending on the size of the dataset.

For simplicity of presentation, the two hyper parameters h and h' are assumed to have two different values $\{v_1, v_2\}$ and $\{v'_1, v'_2\}$ that they can take respectively. Each fold of the outer CV has an inner CV process that will be executed inside it to suggest a best value for the hyper parameter (h, h'). In the grid search process we take all the possible combinations of the hyper parameters values. So in our simple case we have 2×2 possible combinations. Note that this has an exponential growth rate. So, if we have 5 hyper parameters, each with 3 values then to perform the grid search we need to consider $3^5 = 243$ combinations , i.e. we have 243 models to train. So Grid search for this case can quickly becomes intractable. If we take into account also an outer and an inner CV operations that we would like to perform to obtain unbiased results then the results would be $3 \times 3 \times 243 = 2187$ models to train. The bottom line is that we have to be careful on how many models we are training when we use exhaustive search methods such as the Grid search. There are cheaper but inexact methods that we can employ to search for close to optimal values. This include hill climbing, simulated annealing some of these search methods will be covered in the Algorithms module.

As we saw earlier an inner CV gives its own best value for h that stems from its error comparisons. Since we have 3 –fold outer CV we get 3 values which might all be v1 or v2 or a mix of both (ex. $v1$, $v1$ and $v2$). We have represented the best values of the three outer folds as $\dot{v}, \ddot{v}$ and $\vec{v}$ (ex. $\dot{v} = v1, \ddot{v} = v1, \vec{v} = v2$). These values in turn give us 3 (possibly different) models $model_{\dot{v}}, model_{\ddot{v}}$ and $model_{\vec{v}}$, therefore, the final results is an average of the *errors* or accuracy of the 3 different models. We can take a vote on the best value of h to produce a final model (ex. $v1$) but the final averaged errors is not guaranteed to be unbiased unless we do yet another third CV process.

5.2.7 Exercise:

See the following Jupyter notebook exercise on Nested CV.

5.2.8 Model *Comparison* with Cross Validation

When we want to compare the performance of different techniques (like a decision tree and k-nn), we differentiate between two different cases. The first when we are comparing on the **same dataset**. In this case, we can use the results of an evaluation process directly as we did in earlier sections. The difference is that we have different techniques instead of the same technique with different hyper parameters.

Cross validation is a suitable strategy to balance out the performance of the model on all types of patterns that exist in our dataset. The strategy is similar to what we have done for model selection. So first we split the data into testing and training. We keep the testing part for testing the different models and compare between them we take the model selection approach whether with grid search if we have multiple hyper parameters and we select the best candidate of each techniques. If we want to evaluate the expected performance of each model we take the evaluation approach of selected models. In this case we might not be necessarily come with a particular model unless we do some voting regime. Then we compare the performances of the evaluation and come up with a final performance of a preferred technique.

We divide as before into K folds and we try different hyper parameters values until we come up with best set of hyper parameters for Model 1 that uses Techniques 1, we refer to is as Model(Tech1), then we repeat the same process of choosing the best combination of hyper parameters for Model(Tech2). Now we cross validate both models on the outer loop by testing them on S_1, S_2 and S_3 folds and we finally come up with the average performances and we pick the technique with the best average.

On the other hand, in case we are comparing the performance of two techniques on two **different datasets**, then it is less obvious how to choose among the different techniques. This issue often arises when we do benchmarking of different techniques on two different datasets; one of them is already performed earlier and we want to add insight onto the techniques' properties by applying them on a different problem. In this case, we need to employ statistically comparisons on the significance of the variation of the results. Please refer to section 3.9 of Tan et al (2020) for more details.

5.2.9 Exercise RapidMiner

See the following video on RapidMiner Grid search procedure for selecting multiple hyper parameters for a model. The example shows how to optimise the depth of the tree along with the minimum leaf size.

5.3 OVERFITTING

Overfitting occurs when we keep trying to improve the performance of the model to the extent that the model starts to capture the noise in addition to the actual relationship between the features and the class.

It is expected that the performance for the test set would be slightly lower (roughly $< 5\%$) than the training set (in other words the error in the test set is higher than that of the training set) even when the model does not suffer from overfitting. However, if we realise that there is a relatively large difference between the two then it is a sign that overfitting has occurred during training.

5.3.1 Examples of overfitting

Below we see an example of overfitting, as it can be seen the data can be classified by rectilinear decision boundaries that are specified by four conditions, however due to the noise that we added by spreading out some of the class C1 points and overly trying to isolate and classify those pockets, the tree in turn is overly grown and become unnecessarily complex. Bear in mind that there is no perfect solution here due to noise there will be always inaccuracy that occurs in the classification decision of the tree and we just need to live with them.

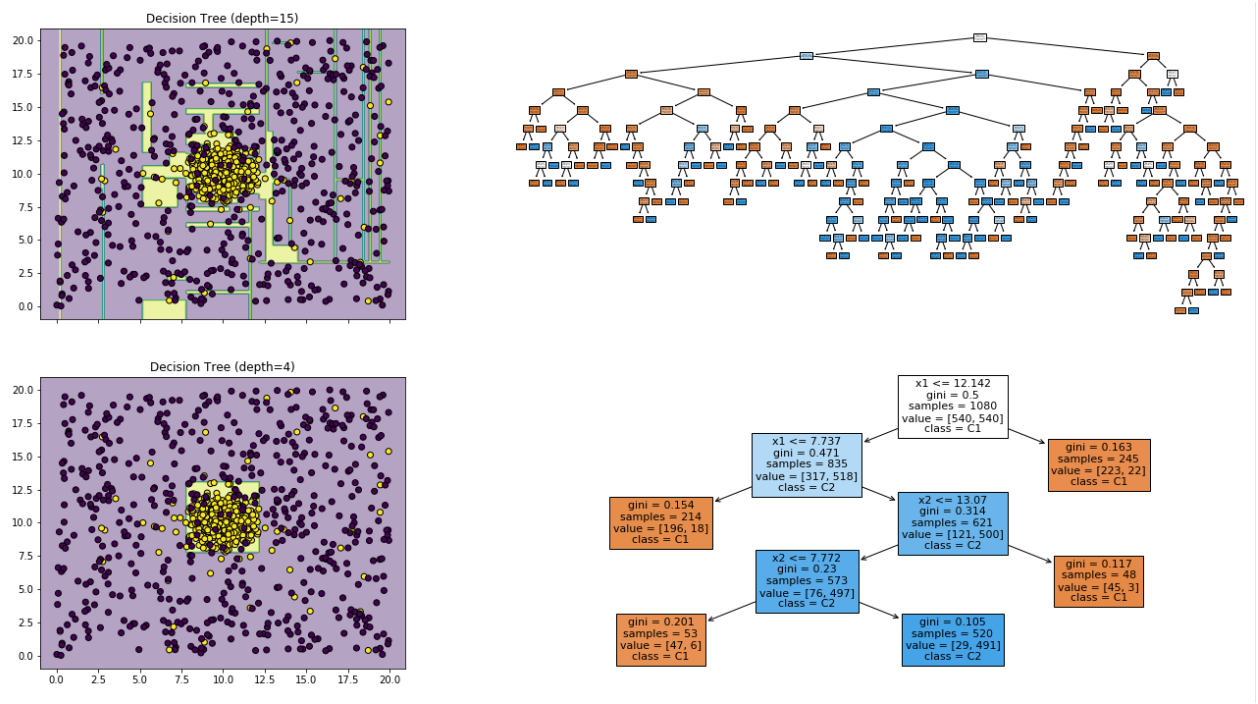


Figure (16.): **Top**: Overfitting of decision trees on the noisy Gaussian data (yellow points). **Top Left**: DT with lots of branching's to accommodate the noise that we added to the Gaussian data. **Top Right**: the decision boundaries of the DT shows how the tree is trying to isolate pockets of data to decrease the training error. **Bottom:** Decision trees with no overfitting for the same noisy Gaussian data. **Bottom Left**: Overfitting is solved by reducing the maximum depth and confining the splitting of leafs to a minimum of around 240 points for 10% (pro rata) of the 5400x2 data points. Note that this is problem specific and it shows that it is hard to overcome overfitting specifically with DT. **Bottom Right**: DT ignores the noise in the Gaussian data and just isolate the Gaussian data in a rectangle.

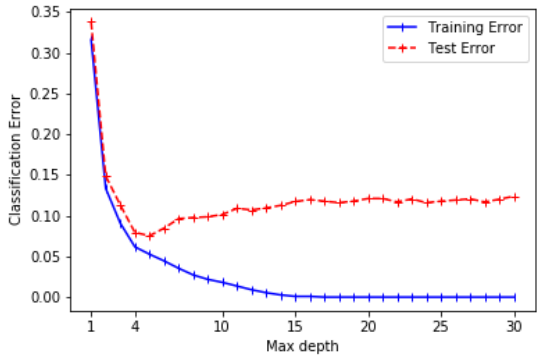


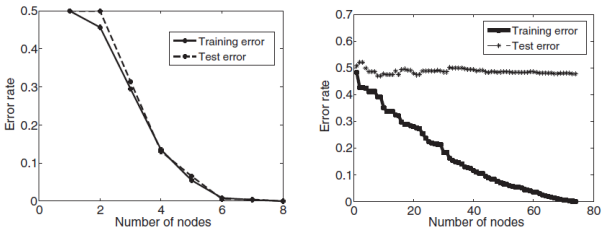
Figure 5.13: Decision trees training and testing with phenomenon of overfitting. Note how when we increase the max depth of the tree the testing error forked from the training error which continued to deceivingly decrease, while in reality the testing error were increased for depth > 4.

The figure above is important to show the signs or symptoms of overfitting. As it can be seen the training error is successfully decreased when we overly grow the tree, however the testing error (the more precise indicator of the generalisation ability of the model) has actually remained more or less the same. The elbow shape of the testing error is a clear indicator of overfitting and the reasonable size of the tree lies exactly around the elbow (angle) itself. So for this example the angle lies around on 4 (the number of conditions/nodes required to classify the data).

5.3.2 Exercise:

See the following Jupyter notebook exercise on some of DT Strengths limitations and overfitting.

On the other hand, overfitting can occur when we excessively add data horizontally. In other words if we increase the number of attributes that are not really needed to make a decision then potentially the tree will over grow and the training error will be reduced without reducing the testing error. So the symptoms of overfitting are the same but the underlying cause is different. In the first the data is noisy in the second the attributes are unnecessary. The figure below shows this phenomena.



(a) Using X and Y attributes only.

(b) After adding 100 irrelevant attributes.

Figure 3.27. Training and test error rates illustrating the effect of multiple comparisons problem on model overfitting.

5.4 UNDERFITTING

Underfitting occurs when we build a decision tree that is incapable of addressing the problem sufficiently. An example of underfitting for decision trees is when we prematurely stop growing the tree on level 2 while the optimal number of levels is 5. Another example is when the tree is severely pre-pruned to have few branches and its performance becomes inadequate.

5.5 WRAPPING EXERCISE:

Please perform the following Jupyter notebook exercise on the iris dataset that wraps up the unit.

5.6 EXERCISE

Now see how to do the same thing as above in RapidMiner, it is fun and easier.

LESSON SUMMARY:

1 In this lesson we have covered the common problems of overfitting and underfitting, and seen how we can detect these problems and address them. We also used cross validation to come up with an accurate measure for the performance of our model and its prediction ability

2 We have seen how the elbow method can provide an effective way to detect overfitting, and we can use a simple early stopping technique to overcome it.

Groups Discussion: On the challenges of model selection, model evaluation and Model comparison

Topics to focus on:

- Model evaluation
- Model selection
- Model comparison

Preparation: student should read the whole unit and particularly focus on the above topics. Reading sections 3.5-3.9 of Tan et al. (2019) will be immensely helpful.

Student should discuss:

- Why we need a separate testing set from the validation and training set.
- Students should come up with their own interpretation of the issues surrounding the partitioning and training/validation/testing of a dataset.
- Indirect influence of model selection on the testing set and how this can make the model evaluation biased.
- Leaking info into the test set.

6 REFERENCES

For unit and other testing see this tool as an example.

How often an algorithm produces optimal tree.

Lots of brilliant code from scikit learn classifiers. (Decision Trees, KNN and perceptron etc.)

Regressors that are resilient to outliers beyond what we covered in unit 3 here.

Datasets: Boston Dataset for regression here, Diabetes dataset here, mnist

British Cattle Vetinary Association, 2011. Decision making tree for bTB testing during COVID 19 restrictions. Available online at: https://www.bcva.org.uk/system/files/whatwedo/BCVA%20TB%20Flowchart%20England%20v7%2011.05.2020.pdf [accessed 26/02/2021]

Han L, Wang Y, Bryant SH. 2008. Developing and validating predictive decision tree models from mining chemical structural fingerprints and high-throughput screening data. *BMC Bioinformatics*. 2008;9:401. Published 2008 Sep 25. doi:10.1186/1471-2105-9-401

NHS Northern Care Alliance, 2020. Decision tree to accompany individual staff/volunteer risk assessment for individuals who may need additional control measures put into place as a result of the risk of COVID 19. Available online at: https://www.pat.nhs.uk/Coronavirus/HR/Decision%20Tree%20for%20COVID-19%20individual%20risk%20assessments.pdf [Accessed 26/01/2021]

MARC DE KAMPS

MACHINE LEARNING - UNIT 5 (v1.0)

UNIVERSITY OF LEEDS

Contents

List of Figures

List of Tables

1 Introduction

In this Unit, we will investigate so-called mixture models. In mixture models it is assumed that a data point can be generated by any of a number of stochastic processes. An often used example is a mixture of Gaussians, but mixtures of other models are also possible, e.g. mixtures of Bernoulli processes.

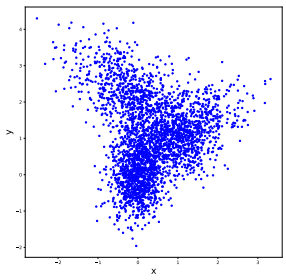


Figure 1.1: An example of a dataset for which a mixture of Gaussians can be considered.

These mixtures are very important because they allow us to model complex multimodal distributions, which we often cannot handle analytically in terms of a collection of Gaussian processes for which we have familiar tools available. Here, an interesting problem arises: if we have a complex distribution modelled as a mixture of Gaussians and we have a dataset that looks like it has a set of clusters (have a look at Fig. 1.1 as an example), then how do we delineate the clusters and how do we infer their parameters? Of course, we must infer the parameters of the Gaussians that are supposed to make up the cluster. We know how to do that for a single Gaussian since we can do maximum likelihood estimation. But here we also must somehow infer how much each individual Gaussian contributes to the data cloud, or more precisely: we must infer the probability that a given data point comes from a certain cluster, as well as estimating the cluster properties themselves, and ideally, do this simultaneously. In order to even start that process, we have to make a guess about K, the number of clusters.

In Fig. 1.1 it looks like $K = 3$ is a reasonable choice, but in a data

space with more than two dimensions we may struggle to visualise the complete dataset and the number of clusters may not be readily estimated. This is an important problem, which we shelf for now. We assume that we have determined the number of clusters, K, that we want to use to model our data, which lives in a space of dimension D, $D = 2$ in this case.

We can represent the probability that a given data point $\boldsymbol{x}$ belongs to cluster j by a K-dimensional vector $(\pi_1, \cdots, \pi_K)^T$. So the probability that a given data point belongs to a certain cluster is a *stochastic variable*. The problem is that we are not observing that variable: we are not given a label that tells which cluster this data point belongs to. A stochastic variable whose realisation cannot be observed but has to be inferred through indirect means is called a *latent variable*. What we will encounter in the Unit is probability distribution functions that are written as $(\boldsymbol{X}, \boldsymbol{Z})$, which is a joint probability over variables X, whose outcome can be observed directly and Z, whose outcomes cannot be observed.

Inferring this joint distribution is clearly much harder to do than the kind of inference in Unit 1, where we had to estimate the parameters of known distributions, but where we were able to produce estimators, either maximum likelihood estimators or posterior distributions that made direct use of the data observations. Here, in general we have to proceed in an iterative process that consists of two steps: the first step is to make a good guess about the distribution of the latent variables. Having numerical values for the latent variable, we can proceed with conventional methods for estimating the observable variables. We then repeat and try to use these estimates to make better estimates for the latent variables, and so on. This leads to a class of algorithms called *E/M algorithms*. We will investigate this algorithm in detail for the *Gaussian Mixture Model* (GMM).

The presence of latent variables is not just a problem. Some of the recent progress in AI and machine learning has come from the approach to look at complex high dimensional data, like images or large volumes of text, as being 'caused' by simpler underlying processes. These methods try to use a large number of observations to estimate the joint probability $p(X, Z)$ where X is high dimensional (think images) and Z much smaller. The aim becomes to learn $p(Z|X)$. In practice, this is so hard that one tries to create a model distribution $q(Z \mid X)$ that is a good approximation. If this process is successful, one has access to the latent distribution $p(Z)$. This process is called *encoding* and we will see that GMMs are a very rudimentary form of this. Encoding is useful in its own right. It has been used successfully in *image compression*.

Modern techniques go one step further. Learning the mapping

$p(X \mid Z)$ is called decoding. If one successfully learns this mapping, it becomes possible to generate synthetic images like that resemble ones from the original dataset from samples that have been generated in the latent space.

The theoretical framework underpinning all of these methods is called *variational inference*. In variational inference the objective is to approximate the true latent variable distribution $p(Z \mid X)$, which is often intractable by an estimate $q(Z)$ which has a simpler functional form. To monitor whether this training is successful, we need something called the *Evidence Lower Bound* (ELBO). The theory of variational inference is outside of the scope of this module, but you will be able to gain some experience using them in Activity 5.B.

The purpose of this chapter is to provide:

1. A rigorous introduction to GMMs and E/M algorithms.
2. An introduction to the concepts ELBO and *posterior latent variable distribution*.
3. A brief overview of where in modern machine learning they feature.
4. Pointers to the literature where more technical expositions of the variational framework are available.

A full discussion of the variational framework is well outside the scope of this module but a gentle introduction to some of its concepts is useful for modern machine learning.

2 Mixtures of Gaussians

2.1 K-Means as a Special Case of Mixtures of Gaussians

In the *Data Science* module the K-Means algorithm was introduced. We will discuss it here briefly to set the stage of Gaussian mixture models. Once we have discuss those, we will see that K-Means is a special case.

The K-Means algorithm assumes that a dataset can be partitioned into K clusters, where K is a number that has to be determined by the data analyst. Once the algorithm is run, each data point has been assigned to a cluster and a mean position for each cluster has been calculated.

The algorithm is straightforward. Assuming there are N data points $\boldsymbol{x}_i$ in a D-dimensional space, a one-over-K coding will indicate to which of the clusters the point belongs. This is recorded in a matrix a $N \times K$, $\boldsymbol{R}$ whose components are r_{ik}, where i, the row index, labels the data point and k the cluster. For a given i, all but one of the r_{ik} are zero, the non-zero component is equal to 1. You can think of this as large matrix with as many rows as there are data points where each row contains a one-over-K coding labelling which cluster the data point belongs to.

The K-means algorithm aims to minimise the *distortion measure*:

$$J = \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \mid\mid \boldsymbol{x}_i - \boldsymbol{\mu}_k \mid\mid^2 \tag{2.1}$$

This represents the sum of squares of distances from each data point to some cluster centre $\boldsymbol{\mu}_k$. Initially, we know neither cluster centres nor cluster assignment. We start out by simply picking random values for the cluster values $\boldsymbol{\mu}_k$. Then we repeat the following two steps until hitting some stopping criterion:

1. Assign each data point to a clusters. This is done by pick $\boldsymbol{R}$ as follows:

$$r_{ik} = \begin{cases} 1 & \text{if } k = \arg\min_j \mid\mid \boldsymbol{x}_i - \boldsymbol{\mu}_j \mid\mid^2 \\ 0 & \text{otherwise} \end{cases}$$

2. Estimate the new cluster centres as follows:

$$\boldsymbol{\mu}_k = \frac{\sum_i r_{ik} \boldsymbol{x}_i}{\sum_i r_{ik}}$$

A stopping criterion can be defined by demanding that J be below a certain value, or by visual inspection of the resulting clusters when possible.

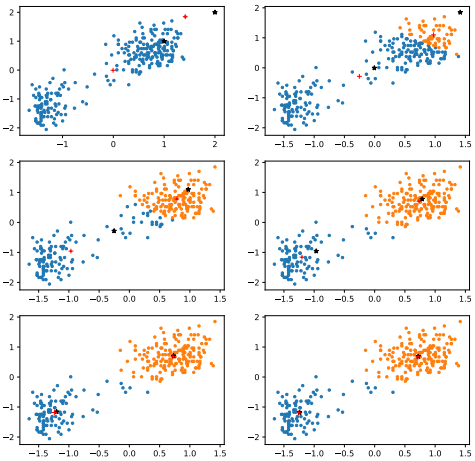


Figure 2.2: The K-Means algorithm on the Old Faithful dataset. Despite the unfortunate choice for the starting values of the centres, the algorithm converges in approximately 5 steps, leaving two well separated clusters. The black marker represents the centre estimate for each cluster at the start of the algorithm. The red crosses the new position at the end of each step.

In Figure 2.2 we show the convergence of the algorithm on the so-called *'Old-Faithful'* dataset.

On separated clusters this algorithm works fairly well. Importantly, it already displays the two-step approach that characterises E/M algorithms, as we will see later.

One disadvantage of this algorithm is that we need a good idea about how many clusters there are. If we assume the wrong number we can create extra clusters that are not really separate. Another disadvantage is that when we know that clusters overlap this algorithm cannot respect the overlap because points are assigned to some cluster based on a notion of distance to some centre and therefore partition boundaries. This can happen when different processes can generate data samples that overlap in data space. The K-means algorithm has to make a hard assignment of a point to each cluster, which can deform the cluster and affect its estimates (see *Activity 5.A*).

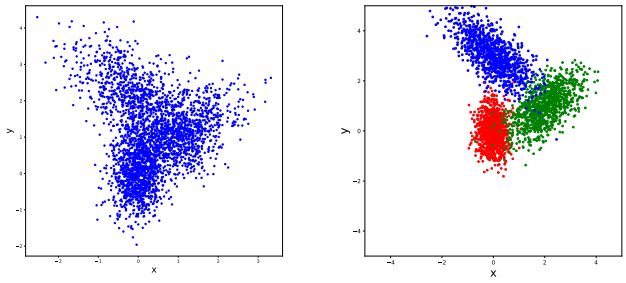


Figure 2.3: The plot on the left shows a situation that occurs often in practice. The data appears to be clustered but without any labels it is not straightforward to assign points to clusters and to calculate the cluster properties. When we do show the labels, as in he right plot, it is clear that different clusters can inhabit the same part of data space, which the standard K-means clustering does not recognise.

2.2 *Mixtures of Gaussians*

A mixture of Gaussian provides a good example of clusters that inhabit the same part of data space. In a mixture of Gaussians, a data point can be created by any number of Gaussian distributions. To each Gaussian, we can assign a probability that indicates how likely a particular Gaussian generates the next data point. Out of these Gaussian a single one is selected, which is then sampled subsequently to generate the actual data point. Figure 2.3 shows and example of a dataset that has been generated this way.

In order to model such a data generation process, we need a generalisation of the Bernoulli process introduced in Unit 1. We assume that there is a K-dimensional vector $\boldsymbol{x}$, where one of the elements x_i has the value one and the other elements are 0. We have already encountered this as the 1-over-K coding scheme. In order to model it, we need a multidimensional version of the Bernoulli process. We define a set of K probabilities, μ_k, which by virtue of them probabilities must add to one:

$$\sum_k \mu_k = 1$$

The probability to generate a particular 1-over-K pattern $\boldsymbol{x}$ is given by:

$$p(\boldsymbol{x} \mid \boldsymbol{\mu}) = \Pi_{k=1}^{K} \mu_k^{x_k} \tag{2.2}$$

The way to read this is that if we want to know the probability, given $K = 5$, what the probability for $(0, 1, 0, 0, 0)$ to occur is that the only $x_k \neq 0$ is $x_2 = 1$. The probability for this particular pattern is therefore μ_2, in other words Eq. 2.2 is a succinct mathematical expression of what we just explained in words. Note that because one and only one of the $x_k = 1$, rather trivially it follows that $\sum_{k=1}^{K} x_k = 1$.

This distribution is properly normalised:

$$\sum_{\boldsymbol{x}} p(\boldsymbol{x} \mid \boldsymbol{\mu}) = \sum_{k=1}^{K} \mu_k = 1$$

and that the expectation value is given by:

$$\mathbb{E}[\boldsymbol{x} \mid \boldsymbol{\mu}] = \sum_{\boldsymbol{x}} p(\boldsymbol{x} \mid \boldsymbol{\mu})\boldsymbol{x} = (\mu_1, \cdots \mu_K)^T = \boldsymbol{\mu}$$

As with the Bernoulli process, we can write down likelihood for a dataset $\mathcal{D} = \boldsymbol{x}_1, \cdots \boldsymbol{x}_N$, where the $\boldsymbol{x}_i$ indicate N independent observations:

$$p(\mathcal{D} \mid \boldsymbol{\mu}) = \Pi_{i=1}^{N} \Pi_{k=1}^{K} \mu_k^{x_{ik}} = \Pi_{k=1}^{K} \mu_k^{\sum_i x_{ik}} = \Pi_{k=1}^{K} \mu_k^{m_k}$$

Here

$$m_k = \sum_i x_{ik},$$

i.e. the number of observations of $x_k = 1$.

We can find the maximum likelihood estimator (MLE) for $\boldsymbol{\mu}$ by maximising this expression, subject to the condition that $\sum_k \mu_k = 1$. You may or may not be familiar with the technique of Lagrangian multipliers, which is the appropriate technique for finding constrained optima. If you have not seen this you can just take the result at face value; we will not assess you on this. If you are curious, a brief exposition is given in appendix E of [1].

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

The technique consists of multiplying the constraint by an arbitrary factor, the so-called multiplier and then adding to the quantity whose extremum we want to find. Again, we work with the log likelihood which means that we want to maximise the quantity:

$$\sum_{k=1}^{K} m_k \ln \mu_k,$$

subject to $\sum_{k=1}^{K} \mu_k = 1$. Adding the constraint term we get:

$$\sum_{k=1}^{K} m_k \ln \mu_k + \lambda(\sum_{k=1}^{K} \mu_k - 1)$$

Optimising with respect to μ_j gives:

$$\mu_j = -\frac{m_j}{\lambda}$$

Substituting this into the constraint that all μ_j must add to one, we find that:

$$-\sum_k \frac{m_k}{\lambda} = 1,$$

or $\lambda = -N$ so that ultimately we find the common sense result:

$$\mu_k = \frac{m_k}{N}$$

In passing we mention that for the multidimensional Bernoulli distribution a generalised version of the Beta distribution exists, which

is called the *Dirichlet* distribution. One would use this distribution as a prior for Bayesian inference, see section 2.2.1 of [2] for details.

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

We are now ready to present the GMM model: We can interpret this as a generative model: first we use a Bernoulli process to select a cluster i, then we sample from the Gaussian with mean $\boldsymbol{\mu}_i$, covariance matrix $\boldsymbol{\Sigma}_i$.

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{2.3}$$

We can also interpret this as a joint distribution over a space of observable variable X and a space of latent variables Z, as follows:

$$p(\boldsymbol{z}) = \Pi_{k=1}^{K} \pi_k^{z_k}, \tag{2.4}$$

where you should remember that all z_k are zero except for one. The conditional distribution is Gaussian:

$$p(\boldsymbol{x} \mid z_k = 1) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

or, more condensed:

$$p(\boldsymbol{x} \mid \boldsymbol{z}) = \Pi_{k=1}^{K} \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}, \tag{2.5}$$

and since:

$$p(\boldsymbol{x}) = \Pi_{k=1}^{K} \pi_k^{z_k}$$

we can write down the joint probability distribution:

$$p(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \Pi_{k=1}^{K} \pi^{z_k} \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_k}, \tag{2.6}$$

where we have introduced

$$\boldsymbol{\mu} = (\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_K)^T$$

and

$$\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_1, \cdots \boldsymbol{\Sigma}_K)^T$$

We are now able to interpret Eq. 2.3 as the marginalisation of Eq. 2.6:

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{z}} p(\boldsymbol{z}) p(\boldsymbol{x} \mid \boldsymbol{z}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

This shows that it makes sense to define a latent variable $\boldsymbol{z}_i$ with each observable variable $\boldsymbol{x}_i$ and that we can write down a joint distribution over both sets of variables. It may not yet be clear why this is useful, given that the GMM Eq. 2.3 does not apparently involve them, but their role will become clear later.

It will also turn out to be useful to introduce $p(z_k = 1 \mid \boldsymbol{x})$. It is sufficiently important to receive its own symbol:

$$\gamma(z_k) \equiv p(z_k = 1 \mid \boldsymbol{x}) \tag{2.7}$$

and is called the *responsibility* that component k takes for explaining data point $\boldsymbol{x}$. Intuitively, this is related to the distance of a data point to the other points of that cluster. This can be made precise through Bayes' Theorem, as it can be related to $p(\boldsymbol{x} \mid z_k = 1)$ and therefore:

$$\begin{aligned}\gamma(z_k) = p(z_k = 1 \mid \boldsymbol{x}) &= \frac{p(z_k = 1)p(\boldsymbol{x} \mid z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1)p(\boldsymbol{x} \mid z_j = 1)} \\ &= \frac{\pi_k \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}\end{aligned} \tag{2.8}$$

We can interpret π_k as a prior probability and $\gamma(z_k)$ as the posterior probability that a given data point $\boldsymbol{x}_k$ belongs to cluster k.

2.3 Maximum Likelihood Estimation

2.3.1 The Exponential Family

In Activity 1.6 you have seen that the likelihood is a product of exponentials but that the log likelihood leads to a sum of quadratic terms. The exponential of the Gaussian is 'eaten' by the logarithm when considering the log likelihood and reduces to sum of quadratic terms, which is easy to handle. Something similar happened to the Bernoulli process although we did not make this explicit. We will do that here. Remember that the Bernoulli process is given by:

$$p(x \mid \mu) = \text{Bern}(x \mid \mu) = \mu^x (1 - \mu)^{1-x}$$

This can be written as

$$p(x \mid \mu) = (1 - \mu) \exp \ln \frac{\mu}{1 - \mu} x$$

In this form, it is clear that a data point again is a product of exponentials (and other factors) and the logarithm reduces to a simple sum. Also, remember that in Unit 2 we studied the loss function that emerged in logistic regression. Again, this likelihood function was a product of (class conditional) Gaussians. There is an entire family of distributions the *exponential family* for which the log likelihood is a much simpler expression because it get rids of the exponential. Bernoulli, Multinomial and Gaussian distributions are examples of this family and together they span such a large part of statistics that taking the logarithm becomes second nature.

It is recommended that you read section 2.4 of [3], which provides a comprehensive overview of the exponential family.

The key point we want to make here is that even when you start out with a joint distribution that has exponential form that as soon as you marginalise, this no longer is necessarily the case. We will

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

perform maximum likelihood estimation (MLE) for Eq. 2.3, which we have seen can be considered to be a marginalisation over latent variables. We will see that although we will employ the same machinery as in Unit 1, the process is harder because the sum term in Eq. 2.3 prevents the logarithm from cancelling the exponentials. We will go through parts of the calculation to make this point explicit. Later, we will show there is an easier way to do this.

Again, like in Unit 1 we will not assess you on being able to perform these calculation yourself. But if you want to get to the point where you are able to access the machine learning independently, you should try and follow the logic here.

2.3.2 *Maximum Likelihood Estimation for the Gaussian Mixture Model*

In this section we will perform maximum likelihood estimation of $\boldsymbol{\mu} = (\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_K)^T$, and $\boldsymbol{\Pi} = (\pi_1, \cdots, \pi_k)^T$, and simply list the result for $(\boldsymbol{\Sigma}_1, \cdots \boldsymbol{\Sigma}_K)^T$. We will postpone its calculation until we have found a simpler way.

First we introduce the shorthand notation:

$$\begin{aligned}\boldsymbol{\mu} &= (\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \cdots, \boldsymbol{\mu}_K)^T \\ \boldsymbol{\Sigma} &= (\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \cdots, \boldsymbol{\Sigma}_K)^T\end{aligned} \tag{2.9}$$

As usual we will represent the data by an $N \times D$ matrix $\boldsymbol{X}$, where N is the number of data points and D is the dimension of the data space, with the i-th row given by $\boldsymbol{x}_i$.

$$\ln p(\boldsymbol{X} \mid \boldsymbol{\Pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^{N} \sum_{k=1}^{K} \ln\{\pi_k \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\} \tag{2.10}$$

To ram the point home: there is no easy way to evaluate for example:

$$\ln(2e^x + 5e^{-x})$$

The innocuous looking summation over k in Eq. 2.10 prevents the logarithm from cancelling the exponentials. We have to estimate parameters on the likelihood instead of the log likelihood. We want to calculate:

$$\frac{\partial \ln p(\boldsymbol{X} \mid \boldsymbol{\Pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma})}{\partial \boldsymbol{\mu}_j} = \sum_{i=1}^{N} \frac{1}{\sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_i \, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)} \frac{\partial \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}{\partial \boldsymbol{\mu}_j}$$

We are now forced to use the definition of the Gaussian directly:

$$\frac{\mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})}{\partial \boldsymbol{\mu}} = (2\pi)^{-D/2} \det(\Sigma)^{-\frac{1}{2}} \frac{\partial}{\partial \boldsymbol{\mu}} e^{-\frac{1}{2}(\boldsymbol{x}_i - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}_i - \boldsymbol{\mu})}$$

We have

$$\frac{\partial}{\partial \boldsymbol{\mu}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}-\boldsymbol{\mu})} = -\frac{1}{2} e^{-\frac{1}{2}\{\cdots\}} \frac{\partial}{\partial \boldsymbol{\mu}} [(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu})]$$

For the last differentiation we can employ the matrix identity that we already used in Unit 1:

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

and using the symmetry of $\boldsymbol{\Sigma}(= \boldsymbol{\Sigma}^T)$, we find that:

$$\frac{\partial}{\partial \boldsymbol{\mu}}\mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu}) \tag{2.11}$$

Substituting Eq. 2.11 into 2.10 then gives:

$$\frac{\partial}{\partial \boldsymbol{\mu}_j} \ln p(\boldsymbol{x} \mid \theta) = \sum_{i=1}^{N} \frac{\pi_j \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)} \boldsymbol{\Sigma}_j^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu}_j)$$

Here we are starting to adopt a single symbol, θ, to signify all parameters. This is still a bit unwieldy, so we introduce the symbol:

$$\gamma(z_{ij}) \equiv \frac{\pi_j \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}, \tag{2.12}$$

which happens to be the responsibility for data point $\boldsymbol{x}_i$ (see Eq. 2.8).

The MLE for $\boldsymbol{\mu}_j$ can now be found by:

$$\frac{\partial}{\partial \boldsymbol{\mu}_j} \ln p(\boldsymbol{x} \mid \theta) \mid_{\boldsymbol{\mu}_j = \boldsymbol{\mu}_{ML,j}} = 0,$$

which leads to:

$$\sum_{i=1}^{N} \gamma(z_{ij}) \boldsymbol{\Sigma}_j^{-1}(\boldsymbol{x}_i - \boldsymbol{\mu}_{ML,j}) = 0,$$

which can be easily solved giving:

$$\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{i=1}^{N} \gamma(z_{ij}) \boldsymbol{x}_i, \tag{2.13}$$

with:

$$N_j = \sum_{i=1}^{N} \gamma(z_{ij})$$

A similar exercise can be carried out for the covariance matrix, using similar techniques as in Unit 1, but requiring much more effort. We just state the result:

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(z_{ik})(\boldsymbol{x}_i - \boldsymbol{\mu}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)^T, \tag{2.14}$$

where the $\boldsymbol{\mu}_k$ are the maximum likelihood estimates given above.

Finally, the MLE for $\boldsymbol{\pi}$ need to consider the constraint

$$\sum_{k=1}^{K} \pi_k = 1.$$

This is done by incorporating a Lagrangian multiplier. Again, you are not assessed on knowing this technique, this derivation is given for completeness.

In order to determine $\boldsymbol{\pi}$ we need to maximise:

$$\ln p(\boldsymbol{X} \mid \theta) + \lambda(\sum_{k=1}^{K} \pi_k - 1)$$

Differentiating this quantity is easy and allows us to eliminate λ

$$0 = \sum_{i=1}^{N} \frac{\mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} + \lambda$$

Multiplying both sides by π_k and summing over k while using $\sum_{k=1}^{K} \pi_k = 1$, one finds

$$\lambda = -N$$

Eliminating λ one finds:

$$\pi_k = \frac{N_k}{N} \tag{2.15}$$

Equations 2.13, 2.14 and 2.15 do not provide a closed form solution for the parameters. To see this, consider that the responsibilities depend $\boldsymbol{\mu}, \boldsymbol{\Sigma}$, and $\boldsymbol{\pi}$, but they in turn are used in the MLE of these parameters. The best we can hope for is an iterative scheme. We will present this in the next section.

2.4 *E/M Algorithm for Gaussian Mixture Models*

The Expectation/Maximisation (E/M) algorithm for Gaussian Mixture Models can now be presented as follows:

1. Determine K, the number of clusters. Variants of the E/M algorithm exist that automatically phase out irrelevant clusters if K is chosen too high but in the basic version of the algorithm that we now will discuss a judicious choice for K is important. This may require some experimentation, for example, if the clusters are not of great quality and clustering boundaries appear to be arbitrary it make make sense to try and reduce K.

2. Initialise the means $\boldsymbol{\mu}_k$ and covariances $\boldsymbol{\Sigma}_k$ and mixing coefficients π_k. This can be done by picking random values, or starting with running the K-nearest neighbour algorithm. To focus on essentials, we will assume random initialisation here.

3. **E-step**. Evaluate the responsibilities:

$$\gamma(z_{ik}) = \frac{\pi_k \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \pi_j \mathcal{N}(\boldsymbol{x}_j \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

This step evaluates for each data point $\boldsymbol{x}_i$ how much responsibility cluster k assumes for explaining this data point. In actual realisation each data point belongs to one and only one cluster, but the higher $\gamma(z_{ik})$, the more likely the data point belongs to cluster k. Given that this step entails the calculation of NK numbers, this is usually the most time consuming step of the algorithm.

4. **M-step**. Re-estimate the parameters using the responsibilities:

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k}\sum_{i=1}^{N}\gamma(z_{ik})\boldsymbol{x}_i \tag{2.16}$$

$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N_k}\sum_{i=1}^{N}\gamma(z_{ik})(\boldsymbol{x}_i - \boldsymbol{\mu}_i^{new})^T(\boldsymbol{x}_i - \boldsymbol{\mu}_i^{new}) \tag{2.17}$$

$$\pi_k = \frac{N_k}{N} \tag{2.18}$$

with

$$N_k = \sum_{i=1}^{N}\gamma(z_{ik}) \tag{2.19}$$

5. Evaluate the log likelihood:

$$\ln p(\boldsymbol{X} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{i=1}^{N}\ln\left\{\sum_{k=1}^{K}\pi_k\mathcal{N}(\boldsymbol{x}_i\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right\} \tag{2.20}$$

Compare this to the previous evaluation of the log likelihood for convergence. If not satisfied repeat from step 3 onwards.

2.4.1 *Numerical Stability*

An unlucky choice of initial parameters may cause numerical instability of the algorithm. If cluster centre $\boldsymbol{\mu}_l$ is chosen that nearly coincides with one of the data points, the algorithm can maximise the log likelihood by shrinking the variance of cluster l, whilst moving $\boldsymbol{\mu}_l$ even closer to the data point. A detailed explanation is given in Section 9.2.1 of [4]. There are various ways to avoid this problem, the easiest is by starting the cluster a number of times from different random values, something that is prudent anyway. To avoid this particular problem, you could pre-check whether the $\boldsymbol{\mu}_i$ coincide with any data points. A Bayesian version of this algorithm would not suffer from this problem, which is typical of problems with maximum likelihood estimation, but a full Bayesian treatment of the algorithm is hard. However, a so-called *variational inference* approach is available. In *Activity 5.B* we will employ *scikit-learn* and you will find that a K-means clustering available as a *GaussianMixture* object and the variational version is available as a *BayesianGausianMixture* object.

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

At the expense of setting more input parameters, which specifies a *prior* on the mixture components and increased computing time, you will have a stabler algorithm, which also regularises away irrelevant clusters, i.e. if the choice for K is unfortunate and too large, the variational variant will produce some clusters with very small mixture components that can be ignored in subsequent analysis.

The subject of variational inference is too technical to consider in this unit, but the *Evidence Lower Bound* or *ELBO* plays a central role in this technique. We will discuss the ELBO below.

2.5 GMMs in Practice

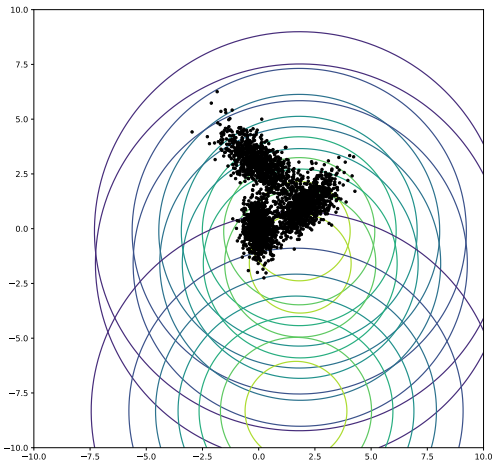


Figure 2.4: The E/M algorithm in action. At the start random values determine the position of the mean of Gaussian clusters, whereas the covariances have been chosen to be isotropic with the same parameter (isotropic means that no direction is preferred, therefore the clusters appear circular). The clusters are indicated by isoprobability contour lines.

In Fig. 2.4, you can see the same dataset that was shown in in Fig. 2.3, as well as the initial guess for three Gaussian clusters. Isoprobability lines are shown for each of these clusters. In Fig. 2.5, you can see the algorithm in action and observe its performance on each subsequent step. In Fig. 2.6, you can see the end result, the placement of the clusters as well as the responsibility that each cluster takes for each data point. Since we have three clusters, we can use RGB codes hat add to one in this case. When there are more or less clusters this kind of visualisation does not work. An alternative could be to use an orientation for each cluster and use saturation level to express the numerical value of the responsibility.

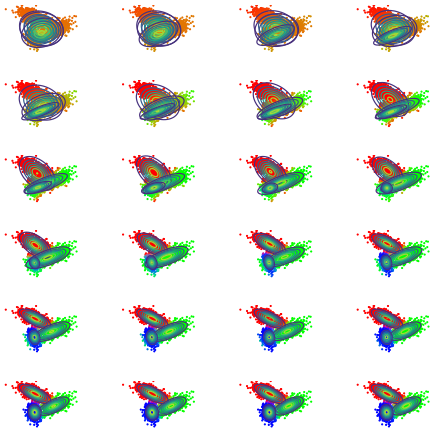


Figure 2.5: As the algorithm progresses, the position of the Gaussian estimates are adapted. The points of the dataset are coloured according to the values of the responsibility in each step, with points mainly associated with cluster 1, being mostly red, cluster 2 green, etc. This colour scheme, which uses RGB values that must sum to one works nicely here, because there are three clusters.

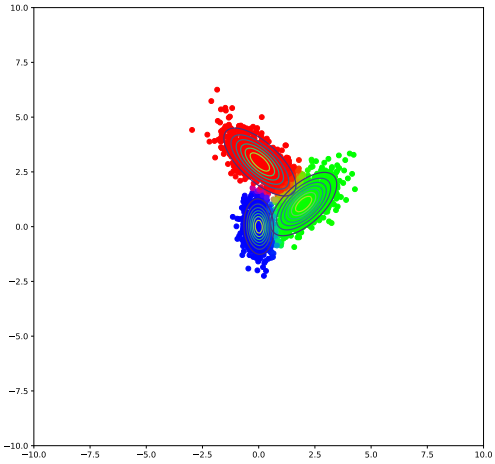


Figure 2.6: The final result in close up. Note that the clusters can be seen to overlap. This is also visible in the responsibilities, although the cluster that is closer usually dominates. Note that the covariant structure is captured well. We started with three isotropic clusters, but the algorithm has found covariance matrices that represent the data well.

2.6 *The E/M Algorithm in General*

The E/M algorithm provides a satisfactory solution for the GMM, and as we will see for other mixture models as well. However, at this stage it may not be clear why the algorithm converges. our treatment has been entirely heuristic. Also, it may not be clear what the advantage is of considering the latent variables z. They feature very indirectly in our estimates of the mixing coefficients π but they do

not seem to be essential.

We will show in this section that by using latent variables the same problem becomes more tractable. It will lead to the same algorithm, so at this stage it may still not be entirely clear that latent variables bring a crucial advantage, but in the next section we will show that using this approach we can proof convergence of the algorithm, and that this proof applies very generally, not just to GMMs. It also sets the stage for a Bayesian approach to mixture models.

In the previous section, we have seen that the maximum likelihood estimation of a marginalised distribution is hard: we tried to maximise

$$\ln p(\boldsymbol{X} \mid \theta) = \ln \sum_{\boldsymbol{Z}} p(\boldsymbol{X}, \boldsymbol{Z})$$

on the marginalised probability, i.e. with the summation already performed and found that the logarithm no longer works on the distributions directly. It is generally true that marginalised probabilities, and this includes conditional probabilities, are much harder to handle.

In Unit 4 you will have seen that complex joint probability distributions can be built from networks where the edges represent conditional probabilities. Evaluating these joint probability distribution functions is cheap, and if necessary, they can be sampled with relative ease. To answer interesting questions, however, we want to marginalise over most variables. Or we want to answer questions as: 'where is 90 % of our probability mass located?'. This requires integration in a multi-dimensional space which is typically analytically intractable and numerically very expensive.

So, whilst $p(x_1, x_2, x_3, x_4, x_5, x_6)$ may be easy to evaluate,

$$p(x_1, x_3) = \int dx_2 dx_4, dx_5, dx_6 p(x_1, x_2, x_3, x_4, x_5, x_6)$$

is not and neither is:

$$p(x_2, x_3, x_4 \mid x_1, x_5, x_6) = \frac{p(x_1, x_2, x_3, x_4, x_5, x_6)}{\int dx_1 dx_5 dx_6 p(x_1, x_2, x_3, x_4, x_5, x_6)}$$

This is possibly counter intuitive: all knowledge is contained in the joint probability distribution. However, when it comes to extracting numerical quantities from them, the rule of the thumb is: $\sum_Z p(\boldsymbol{X}, \boldsymbol{Z})$: *bad*, $p(\boldsymbol{X}, \boldsymbol{Z})$: *good*.

Let us write down the joint probability distribution for a GMM in terms of both the observable variables $\boldsymbol{X}$, and $\boldsymbol{Z}$:

$$p(\boldsymbol{X}, \boldsymbol{Z} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{i=1}^{N} \pi_k^{z_{ik}} \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)^{z_{ik}}$$

Remember that $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots \boldsymbol{x}_N\}$ is a set of N points in data space and that $\boldsymbol{Z} = \{z_1, \cdots, z_N\}$ is a set of N K-dimensional vectors

which have all but one element equal to zero, which indicates to which cluster each data point belongs. Also remember that we do not have access to these vectors, as they are unobserved.

After taking the logarithm:

$$\ln p(\boldsymbol{X}, \boldsymbol{Z} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi}) = \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} \{\ln \pi_k + \ln \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\} \tag{2.21}$$

Here z_{ik} is the k-th component of $\boldsymbol{z}_i$.

Compared to Eq. 2.10 we see that the summation over k and the logarithm are reversed. The logarithm now directly operates on the exponential.

Suppose now, someone would label the data, i.e. gives us access to the variables $\{\boldsymbol{Z}\}$. We could then sort the terms in Eq. 2.21 such that all terms that have $z_{i1} = 1$ (which implies all other components are 0) come first, then all terms with $z_{i2} = 1$ etc., i.e., with access to the labels we would be able to sort the terms in the order of which cluster they appear. But this would amount to a sum of K independent mixtures. We would be able to perform MLE on each mixture individually, and obtain K estimates for $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$.

This is not surprising, but the full joint probability distribution shows the relationship between the labelled case, i.e. where we are given which cluster each data point belongs to and the unlabelled case where the $\boldsymbol{z}_i$ are not observed. In the marginalised distribution, Eq. 2.10, this relationship is not visible.

Of course, we have to deal with the case where the $\boldsymbol{z}_i$ are unobserved. We do, however, have estimates for the posterior distribution of the latent variables through Eq. 2.8 and we can consider the expectation of the complete data log likelihood with respect to the posterior distributions of the latent variables.

To do this requires careful interpretation of the notation, but is not hard. First of all we note that the latent variable distribution is given by Eq. 2.4:

$$p(\boldsymbol{z}) = \Pi_{k=1}^{K} \pi_k^{z_k}.$$

and that the conditional probabilities are Gaussian as per Eq. 2.5

$$p(\boldsymbol{x} \mid \boldsymbol{z}) = \Pi_{k=1}^{K} \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k).$$

Bayes' rule now gives us an expression for the posterior distribution of latent variables:

$$p(\boldsymbol{z} \mid \boldsymbol{x}) \sim p(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) \tag{2.22}$$

This becomes a bit abstract; a simple example will help in interpreting this equation. Imagine we have $K = 3$, then there are three

different vectors z_i, specifically:

$$\begin{aligned} z_1 &= (1,0,0)^T \\ z_2 &= (0,1,0)^T \\ z_3 &= (0,0,1)^T \end{aligned} \tag{2.23}$$

Equation 2.22 states that:

$$\begin{aligned} p((1,0,0)^T \mid x) &\sim \pi_1 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \\ p((0,1,0)^T \mid x) &\sim \pi_2 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) \\ p((0,0,1)^T \mid x) &\sim \pi_3 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_3, \boldsymbol{\Sigma}_3) \end{aligned}$$

All we have to do to find the actual probabilities is to normalise:

$$\begin{aligned} p((1,0,0)^T \mid x) &= \frac{\pi_1 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)}{\pi_1 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) + \pi_2 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) + \pi_3 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_3, \boldsymbol{\Sigma}_3)} \\ p((0,1,0)^T \mid x) &= \frac{\pi_2 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)}{\pi_1 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) + \pi_2 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) + \pi_3 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_3, \boldsymbol{\Sigma}_3)} \\ p((0,0,1)^T \mid x) &= \frac{\pi_3 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_3, \boldsymbol{\Sigma}_3)}{\pi_1 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) + \pi_2 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) + \pi_3 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_3, \boldsymbol{\Sigma}_3)} \end{aligned} \tag{2.24}$$

It is easy to see that in the general case

$$p(\boldsymbol{z} \mid \boldsymbol{x}) = \frac{\Pi_{k=1}^K (\pi_k \mathcal{N}(x \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))^{z_k}}{\sum_{j=1}^K \pi_j \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \tag{2.25}$$

Here the z_k are the components of the vector $\boldsymbol{z}$. The result should make sense. What Eq. 2.24 expresses is that it is more likely to belong to a particular component if you're closer to it and further removed for the others.

We are almost set. We will need one more building block and that is the expectation value of the vector $\boldsymbol{z}$ with respect to the posterior latent distribution. Again, an example will clarify that this is a very simple concept. By definition the expectation value is given by:

$$\begin{aligned} \mathbb{E}_z[\boldsymbol{z}] &= \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} p(\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \mid \boldsymbol{x}) + \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} p(\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \mid \boldsymbol{x}) + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} p(\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \mid \boldsymbol{x}) \\ &= \begin{pmatrix} \frac{\pi_1 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)}{\sum_{j=1}^K \pi_j \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \\ \frac{\pi_2 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)}{\sum_{j=1}^K \pi_j \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \\ \frac{\pi_3 \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_3, \boldsymbol{\Sigma}_3)}{\sum_{j=1}^K \pi_j \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \end{pmatrix} \end{aligned} \tag{2.26}$$

Note that the components of this expectation are nothing but the responsibilities defined in Eq. 2.8, in other words:

$$\mathbb{E}_{\boldsymbol{Z}}[\boldsymbol{z}]_k = \gamma(z_k) \tag{2.27}$$

So far, this may look like wrapping relatively obvious facts in highly abstract notation, but these exercises allow us to evaluate the expectation value of the log likelihood with respect to the posterior latent variable distribution in a straightforward way. Consider the full data log likelihood Eq. 2.21 and taking the expectation value, the result is simple:

$$\mathbb{E}_{\boldsymbol{Z}}[\ln p(\boldsymbol{X}, \boldsymbol{Z} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})] = \sum_{i=1}^{N} \sum_{k=1}^{K} \gamma(z_{ik}) \left\{ \ln \pi_k + (\boldsymbol{x}_i - \ln \mathcal{N}(\boldsymbol{x}_i \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\} \tag{2.28}$$

Note how our work leading up to Eq. 2.27 pays off: the z_{ik} in Eq. 2.21 are the only dependants on $\boldsymbol{z}$.

In *Activity 5.c*, you will be asked to perform maximum likelihood estimation with the aid of Eq. 2.28. You will note that this is much simpler and that you can reuse the results of Unit 1. This is an important example of how joint distributions are easier to handle than marginal ones.

The approach in this section shows why the algorithm is called the Expectation/Maximisation algorithm. First, the *expectation* of the joint probability is calculated. The likelihood of the resulting distribution is then *maximised*. The two step approach that was already recognisable in the K-means clustering is clearly present, but now better motivated. In Sec. 2.7, we will show that this approach always leads to an increased likelihood of the model. So far, this is not obvious. In the M-step, the likelihood will increase by definition, but the E-step, taking the expectation with respect to the posterior latent variables might undo the previous optimisation step in theory. We will see this is not the case. The manner of how this is shown is very important to modern machine learning.

Finally, we mention in passing that to calculate that the likelihood of the posterior distribution of latent variables is simply the product

2.7 *The Evidence Lower Bound (ELBO)*

The argument we will present here is abstract but simple. The main problem in latent variant models such as the GMM is to estimate the distribution of the latent variables $p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta)$, as this is the one we cannot observe. We usually are forced to assume a distribution $q(\boldsymbol{Z})$, which *hope* will be close to the true distribution of latent variables. We will show that for any such $q(\boldsymbol{Z})$ the following decomposition is valid:

$$\ln p(X \mid \theta) = \sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln \left\{ \frac{p(\boldsymbol{X}, \boldsymbol{Z})}{q(\boldsymbol{Z})} \right\} - \sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln \left\{ \frac{p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta)}{q(\boldsymbol{Z})} \right\} \tag{2.29}$$

We will first show that this is true. From Unit 1, we remember the product rule:

$$p(\boldsymbol{X}, \boldsymbol{Z} \mid \theta) = p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta) p(\boldsymbol{X} \mid \theta)$$

and therefore, taking the logarithm:

$$\ln p(\boldsymbol{X}, \boldsymbol{Z} \mid \theta) = \ln p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta) + \ln p(\boldsymbol{X} \mid \theta)$$

We can expand the first term on the right hand side of Eq. 2.29

$$\sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln p(\boldsymbol{X}, \boldsymbol{Z} \mid \theta) - \sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln q(\boldsymbol{Z}) =$$
$$\sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \{\ln p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta) + \ln p(\boldsymbol{X} \mid \theta)\} - \sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln q(\boldsymbol{Z}) \tag{2.30}$$

Adding the second term on the right hand side of Eq. 2.29 cancels most terms. The only remaining contribution is:

$$\sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln p(\boldsymbol{X} \mid \theta)$$

But $\sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) = 1$, because it is a probability distribution and $\boldsymbol{Z}$ does not feature anywhere else. So Eq. 2.29 is true.

We recall the definition of the Kullback-Leibler divergence from Unit 1:

$$\mathrm{KL}(q \mid\mid p) = -\sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln \left\{ \frac{p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta)}{q(\boldsymbol{Z})} \right\}$$

and introduce the so-called *evidence lower bound*:

$$\mathcal{L}(q, \theta) = \sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \left\{ \frac{p(\boldsymbol{X}, \boldsymbol{Z} \mid \theta)}{q(\boldsymbol{Z})} \right\}$$

so that we can write:

$$\ln p(\boldsymbol{X} \mid \theta) = \mathcal{L}(q, \theta) + \mathrm{KL}(q \mid\mid p) \tag{2.31}$$

Despite the abstract notation, some of the actors should look familiar. The goal is to maximise the log likelihood $p(\boldsymbol{X} \mid \theta)$. With the GMM, we have given an example for why this is difficult to do directly and for more complex cases this is generally intractable. In the lower bound, we recognise the expectation of the joint distribution with respect to an estimate for the latent variable distribution $q(\boldsymbol{Z})$, which we found much easier to optimise. The second term on the right hand side, $\mathrm{KL}(q \mid\mid p)$ is a measure for how much q differs from p. Moreover, this measure is only zero when $q = p$ and otherwise positive.

This justifies the name lower bound for $\mathcal{L}$: whatever value of the likelihood $p(\boldsymbol{X} \mid \theta)$ is, it is always higher than the lower bound as the KL term gives adds positively to the lower bound to $\mathcal{L}$, unless $q = p$,

which only happens if estimate $q(\boldsymbol{Z})$ is equal to the true posterior distribution of the latent variable: $p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta)$.

We can now identify the E and M steps in the E/M algorithm as follows. Suppose we have an estimate for the parameters θ^{old}, which results in $\ln p(\boldsymbol{X} \mid \theta^{old})$ for the value of the log likelihood of the marginal distribution. The E-step consists in adapting $q(\boldsymbol{Z})$ so that it matches the posterior likelihood $p(\boldsymbol{Z} \mid \boldsymbol{X})$. In the GMM version of E/M this is done by updating the responsibilities. As explained in Section 2.6, updating the responsibilities and calculating the posterior likelihood are equivalent. At this stage $q = p$, and the KL-divergence is equal to 0, *so for this choice of q, for parameters* θ^{old} *we have*

$$\ln p(\boldsymbol{X} \mid \theta^{old}) = \sum_{\boldsymbol{Z}} q(\boldsymbol{Z}) \ln \frac{p(X, Z \mid \theta^{old})}{q(Z)} \tag{2.32}$$

We now find parameters θ^{new} that maximise $\sum_{\boldsymbol{Z}} p(\boldsymbol{X}, \boldsymbol{Z} \mid \theta)$. Because of the equality in Eq. 2.32, if we manage to find parameters θ^{new} that maximise the lower bound, i.e. increase it (if we cannot find parameters that increase the lower bound we should stop), then necessarily for our current value of $q(\boldsymbol{Z})$, they also increase the marginal likelihood.

It is worthwhile making that explicit. Consider the situation immediately after an E-step. At this stage $q(\boldsymbol{Z}) = p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta_{old})$. Substituting this in the lower bound, we can write:

$$\begin{aligned}\mathcal{L}(q, \theta) &= \sum_{Z} p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta^{old}) \ln p(X, Z \mid \theta) - \sum_{\boldsymbol{Z}} p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta^{old}) \ln p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta^{old}) \\ &= \sum_{\boldsymbol{Z}} p(\boldsymbol{Z} \mid \boldsymbol{X}, \theta^{old}) \ln p(\boldsymbol{X}, \boldsymbol{Z} \mid \theta) + \text{const} \end{aligned} \tag{2.33}$$

where the constant is the negative entropy of q, which does not depend on θ. Since the KL-term is also not dependent on θ, we see that the depency of $\ln p(\boldsymbol{X} \mid \theta)$ is given by that of $\ln p(\boldsymbol{X}, \boldsymbol{Z} \mid \theta)$.

After updating the parameters to θ^{new}, the equality of Eq. 2.32 no longer holds! For our new parameters, the old responsibilities, which were calculated using θ^{old} no longer represent the posterior likelihood p and the KL divergence of p relative to q is greater than 0. We have to calculate new responsibilities, using θ^{new}. We repeat these steps until in the M-step we are no longer capable of improving the lower bound.

Note that in this process the lower bound functions as a ratchet. Upon calculating the responsibilities, we raise the lower bound so that it equals the marginal probability as per Eq. 2.32. If we then manage to increase the expectation value of the joint distribution by picking new parameters we raise the likelihood of the marginal likelihood as well. Then, new responsibilities can be calculated, raising the lower bound to the new marginal likelihood, etc.

The entire process is shown in Fig. 2.7. In the E-step, we choose responsibilities such they match the posterior distribution. The lower bound is equal to $\ln p(\boldsymbol{X} \mid \theta^{old})$, moreover the θ dependency for it and the lower bound are the same, so the curves do not just intersect, but touch. In the M-step the lower bound is optimised for θ and a new value θ^{new} is found. The lower bound is shown as a quadratic function, which indeed it is for a mixture of Gaussians. *The discussion of this Section applies to all members of the exponential family, however.*

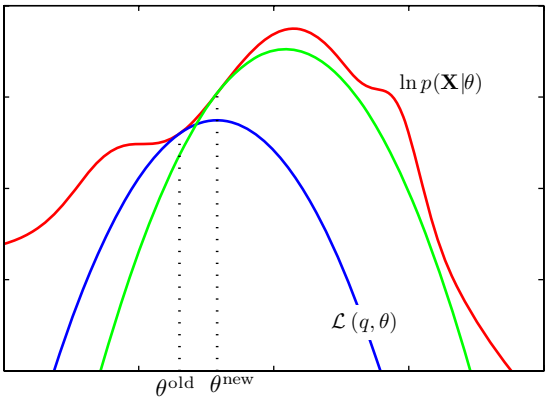


Figure 2.7: This is Figure 9.14 from Bishop (2006).

2.8 *Variational Inference*

It is possible to give a Bayesian treatment of mixture models. This requires a prior distribution on the mixing coefficients π. An advantage of the Bayesian treatment is that mixture components that are not supported by the data are regularised away, in much the same way as large weights are regularised away in Bayesian Linear Regression, at least when they are not supported by subsequent data. This gives the possibility of estimating the number of components from the data. After convergence you can find the mean of the mixing co-

efficient and those components who have clearly smaller means than the others can be cut away.

The theoretical treatment of variational inference is highly technical and is explained in Chapter 10 of [5]. It is well out of scope of this unit, but the result is an algorithm that is not so different from the one we have discussed so far. In *Activity 5.B*, we will give a practical example based on a *scikit-learn* implementation.

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

The discussion introducing the ELBO sets you up to start understanding a recent development in machine learning: *variational autoencoders*. Where in traditional variational inference approximations to the posterior latent variable distributions are extracted by mathematical techniques, in variational autoencoders neural networks are employed to learn the mapping from input data to the latent space are learnt by a neural network. Chapter 8 of [6] gives an engaging introduction of this topic at a level that nicely complements this chapter.

[6] J. V. Stone (2019). *Artificial intelligence engines: A tutorial introduction to the mathematics of Deep Learning*. Sebtel Press

3 Bibliography

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

J. V. Stone (2019). *Artificial intelligence engines: A tutorial introduction to the mathematics of Deep Learning*. Sebtel Press.