

Herbert Jaeger

Machine Learning (KIM.ML09)

Lecture Notes

V 0.4, Nov 22, 2019

Master Program in Artificial Intelligence

Rijksuniversiteit Groningen, Bernoulli Institute

Contents

1	Introduction	4
1.1	Human Versus Machine Learning	4
1.2	The two super challenges of ML - from an eagle's eye	5
1.3	Looking at Human Intelligence, Again	16
1.4	A Remark on "Modeling"	17
1.5	The Machine Learning Landscape	20
2	Decision trees and random forests	25
2.1	A toy decision tree	25
2.2	Formalizing "training data"	27
2.3	Learning decision trees: setting the stage	30
2.4	Learning decision trees: the core algorithm	32
2.5	Dealing with overfitting	36
2.6	Variants and refinements	38
2.7	Random forests	38
3	Elementary supervised temporal learning	44
3.1	Recap: linear regression	44
3.2	Temporal learning tasks	49
3.3	Time series prediction tasks	52
3.4	State-based vs. signal-based timeseries modeling	53
3.5	Takens' theorem	56
4	Basic methods for dimension reduction	59
4.1	Set-up, terminology, general remarks	59
4.2	K-means clustering	62
4.3	Principal component analysis	64
4.4	Mathematical properties of PCA and an algorithm to compute PCs .	68
4.5	Summary of PCA based dimension reduction procedure	70
4.6	Eigendigits	70
4.7	Self-organizing maps	72
4.8	Summary discussion. Model reduction, data compression, dimension reduction	78
5	Discrete symbolic versus continuous real-valued	81
6	The bias-variance dilemma and how to cope with it	86
6.1	Training and testing errors	87
6.2	The menace of overfitting – it's real, it's everywhere	90
6.3	An abstract view on supervised learning	95
6.4	Tuning model flexibility	98
6.5	Finding the right modeling flexibility by cross-validation	104

6.6	Why it is called the bias-variance dilemma	106
7	Representing and learning distributions	109
7.1	Optimal classification	109
7.2	Representing and learning distributions	111
7.3	Mixture of Gaussians; maximum-likelihood estimates by EM algorithms	124
7.4	Parzen windows	135
8	Bayesian model estimation	139
8.1	The ideas behind frequentist statistics	139
8.2	The ideas behind Bayesian statistics	141
8.3	Case study: modeling proteins	147
A	Elementary mathematical structure-forming operations	152
A.1	Pairs, tuples and indexed families	152
A.2	Products of sets	153
A.3	Products of functions	153
B	Joint, conditional and marginal probabilities	154
C	The argmax operator	160
D	Expectation, variance, covariance, and correlation of numerical random variables	160
E	Derivation of Equation 31	164

1 Introduction

1.1 Human Versus Machine Learning

Humans learn. Animals learn. Societies learn. Machines learn. It looks like “learning” were a universal phenomenon and all we had to do is to develop a solid scientific theory of “learning”, turn that into algorithms and then let “learning” happen on computers. Wrong wrong wrong. Human learning is very different from animal learning (and amoebas learn different things in different ways than chimpanzees), societal learning is quite another thing as human or animal learning, and machine learning is as different from any of the former as cars are from horses.

Human learning is incredibly scintillating and elusive. It is as complex and impossible to be fully understood as you can’t fully understand yourself. Think of *all* the things you can *do*, all of your body motions from tying your shoes to playing the guitar; thoughts you can think from “aaagrhhh!” to “I think therefore I am”; achievements personal, social, academic; *all* the things you can remember including your first kiss and what you did 20 seconds ago (you started reading this paragraph, in case you forgot); your plans for tomorrow and the next 40 years; well, just *everything* about you — and almost everything of that wild collection is the result of a fabulous mixing learning of some kind with other miracles and wonders of life. To fully understand human learning, a scientist would have to integrate *at least* the following fields and phenomena:

body, brain, sensor & motor architecture · physiology and neurophysiology · body growth · brain development · motion control · exploration, curiosity, play · creativity · social interaction · drill and exercise and rote learning · reward and punishment, pleasure and pain · the universe, the earth, the atmosphere, water, food, caves · evolution · dreaming · remembering · forgetting · aging · other people, living · other people, long dead · machines, tools, buildings, toys · words and sentences · concepts and meanings · letters and books and schools · traditions ...

Recent spectacular advances in machine learning may have nurtured the impression that machines come already somewhat close. Specifically, neural networks with many cascaded internal processing stages (so-called *deep* networks) have been trained to solve problems that were considered close to impossible only a few years back. A showcase example (one that got me hooked) is automated image caption (technical report: Kiros et al. (2014)). At <http://www.cs.toronto.edu/~nitish/nips2014demo> you can find stunning examples of caption phrases that have been automatically generated by a neural network based system which was given photographic images as input. Figure 1 shows some screenshots. This is a demo from 2014. Since deep learning is evolving incredibly fast, it’s already a little outdated today. Other fascinating examples of deep learning are face recognition

(Parkhi et al., 2015), online text translation (Bahdanau et al., 2015), inferring a Turing machine (almost) from input-output examples (Graves et al, 2016), or playing the game of Go at and beyond the level of human grand-masters (Silver et al, 2016).

So, apparently machine learning algorithms come close to human performance in several tasks or even surpass humans, and these performance achievements have been *learnt* by the algorithms, — thus, machines today can learn like humans??!? The answer is NO. ML researchers (the *really good* ones, not the average TensorFlow user) are highly aware of this. Outside ML however, naive spectators (from the popular press, politics, or other sciences) often conclude that since learning machines can perform similar feats as humans, they also learn like humans. It takes some effort to argue why this is not so (read Edelman (2015) for a refutation from the perspective of cognitive psychology). I cannot embark on this fascinating discussion at this point. Very roughly speaking, it's the same story again as with chess-playing algorithms: the best chess programs win against the best human chess players, but not by fair means — chess programs are *based on larger amounts of data* (recorded chess matches) than humans can memorize, and chess programs can do *vastly more computational operations per second* than a human can do. Brute force wins over human brains at some point when there is enough data and processing bandwidth. Progress has accelerated in the last years because increasingly large training datasets have become available and fast enough computing systems have become cheap enough.

This is not to say that powerful “deep learning” just means large datasets and fast machines. These conditions are necessary but not sufficient. In addition, also numerous algorithmical refinements and theoretical insights in the area of statistical modeling had to be developed. Some of these algorithmical/theoretical concepts will be presented in of this course.

Take-home message: The astonishing learning feats of today’s ML are based on statistical modeling techniques, raw processing power and a lot of a researcher’s personal experience and trial-and-error optimization. It’s technology and maths, not brain biology or psychology. Dismiss any romantic ideas about ML. ML is stuff for sober engineers. But you are allowed to become very excited about that stuff, and that stuff can move mountains.

1.2 The two super challenges of ML - from an eagle’s eye

In this section I want to explain, on an introductory, pre-mathematical level, that large parts of ML can be understood as the art of estimating probability distributions from data. And that this art faces a double super challenge: the unimaginably *complex geometry* of real-world data distributions, and the extreme *lack of information* provided by real-world data. I hope that after reading this section you will be convinced that machine learning is impossible. Since however Siri exists, where is the trick? ... in this way I want to make you read on, like in

Tags

- bicyclers
- riders
- bikers
- equestrians
- bicyclists

Nearest Caption in the Training Dataset

a couple of people on four horses walking the streets .

**Generated Captions**

- a couple of men and horses on a street .
- some people are riding horses on a street .
- a group of people walking on horseback on a street .
- a couple of people with horses walk down the street .
- a group of people riding horses down a street .

Tags

- docks
- pontoon
- dockside
- boats
- pontoons

Nearest Caption in the Training Dataset

a group of people on a bridge beside a boat .

**Generated Captions**

- several people are on a dock in the water .
- a group of people are on a boat in the distance .
- some people on a boat in the harbor .
- a group of people are on the dock on a large boat .
- some people are on a bridge over a boat .

Tags

- ledges
- ledge
- flagstone
- porches
- casters

Nearest Caption in the Training Dataset

three men are sawing a tree into sections .

**Generated Captions**

- the two men are trying to use a tree .
- two men sitting in front of a wood tree .
- four men are working on a concrete structure .
- two men are working on wood .
- a man and a child in a wooden park are sitting on a farm .

Figure 1: Three screenshots from the image caption demo at <http://www.cs.toronto.edu/~nitish/nips2014demo>. A “deep learning” system was trained on some tens of thousands of photos showing everyday scenes. Each photo in the training set came with a few short captions provided by humans. From these training data, the system learnt to generate tags and captions for new photos. The tags and captions on the left were produced by the trained system upon input of the photos at the right.

a crime novel.

1.2.1 The superchallenge of complex geometry

In order to highlight the data geometry challenge, I will use the demo task highlighted in Figure 1 as an example. Let us follow the sacred tradition of ML and first introduce an acronym that is not understandable to outsiders: TICS = the Toronto Image Caption System.

The TICS dataset and learning task and demo implementation come from one of the three labs that have pioneered the field which is now known as *Deep Learning* (DL, our next acronym), namely Geoffrey Hinton’s Lab at the University of Toronto. (The other two widely acclaimed DL pioneers are Yoshua Bengio, University of Montréal, and Yann LeCun, Courant Institute of Mathematics, New York; Hinton, Bengio and LeCun received the 2018 Turing award – the Nobel prize in computer science). The TICS demo is a beautiful case to illustrate some fundamentals of machine learning.

After training, TICS, at first sight, implements a *function*: test image in, caption (and tags) out. If one looks closer, the output however is not just a single caption phrase, but a list of several captions (“generated captions” in Figure 1) rank-ordered by probability: captions that TICS thinks are more probable are placed higher in the list. Indeed TICS computes a relative probability value for each suggested caption. This probability value is not shown in the screenshots.

Let us look at this in a little more detail.

TICS captions are based on a finite vocabulary of English words. For simplicity let us assume that the length of captions that TICS generates is always 10 words. The neural networks that hum behind the curtains of TICS cannot process “words” – the only data type that they can handle is real-valued vectors. One of the innovations which boosted DL is a method to represent words by vectors such that *semantic similarity* of words is captured by *metric closeness* of vectors: the vector representing the word “house” lies close to the vector for “building” but far away from the vector for “mouse”. A landmark paper where you can learn more about this technique is Mikolov et al. (2013), an accessible tutorial is Minnaar (2015). A typical size for such semantic word vectors is a dimension of a few 100’s – let’s say, TICS uses 300-dimensional word vectors (am too lazy while typing this to check it out). Thus, a caption can be represented by a sequence of ten 300-dimensional vectors, which boils down to a single 3000-dimensional vector, that is, a point $c \in \mathbb{R}^{3000}$.

Similarly, an input picture sized 600×800 pixels with 3 color channels is represented to TICS as a vector u of dimension $3 \times 600 \times 800 = 1,440,000$.

Now TICS generates, upon a presentation of an input picture vector u , a list of what it believes to be the five most probable captions for this input. That is, TICS must have an idea of the probability ratios of different caption candidates. Formally, if C denotes the random variable (RV) that returns captions c , and if U denotes the RV which produces sample input pictures u , TICS must compute

ratios of conditional probabilities of the form $P(C = c_i|U = u)/P(C = c_j|U = u)$. In the semantic word vector space, these ratios become ratios of the values of probability density functions (pdf) over the caption vector space \mathbb{R}^{3000} . For every input image u , TICS *must* have some representation of the 3000-dimensional pdf describing the probabilities of caption candidates for image u .

Now follow me on a little excursion into geometric imagination.

Consider some specific vector $c \in \mathbb{R}^{3000}$ which represents a plausible 10-word caption for image u , that is, the pdf value $p(c)$ is relatively large. What happens to the pdf value if we move away from c by taking a little step $\delta \in \mathbb{R}^{3000}$ of length, say, $\|\delta\| = 0.1$, that is, how does $p(c + \delta)$ compare to $p(c)$? This depends on the direction in which δ points. In a few directions, $p(c + \delta)$ will be about as large as $p(c)$. This happens when δ points toward another caption vector c' which has one word replaced by another word that is semantically close. For example, consider the caption “*A group of people on a bridge beside a boat*”. The last 300 elements in the 3000-dimensional vector c coding this caption stand for the word *boat*. Replacing this caption by “*a group of people on a bridge beside a ship*” gives another codevector $c' \in \mathbb{R}^{3000}$ which is the same as c except in the last 300 components, which have been replaced by the semantic word vector for *ship*. Then, if δ points from c toward c' (that is, δ is a fraction of $c' - c$), $p(c + \delta)$ will not differ from $p(c)$ in a major way.

If you think a little about it, you will come to the conclusion that such δ which leave $p(c + \delta)$ roughly at the same level are always connected with replacing words by semantically related words. Other change directions δ^* will either make no semantical sense or destroy the grammatical structure connected with c . The pdf value $p(c + \delta^*)$ will drop dramatically compared to $p(c)$ in those cases.

Now, in a 10-word caption, how many replacements of some word with a related one exist? Some words will be grammatical function words (“*a*”, “*of*” etc.) which admit only a small number of replacements, or none at all. The words that carry semantic meaning (“*group*”, “*people*” etc.) typically allow for a few sense-making replacements. Let us be generous and assume that a word in a 10-word caption, on average, can be replaced by 5 alternative words such that after the replacement, the new caption still is a reasonable description of the input image.

This means that around c there will be $5 \cdot 10 = 50$ directions in which the relatively large value of $p(c)$ stays large. Assuming these 50 directions are given by linearly independent δ vectors, we find that around c there is a 50-dimensional affine linear subspace S of \mathbb{R}^{3000} in which we can find high p values in the vicinity of c , while in the 2950 directions orthogonal to S , the value of p will drop fast if one moves away from c .

Ok., this was a long journey to a single geometric finding: locally around some point c where the pdf is relatively large, the pdf will stay relatively large only a small fraction of the directions - these directions span a low-dimensional hyperplane around c . If you move a little further away from c on this low-dimensional “sheets”, following the lead of high pdf values, you will find that this

high-probability surface will take you on a curved path - these high-probability sheets in \mathbb{R}^{3000} are not flat but curved.

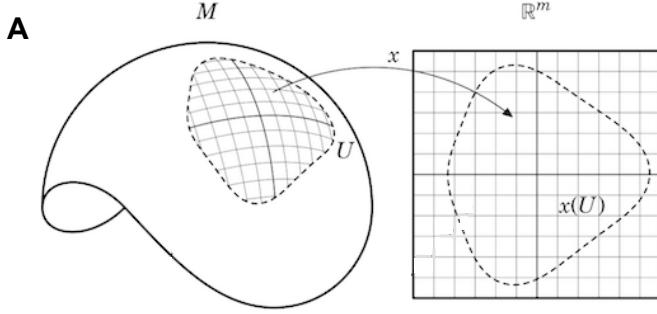
The mathematical abstraction of such relatively high-probability, low-dimensional, curved sheets embedded in \mathbb{R}^{3000} is the concept of a *manifold*. Machine learning professionals often speak of the “data manifold” in a general way, in order to indicate that the geometry of high-probability areas of real-world pdfs consists of “thin” (low-dimensional), curved sheet-like domains curled into the embedding data space. It is good for ML students to know the clean mathematical definition of a manifold. Although the geometry of real-world pdfs will be less clean than this mathematical concept, it provides a useful set of intuitions, and advanced research in DL algorithms frequently starts off from considering manifold models.

So, here is brief intro to the mathematical concept of a manifold. Consider an n -dimensional real vector space \mathbb{R}^n (for TICS this would be $n = 3000$). Let $m \leq n$ be a positive integer not larger than n . An m -dimensional manifold \mathcal{M} is a subset of \mathbb{R}^n which locally can be smoothly mapped to \mathbb{R}^m , that is, at each point of \mathcal{M} one can smoothly map a neighborhood of that point to a neighborhood of the origin in the m -dimensional Euclidean coordinate system (Figure 2A).

1-dimensional manifolds are just lines embedded in some higher-dimensional \mathbb{R}^n (Figure 2B), 2-dimensional manifolds are surfaces, etc. Manifolds can be wildly curved, knotted (as in Figure 2C), or fragmented (as in Figure 2B). Humans cannot visually imagine manifolds of dimension greater than 2.

The “data manifold” of a real-world data source is not a uniquely or even well-defined thing. For instance, returning to the TICS example, the dimension of the manifold around a caption c would depend on an arbitrary threshold fixed by the researcher – a direction δ would / would not lead out of the manifold if the pdf decreases in that direction faster than this threshold. Also (again using the caption scenario) around some good captions c the number of “good” local change directions will differ from the number around another, equally good, but differently structured caption c' . For these and other reasons, claiming that data distributions are shaped like manifolds is a strongly simplifying abstraction.

Despite this abstraction inaccuracy, the geometric intuitions behind the manifold concept have led to substantial insight into the (mal-)functioning of machine learning methods. *Adversarial attacks* on deep networks is a good example of the usefulness of the data manifold concept. Take a look at Figure 3 – taken from a much-cited paper (Goodfellow et al., 2014) which explores the phenomenon of adversarial attacks. The left panel shows a photo of a panda (one would think). It is given as input to a deep network that was trained to classify images. The network correctly classifies it as “panda”, with a “confidence” of 58% (this “confidence” level is derived from probability ratios). The middle panel shows a noise pattern. If a very small multiple (factor 0.007) of this noise pattern is added to the panda picture, one gets the picture shown in the right panel. For the human eye there is no perceptible change. However, the neural network now classifies it as “gibbon” with a dead-sure confidence level of 99%.



kahrstrom.com/mathematics/illustrations.php



en.wikipedia.org/wiki/Manifold



www.math.utah.edu/carlson60/

Figure 2: **A** An m -dimensional manifold can be locally “charted” by a bijective mapping to a neighborhood of the origin in \mathbb{R}^m . Example shows a curved manifold of dimension $m = 2$ embedded in \mathbb{R}^3 . **B** Some examples of 1-dimensional manifolds embedded in \mathbb{R}^2 (each color corresponds to one manifold — manifolds need not be connected). **C** A more wildly curved 2-dimensional manifold in \mathbb{R}^3 (the manifold is the surface of this strange body).

What has happened here? Well, this is a manifold story. Let’s say that the panda image is sized 600×600 pixels (I didn’t check), with three color channels. Thus, mathematically, such images are points in $\mathbb{R}^{1080000}$. When the neural network was trained, it was presented with a large number of example images, that is, a point cloud in $\mathbb{R}^{1080000}$. The outcome of the learning algorithm is an estimate (up to an undetermined scaling factor) of a pdf in $\mathbb{R}^{1080000}$. Geometrically speaking, this pdf is highly concentrated along a low-dimensional manifold, call it \mathcal{M} . The dimension of this manifold is given by the number of neurons in the most narrow layer of the neural network, say, it is $m = 1000$ (this would be a standard order of magnitude in such deep networks; I didn’t check). Thus, the original panda picture corresponds to a point u on a 1000-dimensional manifold which is curled into a 1,080,000-dimensional embedding space. In terms of dimensions, the manifold uses only about one out of thousand dimensions! Now, add that noise

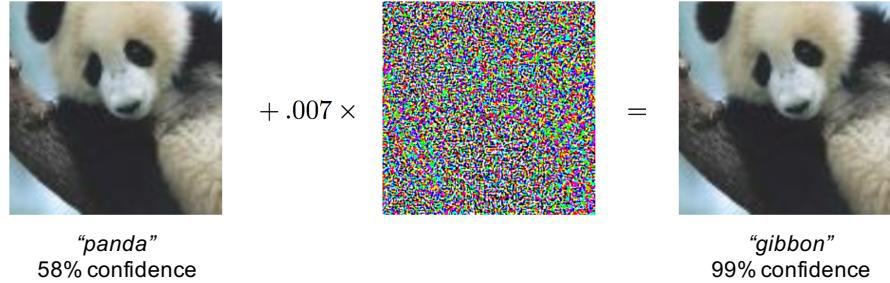


Figure 3: Manifold magic: Turning a panda into a gibbon. For explanation see text. Picture taken from Goodfellow et al. (2014)

image (call it δ) to u , getting the rightmost panel picture as $u + \delta$. If δ is prepared in a way that it points in a direction orthogonal to \mathcal{M} , the value of the pdf will shrink dramatically. The network, when presented with input $u + \delta$, will “think” that it has never seen anything like this $u + \delta$ image, and will return a random classification – “gibbon” in this case.

Adversarial examples are today widely used in a number of different ways to improve the quality of deep learning applications. Check out Section 7.13 in the “bible” of deep learning (Goodfellow et al., 2016) for a primer.

The TICS engineers enforced by neural network design that the 1,440,000 dimensional raw input image vectors were projected on a manifold that had only 4096 dimensions. This was achieved by setting up a layered neural network for image preprocessing whose input layer (the “retina” of the network) had 1,440,000 neurons (one neuron per color pixel), and whose output layer had 4096 neurons. The right “folding into shape” of this manifold was effected by the neural network training.

Takehome summary: real-world data distributions are typically concentrated in exceedingly thin (low-dimensional), badly curled-up sheets in the embedding (high-dimensional) data space \mathbb{R}^n . Machine learning models, such as the TICS system, must contain some kind of model of this distribution. Such models will also typically feature a concentration of probability mass in very thin, curled sheets. A universal challenge for machine learning is to develop modeling methods that can represent such complex, manifold-like distribution geometries, and “learn” to align the model distribution with the data distribution. A catchword to point to this bundle of ideas and challenges is “data manifold”.

1.2.2 The superchallenge of lack of information

I continue to use the TICS system to illustrate this second challenge. What follows will again require an effort in geometrical thinking, but that’s the nature of vector data and the information contained in them.



*Gray haired man in black suit and yellow tie working in a financial environment.
A graying man in a suit is perplexed at a business meeting.
A businessman in a yellow tie gives a frustrated look.
A man in a yellow tie is rubbing the back of his neck.
A man with a yellow tie looks concerned.*



*A butcher cutting an animal to sell.
A green-shirted man with a butcher's apron uses a knife to carve out the hanging carcass of a cow.
A man at work, butchering a cow.
A man in a green t-shirt and long tan apron hacks apart the carcass of a cow
while another man hoses away the blood.
Two men work in a butcher shop; one cuts the meat from a butchered cow, while the other hoses the floor.*

Figure 4: Two images and their annotations from the training dataset. Taken from Young et al. (2014).

The data used for training TICS contained about 30,000 photos taken from Flickr, each of which was annotated by humans with 5 captions. Figure 4 shows two examples. Each photo was sized 600×800 pixels, with three color channels, making a total of 1,440,000 numbers. We will consider the intensity values as normalized to the range $[0, 1]$, which makes each photo a vector in the unit hypercube $[0, 1]^{1,440,000}$. Assuming as before that captions can be coded as 3000-dimensional vectors, which we will also take as normalized to a numeric range of $[0, 1]$, a photo-caption pair makes a vector in $[0, 1]^{1,443,000}$. In order to simplify the following discussion we assume that each training image came with a single caption only. Thus, the training data consisted in 30,000 points in the 1,443,000-dimensional unit hypercube.

It is impossible for humans to visualize, imagine or even hallucinate the wild ways of how points can be spatially distributed in a 1,443,000-dimensional vector space. In Figure 5 I attempt a visualization of the TICS data scenario in a 3-dimensional projection of this 1,443,000-dimensional space – three dimensions being the largest spatial dimension that our brains can handle. The rendering in Figure 5 corresponds to a dataset where each “photo” is made of merely two grayscale pixels. Each such “photo” is thus a point in the square $[0, 1]^2$ (light blue area in the figure, spanned by the two pixel intensities x_1, x_2). Each caption is coded by a single number $y \in [0, 1]$. We furthermore simplify the situation by assuming that the training dataset contains only two photos u_1, u_2 . The caption coding vector of a photo is here reduced to a single number, plotted above the photo’s coordinates on the y -axis (blue crosses). The two blue crosses in the figure thus represent the information contained in the training data.

Now consider a new test image u^* (orange diamond in the figure). The TICS system must determine a suitable caption for u^* , that is, an appropriate value y^* – a point somewhere on the orange broken ??-line in the figure.

But all that TICS knows about captions is contained the two training data points (blue crosses). If you think about it, there is *no way* for TICS to infer from these two points where y^* should lie. *Any* placement along the ??-line is logically

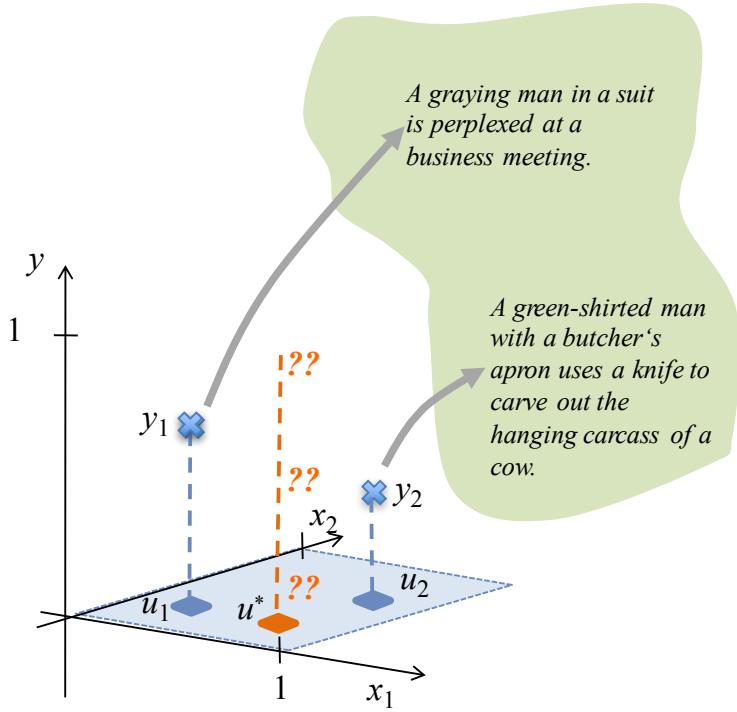


Figure 5: Training data for TICS (highly simplified). The photo dataspace (light blue square spanned by x_1, x_2) is here reduced to 2 dimensions, and the caption dataspace to a single ony (y -axis). The training dataset is assumed to contain two photos only (blue diamonds u_1, u_2), each with one caption (blue crosses with y -values y_1, y_2). A test image u^* (orange diamond) must be associated by TICS with suitable caption which lies somewhere in the y -direction above u^* (dashed orange ??-line).

possible!

In order to determine a caption position along the ??-line, TICS *must* add some optimization criterion to the scenario. For instance, one could require one of the following conditions:

1. Make y^* that point on the ??-line which has the smallest total distance to all the training points.
2. One might wish to grant closer-by training images a bigger impact on the caption determination than further-away training images. Thus, if $d(u_i, u^*)$ denotes the distance between training image u_i and test image u^* , set y^* to the weighted mean of the training captions y_i , where the weights are inversely proportional to the distance of the respective images:

$$y^* = \frac{1}{\sum_i d(u_i, u^*)^{-a}} \sum_i d(u_i, u^*)^{-a} y_i.$$

The parameter a modulates how the impact of further-away images decays with distance.

3. Place a spherical 3-dimensional Gaussian with variance σ^2 around every blue cross. Determine $p(y^*)$ to be that point on the ??-line where the sum of these Gaussians is maximal.
4. Consider all smoothly curved line segments L in the 3-dimensional data cube in Figure 5 which connect the blue crosses and cut through the ??-line. For any such L , consider the integral $\gamma(L)$ of absolute curvature along L . Find that curved line segment L_{opt} which minimizes $\gamma(L)$. Then declare the crossing of this L_{opt} with the ??-line to be y^* . – Like in the second proposal above, one would also want to emphasize the impact of close-by images, which would lead to a weighted curvature integral.

I collect some observations from this list:

- All of these optimization criteria make some intuitive sense, but they would lead to different results. The generated captions will differ depending on which criterion is chosen.
- The criteria rest on quite different intuitions. It is unclear which one would be better than another one, and on what grounds one would compare their relative merits. A note in passing: finding comparison criteria to assess relative merits of different statistical estimation procedures is a task that constitutes an entire, important branch of statistics. For an introduction you might take a look at my lecture notes on “Principles of statistical modeling”, Section 19.3.1 “Comparing statistical procedures” (<http://minds.jacobs-university.de/teaching/ln/>).
- Criteria 2 and 3 require the choice of design parameters (a, σ^2) . The other criteria could be upgraded to include reasonable design parameters too. It is absolutely typical for machine learning algorithms to have such *hyperparameters* which need to be set by the experimenter. The effects of these hyperparameters can be very strong, determining whether the final solution is brilliant or useless.
- Casting an optimization criterion into a working algorithmic procedure may be challenging. For criteria 1, 2 and 3 in the list, this would be relatively easy. Turning criterion 4 into an algorithm looks fearsome.
- Each of the criteria leads to implicit constraints on the geometric shape of the learnable data manifolds.

Some methods in ML are cleanly derived from optimization criteria, with mathematical theory backing up the design of algorithms. Other branches employ

learning algorithms that are so complex that one loses control over what is actually optimized, and one has only little insight in the geometrical and statistical properties of the models that are delivered by these methods. This is particularly true for deep learning methods.

There is something like an irreducible arbitrariness of choosing an optimization condition, or, if one gives up on fully understanding what's happening, an arbitrariness in the design of complex learning algorithms. This turns machine learning engineering into a quite personal thing, even into an art.

To put this all in a nutshell: the available training data don't include the information of how the information contained in them should be "optimally" extracted, or *what* kind of information should be extracted. This could be called a *lack of epistemic information* (my private terminology; epistemology is the branch of philosophy that is concerned with the question by which methods of reasoning humans get acquire knowledge, and with the question what "knowledge" is in the first place - check out <https://en.wikipedia.org/wiki/Epistemology> if you are interested in these deep, ancient, unsolved questions).

The discussion of lacking epistemic information is hardly pursued in today's ML, although there have been periods in the last decades when fierce debates on such issues have been led.

But there is also another lack of information which *is* a standard theme in today's ML textbooks. This issue is called the *curse of dimensionality*. I will highlight this curse with our TICS demo again.

In mathematical terminology, after training our (simplified) TICS system is able to compute a function $f : [0, 1]^2 \rightarrow [0, 1]$ which computes a caption $f(u^*)$ for every input test image $u^* \in [0, 1]^2$. The only information TICS has at learning time is contained in the training data points (blue crosses in our figure).

Looking at Figure 5, estimating a function f from just the two blue crosses is clearly a dramatically underdetermined task. You may argue that the version of the TICS learning task which I gave in Figure 5 has been simplified to an unfair degree, and that the real TICS system had not just 2, but 30,000 training images to learn on.

But, in fact, for the full-size TICS the situation is even much, *much* worse than what appears in Figure 5:

- In the caricature demo there were 2 training images scattered in a 2-dimensional (pixel intensity) space. That is, as many data points as dimensions. In contrast, in TICS there are 30,000 images scattered in a 1,440,000-dimensional pixel intensity space: that is, about 50 times more dimensions than data points! How should it be possible at all to estimate a function from $[0, 1]^{1440000}$ to $[0, 1]^{3000}$ when one has so much fewer training points than dimensions!?
- There is another fact about high-dimensional spaces which aggravates the situation. In the n -dimensional unit hypercube, the greatest possible distance between two points is equal to the longest diagonal in this hypercube,

\sqrt{n} , which amounts to 1200 in the TICS image space. This growth of length of the main diagonal with dimension n carries over to a growth of average distances between random points in n -dimensional hypercubes. In simple terms, the higher the dimension, the wider will training data points lie apart from each other.

The two conditions together – fewer data points than dimensions, and large distances between the points – make it appear impossible to estimate a function on $[0, 1]^{1440000}$ from just those 30,000 points.

This is the dreaded “curse of dimensionality”.

In plain English, the curse of dimensionality says that in high-dimensional data spaces, the available training data points will lie exceedingly far apart from each other. Metaphorically speaking, they are a few isolated flickering stars in a vast and empty universe. But, most learning tasks require one to fill the empty spaces with information.

1.3 Looking at Human Intelligence, Again

An adult human has encoded in his/her brain a very useful, amazingly accurate and detailed model of that human’s outer world, which allows the human to almost immediately create a “situation model” of his/her current physical and social environment, based on the current, ever-new sensor input. This situation model is a condensed, meaningful representation of the current environment. This is analog to the TICS scenario (after training), where the new input is a test image and the situation model corresponds to a distribution over captions.

An adult human has had a childhood and youth’s time to learn most of his/her world model. The “training data” are the sensory impressions collected in, say, the first 25 years of the human’s life. Thinking of a sensory impression as a vector, it is hard to say what is the dimension of this vector. A lower bound could be found in the number of sensory neural fibers reaching the brain. The optical nerve has about 1,000,000 fibres. Having two of them, plus all sorts of other sensory fibers reaching the brain (from the nose, ears, body), let us boldly declare that the dimension of sensor inputs to the brain is 3 Mio. Now the learning period for a human differs from TICS’s learning period in that sensory input arrives in a continuous stream, not in isolated training images. In order to make the two scenarios comparable, assume furthermore that a human organizes the continuous input stream into “snapshots” at a rate of 1 snapshot “picture” of the current sensory input per second (cognitive psychologists will tell us that the rate is actually less). In the course of 25 years, with 12 wake hours per day, this makes about $25 \times 360 \times 12 \times 3600 \approx 390$ Mio “training snapshot images”. This gives a ratio of number of training points over data dimension of $390 \text{ Mio} / 3 \text{ Mio} = 130$, which looks so much better than the ratio of $30,000 / 1,443,000 \approx 0.02$ in TICS’s case.

But even the ratio of 130 data points per dimension is still hopeless, and the curse of dimensionality strikes just as well. Why?

For simplicity we again assume that the 3-Mio dimensional sensor image vectors are normalized to a range of $[0, 1]$, making a sensor impression a point in $[0, 1]^{3000000}$. Statisticians, machine learners, and many cognitive scientists would tell us that the “world model” of a human can be considered to be a probability distribution over the sensory image space $[0, 1]^{3000000}$. This is a hypercube with $2^{3000000}$ corners. In order to estimate a probability distribution over an n -dimensional hypercube, a statistician would demand to have many more data-points than the cube has corners (to see why, think about estimating a probability distribution over the one-dimensional unit hypercube $[0, 1]$ from data points on that interval, then extrapolate to higher dimensions). That is, a brain equipped with ordinary statistical methods would demand to have many more than $2^{3000000}$ training data points to distil a world model from that collection of experiences. But there are only 390 Mio such datapoints collected in 25 years. The ratio 390 Mio / $2^{3000000}$ is about $2^{-2999972}$. That is, a human would have to have a youth lifetime of about $25 \times 2^{2999972} \approx 2^{2999977}$ years in order to learn a statistically defendable world model.

Still, the human brain (and the human around it, with the world around the human around that brain) somehow can do it in 25 years. Cognitive scientists believe that the key to this magic lies in the evolutionary history of man. Through millions of years of incremental, evolutionary brain structure optimization, starting from worms’ brains or earlier, the human brain is *pre-structured* in exactly such ways that it comes with a built-in-by-birth data manifold geometry which reduces the 3 Mio-dimensional raw sensor data format to a *much* lower-dimensional manifold surface. Then, 390 Mio data points may be enough to cover this manifold densely enough for meaningful distribution estimates.

The question which built-in experience pre-shapings a human brings to the table at birth time has a long tradition in philosophy and psychology. A recent line of work that brings this tradition to bear on machine learning is in the research of Joshua Tenenbaum – check out, for instance, Tenenbaum et al. (2006) if you are interested.

1.4 A Remark on “Modeling”

Machine learning in its modern form aims at modeling real-world systems, just like what the natural sciences aim for. But the motivation is different.

Physicists, chemists, biologists et al. want to *understand* reality through their models. Their models should tell the truth – in suitable abstraction – about reality; the models should be *veridical* (from Latin, “saying the truth”). The inner workings of a model should reflect the inner workings of reality. For instance, a detailed chemical model of a reaction which changes a substance A into a substance B should give an account of the kinetic and quantum-mechanical substeps that actually take place in this reaction. Newton’s laws of motion explain the dance of planets around the sun by formulas made of variables which should correspond to

physically real quantities – like gravitational forces, masses or velocities. Models whose inner mechanisms or mathematical variables are intended to capture real mechanisms and real quantities are called *analytical* models.

Machine learners, in contrast, are application oriented. They want *useful* models. ML models of pieces of reality must *function* well in their respective application context. The inner workings of an ML model need not mirror the inner workings of the modeled system. Machine learning models are thus almost always *blackbox* models (a possible exception being Bayesian networks, will be treated in Session 9 of this course). A blackbox model of some real-world system just captures the externally observable input-output behavior of the modeled system, but it may use any internal math or algorithm tricks that its designer can think of. Neural networks are a striking example. A neural network trained for predicting the next day's stock index will be made of hundreds or even millions of interacting variables ("neuron activations") which have no corresponding counterpart in the reality of stock markets.

Figure 6 sketches how blackbox models work. They are derived ("learnt", "estimated") from data emitted by the source system, and they should generate synthetic data that "fit" (has a similar distribution as) the real-world data. And that's it. The structure of the blackbox model need not correspond in any way with the structure of the source system – in Figure 6, this is a robot, while the model is a neural network whose structure has no correspondence whatsoever in the robot design.

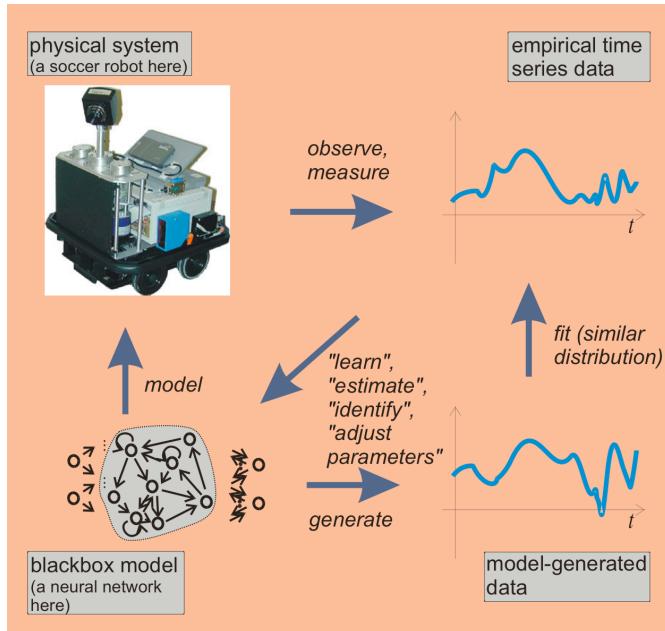


Figure 6: How blackbox models work. For explanation see text.

Analytical and blackbox models have complementary merits:

- Setting up the structure of an analytical model requires from the modeler to have insight into the concerned laws of nature – an expert’s business. A blackbox model can be successfully created by a village idiot with access to TensorFlow.
- A blackbox model requires training data. There are cases where such data are not available in sufficient quantity. Then analytical modeling is the only way to go.
- As a rule, analytical models are much more compact than blackbox models. Compare $E = m c^2$ with the TICS system, which consists of several modules, some of which are neural networks with hundreds of thousands of variables.
- When the target system that one wants to model is very complex, there is typically no chance for an analytical model. Example: an analytical model of how humans describe images in captions would have to include accounts of the human brain and the cultural conditions of language use. No way. Only blackbox models can be used here.
- The great – really very great – advantage of analytical models is that they generalize to all situations within their scope. The laws of gravitation can be applied to falling apples as well as to the majestic whirling of galaxies. If a blackbox model would have been trained by Newton on data collected from the sun and planets motions, this model would be exclusively applicable to our planetary system, not to apples and not to galaxies.

An interesting and very relevant modeling task is to model the earth’s atmosphere for weather forecasting. This modeling problem has been intensely worked on for many decades, and an interesting mix of analytical and blackbox methods marks the state of the art. If there is time and interest I will expand on this in the tutorial session.

I conclude this section with a tale from my professional life which nicely illustrates the difference between analytical and blackbox modeling. I was once called to consult a company in the chemical industry. They wanted a machine learning solution for the following problem. In one of their factories they had built a production line whose output was a certain artificial resin. Imagine a large hall full of vessels, heaters, pumps and valves and tubes. The production process of that resin was prone to a particularly nasty possible failure: if the process would not be controlled correctly, some intermediate products might solidify. The concerned vessels, tubes and valves would be blocked and could not be cleared – necessitating to disassemble the facility and replace the congested parts with new ones. Very expensive! The chemical engineers had heard of the magic of neural networks and wanted one of these, which should give them early warnings if the production process was in danger of drifting into the danger zone. I told them that this was (maybe) possible if they could provide training data. What training data, please?

Well, in order to predict failure, a neural network needs examples of this failure. So, could the engineers please run the facility through a reasonably large number of solidifications, a few hundreds maybe for good statistics? Obviously, that was that. Only analytical modeling would do here. A good analytical model would be able to predict any kind of imminent solidification situations. But that wasn't an option either because the entire production process was too complex for an accurate enough analytical model. Now put yourself into the skin of the responsible chief engineer. What should he do to prevent the dreaded solidification to happen, ever? Another nice discussion item for our tutorial session.

1.5 The Machine Learning Landscape

ML as a field which perceives itself as a field under this name is relatively young, say, about 40 years. It is interdisciplinary and has historical and methodological connections to neuroscience, cognitive science, linguistics, mathematical statistics, AI, signal processing and control, and pattern recognition (the latter is a traditional academic subdiscipline of computer science); it uses mathematical methods from statistics (of course), information theory, signal processing and control, dynamical systems theory, mathematical logic and numerical mathematics; and it has a very wide span of application types. This diversity in traditions, methods and applications makes it difficult to study "Machine Learning". Any given textbook, even if it is very thick, will render the individual author's specific view and knowledge of the field and will be partially blind to other perspectives. This is quite different from other areas in CS, say for example formal languages / theory of computation / computational complexity where a widely shared repertoire of standard themes and methods cleanly define the field.

I have a brother, Manfred Jaeger, who is a machine learning professor at Aalborg university (<http://people.cs.aau.dk/~jaeger/>). We naturally often talk with each other, but never about ML because I wouldn't understand what he is doing and vice versa. We have never met at scientific conferences because we attend different ones, and we publish in different journals.

The leading metaphors of ML have changed over the few decades of the field's existence. The main shift, as I see it, was from "cognitive modeling" to "statistical modeling". In the 1970-1980ies, a main research motif/metaphor in ML (which was hardly named like that then) was to mimic human learning on computers, which connected ML to AI, cognitive science and neuroscience. While these connections persist to the day, the mainstream self-perception of the field today is to view it very soberly as the craft of estimating complex probability distributions with efficient algorithms and powerful computers.

My personal map of the ML landscape divides it into four main segments with distinct academic communities, research goals and methods:

Segment 1: Theoretical ML. Here one asks what are the fundamental possibilities and limitations of inferring knowledge from observation data. This

is the most abstract and “pure maths” strand of ML. There are cross-connections to the theory of computational complexity. Practical applicability of results and efficient algorithms are secondary. Check out https://en.wikipedia.org/wiki/Computational_learning_theory and https://en.wikipedia.org/wiki/Statistical_learning_theory for an impression of this line of research.

Segment 2: Symbolic-logic learning, data mining. Here the goal is to infer symbolic knowledge from data, to extract logical rules from data, to infer facts about the real world expressed in fragments of first-order logic or other logic formalisms, often enriched with probabilistic information. Neural networks are rarely used. A main motif is that these resulting models be human-understandable and directly useful for human end-users. Key terms are “knowledge discovery”, “data mining”, or “automated knowledge base construction”. This is the area of my brother’s research. Check out https://en.wikipedia.org/wiki/Data_mining or https://en.wikipedia.org/wiki/Inductive_logic_programming or Suchanek et al. (2013) for getting the flavor. This is an application-driven field, with applications e.g. in bioinformatics, drug discovery, web mining, document analysis, decision support systems.

A beautiful case study is the PaleoDeepDive project described in Peters et al. (2014). This large-scale project aimed at making paleontological knowledge easily searchable and more reliable. Palaeontology is the science of extinct animal species. Its “raw data” are fossil bones. It is obviously difficult to reliably classify a handful of freshly excavated bones as belonging to a particular species – first, because one usually doesn’t dig out a complete skeleton, and second because extinct species are not known in the first place. The field is plagued by misclassifications and terminological uncertainties – often a newly found set of bones is believed to belong to a newly discovered species, for which a new name is created, although in reality other fossil findings already named differently belong to the same species. In the PaleoDeepDive project, the web was crawled to retrieve virtually all scientific pdf documents relating to paleontology – including documents that had been published in pre-digital times and were just image scans. Using optical character recognition and image analysis methods at the front end, these documents were made machine readable, including information contained in tables and images. Then, unsupervised, logic-based methods were used to identify suspects for double naming of the same species, and also the opposite: single names for distinct species – an important contribution to purge the evolutionary tree of the animal kingdom.

Segment 3: Signal and pattern modeling. This is the most diverse sector in my private partition of ML and it is difficult to characterize globally. The basic attitude here is one of quantitative-numerical blackbox modeling. Our

TICS demo would go here. The raw data are mostly numerical (like physical measurement timeseries, audio signals, images and video). When they are symbolic (texts in particular), one of the first processing steps typically encodes symbols to some numerical vector format. Neural networks are widely used and there are some connections to computational neuroscience. The general goal is to distil from raw data a numerical representation (often implicit) of the data distribution which lends itself to efficient application purposes, like pattern classification, time series prediction, motor control to name a few. Human-user interpretability of the distribution representation is not easy to attain, but has lately become an important subject of research. Like Segment 2, this field is decidedly application-driven. Under the catchterm “deep learning” a subfield of this area has recently received a lot of attention.

Segment 4: Agent modeling and reinforcement learning. The overarching goal here is to model entire intelligent agents — humans, animals, robots, software agents — that behave purposefully in complex dynamical environments. Besides learning, themes like motor control, sensor processing, decision making, motivation, knowledge representation, communication are investigated. An important kind of learning that is relevant for agents is *reinforcement learning* — that is, an agent is optimizing its action-decision-making in a lifetime history based on reward and punishment signals. The outcome of research often is *agent architectures*: complex, multi-module “box-and-wiring diagrams” for autonomous intelligent systems. This is likely the most interdisciplinary corner of ML, with strong connections to cognitive science, the cognitive neurosciences, AI, robotics, artificial life, ethology, and philosophy.

It is hard to judge how “big” these four segments are in mutual comparison. Surely Segment 1 receives much less funding and is pursued by substantially fewer researchers than segments 2 and 3. In this material sense, segments 2 and 3 are both “big”. Segment 4 is bigger than Segment 1 but smaller than 2 or 3. My own research lies in 3 and 4. In this course I focus on the third segment — you should be aware that you only get a partial glimpse of ML.

A common subdivision of ML, partly orthogonal to my private 4-section partition, is based on three fundamental kinds of learning tasks:

Supervised learning. Training data are “labelled pairs” (x_n, y_n) , where x is some kind of “input” and y is some kind of “target output” or “desired / correct output”. TICS is a typical example, where the x_n are images and the y_n are captions. The learning objective is to obtain a mechanism which, when fed with new test inputs x_{test} , returns outputs y_{test} that generalize in a meaningful way from the training sample. The underlying mathematical task is to estimate the conditional distributions $P_{Y|X}$ from the training sample (check the end of Appendix B for a brief explanation of this notation).

The learnt input-output mechanism is “good” to the extent that upon input x_{test} it generates outputs that are distributed according to the true conditional distribution $P(Y = y | X = x_{\text{test}})$, just as we have seen in the TICS demo. Typical and important special cases of supervised learning are *pattern classification* (the y are correct class labels for the input patterns x) or *timeseries prediction* (the y are correct continuations of initial timeseries x). Segment 3 from my private segmentation of ML is the typical stage for supervised learning.

Unsupervised learning. Training data are just data points x_n . The task is to discover some kind of “structure” (regularities, symmetries, redundancies...) in the data distribution which can be used to create a compressed representation of the data. Unsupervised learning can become very challenging when data points are high-dimensional and/or when the distribution has a complex shape. Unsupervised learning is often used for *dimension reduction*. The result of an unsupervised learning process is a dimension-reducing, *encoding* function e which takes high-dimensional data points x as inputs and returns low-dimensional encodings $e(x)$. This encoding should preserve most of the information contained in the original inputs x . That is, there should also exist a *decoding* function d which takes encodings $e(x)$ as inputs and transforms them back to the high-dimensional format of x . The overall loss in the encoding-decoding process should be small, that is, one wishes to obtain $x \approx d(e(x))$. A discovery of underlying rules and regularities is the typical goal for data mining applications, hence unsupervised learning is the main mode for Segment 2 from my private dissection of ML. Unsupervised methods are often used for data preprocessing in other ML scenarios, because most ML techniques suffer from curse of dimensionality effects and work better with lower-dimensional input data formats.

Reinforcement learning. The set-up for reinforcement learning (RL) is quite distinct from the above two. It is always related to an agent that can choose between different *actions* which in turn change the *state* of the environment the agent is in, and furthermore the agent may or may not receive *rewards* in certain environment states. RL thus involves at least the following three types of random variables:

- action random variables A ,
- world state random variables S ,
- reward random variables R .

In most cases the agent is modeled as a stochastic process: a temporal sequence of actions A_1, A_2, \dots leads to a sequence of world states S_1, S_2, \dots , which are associated with rewards R_1, R_2, \dots . The objective of RL is to learn a strategy (called *policy* in RL) for choosing actions that maximize

the reward accumulated over time. Mathematically, a policy is a conditional distribution of the kind

$$P(A_n = a_n |, S_1 = s_1, \dots, S_{n-1} = s_{n-1}; A_1 = a_1, \dots, A_{n-1} = a_{n-1}),$$

that is, the next action is chosen on the basis of the “lifetime experience” of previous actions and the resulting world states. RL is naturally connected to my Segment 4. Furthermore there are strong ties to neuroscience, because neuroscientists have reason to believe that individual neurons in a brain can adapt their functioning on the basis of neural or hormonal reward signals. Last but not least, RL has intimate mathematical connections to a classical subfield of control engineering called *optimal control*, where the (engineering) objective is to steer some system in a way that some long-term objective is optimized. An advanced textbook example is to steer an interplanetary missile from earth to some other planet such that fuel consumption is minimized. Actions here are navigation manoeuvres, the (negative) reward is fuel consumption, the world state is the missile’s position and velocity in interplanetary space.

The distinction between supervised and unsupervised learning is not clear-cut. Training tasks that are globally supervised (like TICS) may benefit from, or plainly require, unsupervised learning subroutines for transforming raw data into meaningfully compressed formats (like we saw in TICS). Conversely, globally unsupervised training mechanisms may contain supervised subroutines where intermediate “targets” y are introduced by the learning system. Furthermore, today’s advanced ML applications often make use of *semi-supervised* training schemes. In such approaches, the original task is supervised: learn some input-output model from labelled data (x_n, y_n) . This learning task may strongly benefit from including additional unlabelled input training points \tilde{x}_m , helping to distil a more detailed model of the input distribution P_X than would be possible on the basis of only the original x_n data. Again, TICS is an example: the data engineers who trained TICS used 70K un-captioned images in addition to the 30K captioned images to identify that 4096-dimensional manifold more accurately.

Also, reinforcement learning is not independent of supervised and unsupervised learning. A good RL scheme often involves supervised or unsupervised learning subroutines. For instance, an agent trying to find a good policy will benefit from data compression (= unsupervised learning) when the world states are high-dimensional; and an agent will be more capable of choosing good actions if it possesses an input-output model (= supervised learning) of the environment — inputs are actions, outputs are next states.

2 Decision trees and random forests

This section describes a class of machine learning models which is classical and simple and intuitive and useful. Decision trees and random forests are not super “fashionable” in these deep learning times, but practitioners in data analysis use them on a daily basis. The main inventions in this field have been made around the 1980-2000 decades. In this chapter I rely heavily on the decision tree chapters in the classical ML textbooks of Mitchell (1997) and Duda et al. (2001), and for the random forest part my source is the landmark paper Breiman (2001) which, as per today (Nov 6, 2019), has been cited 51022 times (Google Scholar). Good stuff to know, apparently.

Note: In this chapter of the lecture notes I will largely adhere to the notation used in Breiman (2001) in order to make it easier for you to read that key paper if you want to dig deeper. If in your professional future you want to use decision tree methods, you will invariably use them in random forests, and in order to understand what you are actually doing, you will have to read Breiman (2001) (like 50,000 others before you). Unfortunately, the notation used by Breiman is inconsistent, which makes the mathematical part of that paper hard to understand. My hunch is that most of the 50,000 readers skipped the two mathy sections and read only the experimental sections with the concrete hints for algorithms and gorgeous results. Inconsistent or even plainly incorrect mathematical notation (and mathematical thinking) happens a lot in “engineering applied math” papers and makes it difficult to really understand what the author wants to say (see my ramblings in my lecture notes on “Principles of Statistical Modeling”, Chapter 14, “A note on what you find in textbooks”, online at <http://minds.jacobs-university.de/teaching/ln/>). Therefore I will not use Breiman’s notation 1-1, but modify it a little to make it mathematically correct.

2.1 A toy decision tree

Figure 7 shows a simple decision tree. It can be used to *classify* fruit. If you have a fruit in your hands that you don’t know by name, then you can use this decision tree to find out what fruit you have there, in an obvious way:

- Start at the root node of the tree. The root node is labeled with a property that fruit have (`color?` is the root property used in the figure).
- Underneath the root node you find child nodes, one for each of the three possible color attributes *green*, *yellow*, *red*). Decide which color your fruit has and proceed to that child node. Let us assume that your fruit was yellow. Then you are now at the child node labelled `shape?`.
- Continue this game of moving downwards in the tree according to the direction decisions taken at each node according to the attributes of the fruit in

your hand. If you reach a leaf node of this tree, it will be labeled with the type of your fruit. If your fruit is yellow, round and small, it's a lemon!

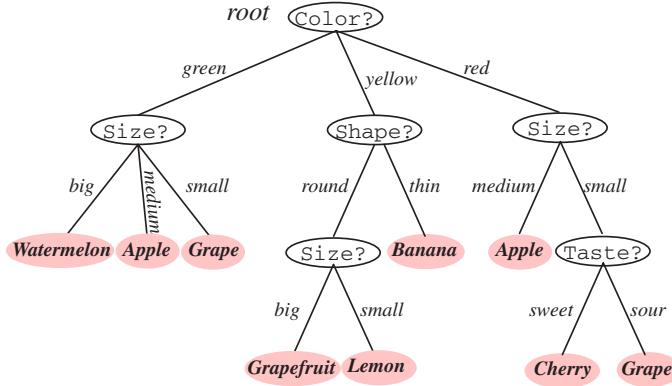


Figure 7: A simple decision tree. Taken from Duda et al. (2001).

A decision tree is not unique. You could, for instance, create another tree whose root node queries the **size?** property, and which would lead to exactly the same ultimate decision as the tree shown in the figure (exercise!).

If each internal (= non-leaf) node in a decision tree has exactly two child nodes, one speaks of a binary decision tree. These are particularly pleasant for mathematical analysis and algorithm design. Every decision tree can be transformed into an equivalent binary decision tree with an obvious procedure. Figure 8 shows the binary tree obtained from the fruit tree of Figure 7.

This is obviously a toy example. But decision trees can become really large and useful. A classical example are the botanic field guide books used by biologists to classify plants (Figure 9). I am a hobby botanist and own three such decision trees. Together they weigh about 1.5 kg and I have been using them ever since I was eighteen or so.

In the basic versions of decision tree learning, they are learnt from data which are given as class-labeled attribute vectors. For instance, training data for our fruit tree might look like this:

nr.	Color	Size	Taste	Weight	Shape	Class
1	green	big	sweet	heavy	round	Watermelon
2	green	?	?	heavy	round	Watermelon
3	red	small	sweet	light	round	Cherry
4	red	small	sweet	light	round	Apple
...
3000	red	big	?	medium	round	Apple

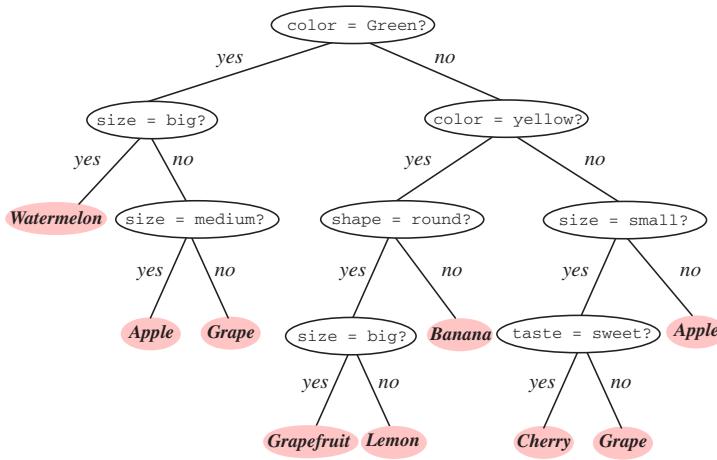


Figure 8: A binary version of the decision tree shown in Figure 7. Taken from Duda et al. (2001).

Here we get a first inkling that decision tree learning might not be as trivial as the final result in Figure 7 makes it appear. The above fruit training data table has missing values (marked by “?”); not all properties from the training data are used in the decision tree (property “Weight” is ignored); some examples of the same class have different attribute vectors (first two rows give different characterizations of watermelons); some identical attribute vectors have different classes (rows 3 and 4). In summary: real-world training data will be partly redundant, partly inconsistent and will be containing errors and gaps. All of this points in the same direction: statistical analyses will be needed to learn decision trees.

2.2 Formalizing “training data”

Before we describe decision tree learning, I want to squeeze in a little rehearsal of mathematical concepts and their notation.

In my notation in this chapter I will be leaning on Breiman’s notation as far as possible. Breiman’s notation is unfortunately incomplete and inconsistent. Here I use a complete, correct notation, adhering to the standards of mathematical probability theory. I am aware that many participants of this course will be unfamiliar with probability theory and its formal notation standards. Because machine learning ultimately rests on probability theory, I strongly recommend that at some point you learn to master the basics of probability theory (for instance, from my lecture notes on the legacy course “Principles of Statistical Modeling”, online: <http://minds.jacobs-university.de/teaching/ln/>). Only if you understand probability, you can get a full grasp on machine learning. But it is clear that many students will not have the time or inclination to learn about probability theory.

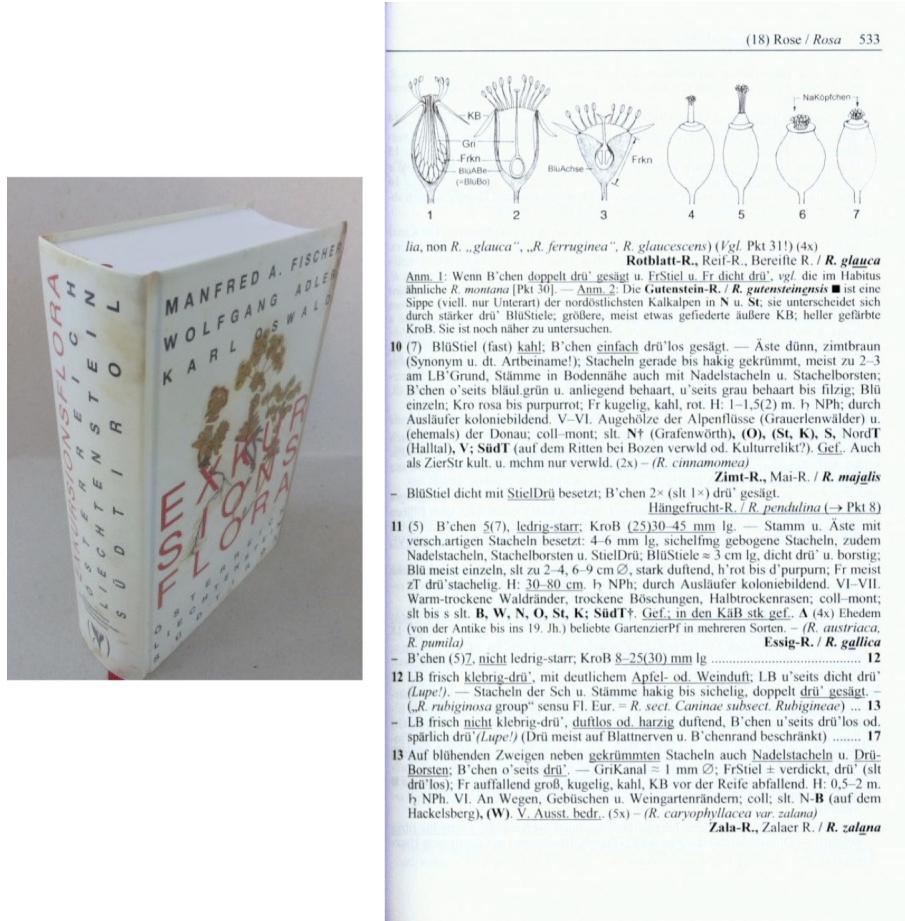


Figure 9: Another decision tree. The right image shows a page from this botany field guide. What you see on this page is, in fact, a small section of a large binary classification tree. Image source: iberlibro.com, booklooker.de.

Therefore I will try to introduce notation and concepts in two ways throughout this course: in a rigorous “true math” version, and in an intuitive “makes some sense” version.

I called the fruit attribute data table above a *sample*. Samples are mathematical objects of key importance in statistics and machine learning (where they are also called “training data”). Samples are always connected with *random variables* (RVs). Here is how.

First, the intuitive version. As an empirical fruit scientist, you would obtain a “random draw” to get the training data table in a very concrete way: you would go to the fruit market, collect 3000 fruit “at random”, observe and note down their color, size, taste, weight and shape attributes in an Excel table, and for each of the 3000 fruit you also ask the fruit vendor for the name of the fruit to get the

correct class label which you also note down in the last column of the table.

The mathematical representation of observing and noting down the attributes of the i -th fruit that you have picked (where $i = 1, \dots, 3000$) is $X_i(\omega)$. X_i is the random variable which is the mathematical model of the observation procedure – X_i could be described as the procedure “pick a random fruit and observe and report the attribute vector”. The ω in $X_i(\omega)$ is the occasion when you actually executed the procedure X_i – say, ω stands for the concrete data collection event when you went to the Vismarkt in Groningen last Tuesday and visited the fruit stands. If the entire procedure of collecting 3000 fruit specimen is executed on another occasion – for instance, a week earlier, or by your friend on the same day – this would be mathematically represented by another ω . For instance, $X_i(\omega)$ might be the attribute vector that you observed for the i -th fruit last Tuesday, $X_i(\omega')$ would be the attribute vector that you observed for the i -th fruit one week earlier, $X_i(\omega'')$ would be the attribute vector that your friend observed for the i -th last Tuesday when he did the whole thing in parallel with you, etc. In mathematical terminology, these “observation occasions” ω are called *elementary events*.

Similarly, $Y_i(\omega)$ is the fruit class name that you were told by the vendor when you did the data sampling last Tuesday; $Y_i(\omega')$ would be the i th fruit name you were told when you did the whole exercise a week earlier already, etc.

The three thousand $X_i(\omega), Y_i(\omega)$ correspond to the rows in the data table. I will call each of these $X_i(\omega), Y_i(\omega)$ a *data point*. To get the entire table as a single mathematical object – namely, the sample – one combines these single fruit data points by a product operation, obtaining $(\mathbf{X}(\omega), \mathbf{Y}(\omega))$, where $\mathbf{X} = \bigotimes_{i=1, \dots, N} X_i$ and $\mathbf{Y} = \bigotimes_{i=1, \dots, N} Y_i$.

And here is the rigorous probability theory account of the $(\mathbf{X}(\omega), \mathbf{Y}(\omega))$ notation. As always in probability theory and statistics, we have an underlying probability space $(\Omega, \mathfrak{A}, P)$. In this structure, Ω is the set of all possible *elementary events* $\omega \in \Omega$. \mathfrak{A} is a σ -field on Ω , and P is a *probability measure* on (Ω, \mathfrak{A}) . The random variable (X) is a function which returns *samples*, that is collections of N training data points. We assume that all samples which could be drawn have N data points; this assumption is a matter of convenience and simplifies the ensuing notation and mathematical analysis a lot. – It is a good exercise to formalize the structure of a sample in more detail. A single data point consists of an attribute vector \mathbf{x} and a class label y . In formal notation, we have m properties Q_1, \dots, Q_m , and property Q_i has a set A_i of possible attribute values. Thus, $\mathbf{x} \in A_1 \times \dots \times A_m$. We denote the set of possible class labels by C . A data point $(X_i(\omega), Y_i(\omega))$ is thus an element of the set $(A_1 \times \dots \times A_m \times C)$. See Appendix A for the mathematical notation used in creating product data structures. We denote the *sample space* of any random variable Z by S_Z . The sample space for X_i is thus $S_{X_i} = A_1 \times \dots \times A_m$ and for Y_i it is $S_{Y_i} = C$. Because $S_{X_i} = S_{X_j}$, $S_{Y_i} = S_{Y_j}$ for all $1 \leq i, j \leq N$, for simplicity we also write S_X for $A_1 \times \dots \times A_m$ and S_Y for C .

The entire training data table is an element of the sample space $S_{\mathbf{X} \otimes \mathbf{Y}}$ of the product RV $\mathbf{X} \otimes \mathbf{Y} = \bigotimes_{i=1, \dots, N} X_i \otimes \bigotimes_{i=1, \dots, N} Y_i$. Just to exercise formalization

skills: do you see that

$$S_{\mathbf{X}} \times S_{\mathbf{Y}} = (A_1 \times \dots \times A_m)^N \times C^N = (A_1 \times \dots \times A_m \times C)^N = S_{\mathbf{X} \otimes \mathbf{Y}}?$$

To round off our rehearsal of elementary concepts and notation, I repeat the basic connections between random variables, probability spaces and sample spaces: a random variable Z always comes with an underlying probability space $(\Omega, \mathfrak{A}, P)$ and a sample space S_Z . The RV is a *function* $Z : \Omega \rightarrow S_Z$, and induces a *probability distribution* on S_Z . This distribution is denoted by P_Z .

I torture you with these exercises in notation like a Russian piano teacher will torture his students with technical finger-exercising *études*. It is technical and mechanical and the suffering student may not happily appreciate the necessity of it all, – and yet this torture is a precondition for becoming a virtuoso. The music of machine learning is played in tunes of probability, no escape.

I am aware that, if you did not know these probability concepts before, this condensed rehearsal cannot possibly be understandable. Probability theory is one of the most difficult to learn sectors of mathematics and it takes weeks of digesting and exercising to embrace the concepts of probability spaces, random variables and sample spaces (not to speak of σ -fields). I will give a condensed probability basics tutorial in the tutorial sessions accompanying this course if there is a demand. Besides that I want to recommend my lecture notes “Principles of Statistical Modeling” which I mentioned before. They document a graduate course whose main purpose was to give a detailed, understandable, slow enough, fully correct introduction to the concepts of probability theory and statistics.

2.3 Learning decision trees: setting the stage

After this excursion into the proper use of notation, we can start to discuss decision tree learning.

Despite the innocent looks of decision trees, no universally “optimal” method is known how to learn them from training data. But it is known what are the basic options one has in designing a decision tree learning algorithm. There are a few key decisions the algorithm designer has to make in order to assemble a learning algorithm for decision trees, and standard algorithmic building blocks for each of the design decisions are known. We will now present these design options in turn.

Two things must be given before a learning algorithm can be assembled and learning can start: (A) the training data, and (B) an optimization criterion. While (A) is obvious, (B) deserves a few words of explanation.

We observe that decision tree learning is a case of supervised learning. The training data is a collection of labeled samples $(x_i, y_i)_{i=1,\dots,N} = (X_i(\omega), Y_i(\omega))_{i=1,\dots,N}$, where $x_i \in A_1 \times \dots \times A_m = S_X$ is an attribute vector and $y_i \in C = S_Y$ is a class label. A decision tree represents a function $h : S_X \rightarrow S_Y$ in an obvious way: using the sequential decision procedure outlined in Section 2.1, a vector of observed attributes leads you from the root node to some leaf, and that leaf gives a class label.

In statistics, such functions $h : S_X \rightarrow S_Y$ are generally called *decision functions* (also in other supervised learning settings that do not involve decision trees), a term that I will also sometimes use.

Decision tree learning is an instance of the general case of supervised learning: the training data are labeled pairs $(\mathbf{x}_i, y_i)_{i=1,\dots,N}$, with $\mathbf{x}_i \in S_X, y_i \in S_Y$, and the learning aims at finding a decision function $h : S_X \rightarrow S_Y$ which is “optimal” in some sense.

In what sense can such a function $h : S_X \rightarrow S_Y$ (in this section: such a decision tree) be “optimal”? How can one quantify the “goodness” of functions of the type $h : S_X \rightarrow S_Y$?

This is a very non-trivial question as we will learn to appreciate as the course evolves. For supervised learning tasks, the key to optimizing a learning procedure is to declare a *loss function* at the very outset of a learning project. A loss function is a function

$$L : S_Y \times S_Y \rightarrow \mathbb{R}^{\geq 0}, \quad (1)$$

which assigns a nonnegative real number to a pair of class labels. The loss function is used to compare the classification $h(\mathbf{x})$, returned by a decision function h on input \mathbf{x} , with the correct value. After having learnt a decision function h from a training sample $S_X \times S_Y$, one can quantify the performance of h averaged over the training data, obtaining a quantity R^{emp} that is called the *empirical risk*:

$$R^{\text{emp}}(h) = \frac{1}{N} \sum_{i=1,\dots,N} L(h(\mathbf{x}_i), y_i). \quad (2)$$

Most supervised machine learning procedures are built around some algorithmic optimization procedure which searches for decision functions which minimize the empirical risk, that is, which try to give a small average loss on the training data.

I emphasize that minimizing the empirical loss (or in another common wording, minimizing the “training error”) by a clever learning algorithm will usually not lead to a very useful decision function. This is due to the problem of *overfitting*. “Overfitting” means, intuitively speaking, that if the learning algorithm tries everything to be good (= small loss) on the training data, it will attempt to minimize the loss individually for each training data point. This means, in some way, to learn the training data “by heart” – encode an exact memory of the training data points in the learnt decision function h . This is not a good idea because later “test” data points will be different, and the decision function obtained by “rote learning” will not know how to *generalize* to the new data points. The ultimate goal of supervised machine learning is not to minimize the empirical loss (that is, to have small loss on the training data), but to minimize the loss on future “test” data which were not available for training. Thus, the central optimization criterion for supervised learning methods is to find decision functions h which have a small *risk*

$$R(h) = E[L(h(X), Y)], \quad (3)$$

where E is the expectation operator (see Appendix D). That is, a good decision function h should incur a small expected loss – it should have small “testing error”. At a later point in this course we will analyse the problem of avoiding overfitting in more detail. For now it’s enough to be aware that overfitting is an important threat, and that in order to avoid it one should not allow the models h to dress themselves too closely around the training data points.

For a classification task with a finite number of class labels, like in our fruit example, a natural loss function is one that simply counts misclassifications:

$$L^{\text{count}}(h(\mathbf{x}), y) = \begin{cases} 0, & \text{if } h(\mathbf{x}) = y \\ 1, & \text{if } h(\mathbf{x}) \neq y \end{cases} \quad (4)$$

This counting loss is often a natural choice, but in many situations it is not appropriate. Consider, for instance, medical diagnostic decision making. Assume you visit a doctor with some vague complaints. Now compare two scenarios.

- Scenario 1: after doing his diagnostics the doctor says, “sorry to tell you but you have cancer and you should think about making your will” – and this is a wrong diagnosis; in fact you are quite healthy.
- Scenario 2: after doing his diagnostics the doctor says, “good news, old boy: there’s nothing wrong with you except you ate too much yesterday” – and this is a wrong diagnosis; in fact you have intestinal cancer.

These two errors are called the “false positive” and the “false negative” decisions. A simple counting loss would optimize medical decision making such that both kinds of errors are minimized. But in medicine, false negatives should be avoided as much as possible because their consequences can be lethal, whereas false positives will only cause a passing anxiety and inconvenience. Accordingly, a loss function used to optimize medical decision making should put a higher penalty (= larger L values) on false negatives than on false positives.

Generally, and specifically so in operations research and decision support systems, loss functions can become very involved, including a careful balancing of conflicting ethical, financial or other factors. However, we will not consider complex loss functions here and stick to the simple counting loss. This loss is the one that guided the design of the classical decision tree learning algorithms. And now, finally, we have set the stage for actually discussing learning algorithms for decision trees!

2.4 Learning decision trees: the core algorithm

Ok., let us go concrete. We are given training data (\mathbf{x}_i, y_i) (in the form of an Excel table, in real life), and – opting for the counting loss – we want to find an optimal (more realistically: a rather good) decision tree h^{opt} which will minimize misclassifications on new “test” data.

In order to distil this h^{opt} from the training data, we need to set up a *learning algorithm* which, on input (\mathbf{x}_i, y_i) , outputs h^{opt} .

All known learning algorithms for decision trees incrementally build h^{opt} from the root node downwards. The first thing a decision tree learning algorithm (DTLA) must do is therefore to decide which property is queried first, making it the root node.

To understand the following procedures, notice that a decision tree iteratively splits the training dataset in increasingly smaller, disjoint subsets. The root node can be associated with the entire training dataset – call it D . If the root node ν_{root} queries the property Q_j and this property has k attributes $a_1^j, \dots, a_{k_j}^j$, there will be k child nodes ν_1, \dots, ν_k where node ν_l will be covering all training datapoints that have attribute a_l^j ; and so forth down the tree. In detail: if $\nu_{l_1 l_2 \dots l_r}$ is a tree node at level r (counting from the root), and this node is associated with the subset $D_{l_1 l_2 \dots l_r}$ of the training data, and this node queries property Q_u which has attributes $a_1^u, \dots, a_{k_u}^u$, then the child node $\nu_{l_1 l_2 \dots l_r l_s}$ will be that subset of $D_{l_1 l_2 \dots l_r}$ which contains those training data points that have attribute a_s^u of property Q_u .

The classical solution to this problem of selecting a property Q for the root node is to choose that property which leads to “maximally informative” split of the training dataset in the first child node level. Intuitively, the data point subsets associated with the child nodes should be as “pure” as possible with respect to the classes $c \in C$.

In the best of all cases, if there are q different classes (that is, $|C| = q$), there would be a property $Q_{\text{supermagic}}$ with q attributes which already uniquely identify classes, such that each first-level child node is already associated with a “pure” set of training examples all from the same class – say, the first child node covers only apples, the second only bananas, etc.

This will usually not be possible, among other reasons because normally there is no property with exactly as many attributes as there are classes. Thus, “purity” of data point sets associated with child nodes needs to be measured in a way that tolerates class mixtures in each node. The measure that is traditionally invoked in this situation comes from information theory. It is the *entropy* of the class distribution within a child node. If D_l is the set of training points associated with a child node ν_l , and n_i^l is the number of data points in D_l that are from class i (where $i = 1, \dots, q$), and the total size of D_l is n^l , then the entropy S_l of the “class mixture” in D_l is given by

$$S_l = - \sum_{i=1, \dots, q} \frac{n_i^l}{n^l} \log_2 \left(\frac{n_i^l}{n^l} \right). \quad (5)$$

If in this sum a term $\frac{n_i^l}{n^l}$ happens to be zero, by convention the product $\frac{n_i^l}{n^l} \log_2 \left(\frac{n_i^l}{n^l} \right)$ is set to zero, too. I quickly rehearse two properties of entropy which are relevant here:

- Entropies are always nonnegative.

- The more “mixed” the set D_l , the higher the entropy S_l . In one extreme case, D_l is 100% clean, that is, it contains only data points of a single class. Then $S_l = 0$. The other extreme is the greatest possible degree of mixing, which occurs when there is an equal number of data points from each class in D_l . Then S_l attains its maximal possible value of $-q \log_2(1/q) = q \log q$.

The entropy of the root node is

$$S_{\text{root}} = - \sum_{i=1,\dots,q} \frac{n_i^{\text{root}}}{N} \log_2 \left(\frac{n_i^{\text{root}}}{N} \right),$$

where n_i^{root} is the number of examples of class i in the total training data set D and N is the number of training data points. Intuitively, S_{root} is a measure of the *impurity* of the training data set. Following the terminology of Duda et al. (2001), I will denote the entropy-based impurity measure of the root by $i_{\text{entropy}}(\nu_{\text{root}}) := S_{\text{root}}$.

More generally, if ν is any node in the tree, and this node is associated with the training data point set D_ν , and $|D_\nu| = n$, and the q classes are represented in D_ν by subsets of size n_1, \dots, n_q , the entropy impurity of ν is given by

$$i_{\text{entropy}}(\nu) = - \sum_{i=1,\dots,q} \frac{n_i}{n} \log_2 \left(\frac{n_i}{n} \right). \quad (6)$$

If the root node ν_{root} queries the property Q_j and this property has k attributes, then the mixing of classes averaged over all child nodes ν_1, \dots, ν_k is given by

$$\sum_{l=1,\dots,k} \frac{|D_l|}{N} i_{\text{entropy}}(\nu_l),$$

where D_l is the point set associated with ν_l .

Subtracting this average class mixing found in the child nodes from the class mixing in the root, one obtains the *information gain* achieved by opting for property Q_j for the root node:

$$\Delta i_{\text{entropy}}(\nu_{\text{root}}, Q_j) = i_{\text{entropy}}(\nu_{\text{root}}) - \sum_{l=1,\dots,k} \frac{|D_l|}{N} i_{\text{entropy}}(\nu_l). \quad (7)$$

It can be shown that this quantity is always nonnegative. It is maximal when all child nodes have “class-pure” data point sets associated with them.

For a generic node ν labeled with property Q , where Q has k attributes, and the size of the set associated with ν is n and the sizes of the sets associated with the k child nodes ν_1, \dots, ν_k are n_1, \dots, n_k , the information gain for node ν is

$$\Delta i_{\text{entropy}}(\nu, Q) = i_{\text{entropy}}(\nu) - \sum_{l=1,\dots,k} \frac{n_l}{n} i_{\text{entropy}}(\nu_l). \quad (8)$$

The procedure to choose a property for the root node is to compute the information gain IG for all properties and select the one which maximizes the information gain.

This procedure of choosing that query property for a node which leads to the greatest information gain is repeated tree-downwards as the tree is grown by the DTLA. A node is not further expanded and thus becomes a leaf node if (i) either the training dataset associated with this node is 100% pure (contains only examples from a single class), or if (ii) one has reached the level q , that is, all available properties have been queried on the path from the root to that node. In case (i) the leaf is labeled with the unique class of its associated data point set. In case (ii) it is labeled with the class that has the largest number of representatives in the associated data point set.

Information gain is the most popular and historically first criterion used for determining the query property of a node. But other criteria are used too. I mention three.

The first is a normalized version of the information gain. The motivation is that the information gain criterion (8) favours properties that have more attributes over properties with fewer attributes. This requires a little explanation. Generally speaking, properties with many attributes statistically tend to lead to purer data point sets in the children nodes than properties with only few attributes. To see this, compare the extreme cases of a property with only a single attribute, which will lead to a zero information gain, with the extreme case of a property that has many more attributes than there are data points in the training dataset, which will (statistically) lead to many children node data point sets which contain only a single example, that is, they are 100% pure. A split of the data point set D_ν into a large number of very small subsets is undesirable because it is a door-opener for overfitting. This preference for properties with more attributes can be compensated if the information gain is normalized by the entropy of the data set splitting, leading to the *information gain ratio* criterion, which here is given for the root node:

$$\Delta_{\text{ratio}} i_{\text{entropy}} = \frac{\Delta i_{\text{entropy}}(\nu_{\text{root}}, Q_j)}{- \sum_{l=1, \dots, k} \frac{|D_l|}{N} \log_2 \frac{|D_l|}{N}}. \quad (9)$$

The second alternative that I mention measures the impurity of a node by its *Gini impurity*, which for a node ν associated with set D_ν , where $|D_\nu| = n$ and the subsets of points in D_ν of classes $1, \dots, q$ have sizes n_1, \dots, n_q , is

$$i_{\text{Gini}}(\nu) = \sum_{1 \leq i, j \leq q; i \neq j} \frac{n_i}{n} \frac{n_j}{n} = 1 - \sum_{1 \leq i \leq q} \left(\frac{n_i}{n} \right)^2,$$

which is the error rate when a category decision for a randomly picked point in D_ν is made randomly according to the class distribution within D_ν .

The third alternative is called the *misclassification impurity* in Duda et al.

(2001) and is given by

$$i_{\text{misclass}}(\nu) = 1 - \max\left\{\frac{n_l}{n} \mid l = 1, \dots, q\right\}.$$

This impurity measure is the minimum (taken over all classes c) probability that a point from D_ν which is of class c is misclassified if the classification is randomly done according to the class distribution in D_ν .

Like the entropy impurity, the Gini and misclassification impurities are non-negative and equal to zero if and only if the set D_ν is 100% pure. Like it was done with the entropy impurity, these other impurity measures can be used to choose a property for a node ν through the gain formula (8), plugging in the respective impurity measure for i_{entropy} .

This concludes the presentation of the core DTLA. It is a *greedy* procedure which incrementally constructs the tree, starting from its root, by always choosing that property for the node currently being constructed which maximizes the information gain (7) or one of its versions obtained from alternative impurity measures.

You will notice two unfortunate facts:

- An optimization scheme which works by iteratively applying *local* greedy optimization steps will generally not yield a *globally* optimal solution.
- We started from claiming that our goal is to learn a tree which minimizes the count loss (or any other loss we might opt for). However, none of the impurity measures and the associated local gain optimization is connected in a mathematically transparent way with that loss. In fact, no computationally tractable method for learning an empirical-loss-minimal tree exists: the problem is NP-complete (Hyafil and Rivest, 1976).
- It is not clear which of the impurity measures is best for the goal of minimizing the empirical loss.

In summary, we have here a *heuristic* learning algorithm (actually, a family of algorithms, depending on the choice of impurity measure) which is based on intuition and the factual experience that it works reasonably well in practice. Only heuristic algorithms are possible due to the NP-completeness of the underlying optimization problem.

2.5 Dealing with overfitting

If a decision tree is learnt according to the core algorithm, there is the danger that it overfits. This means that it performs well on the training data – that is, it has small empirical loss – which boils down to a condition where all leaf nodes are rather pure. A zero empirical loss may, for instance, be attained if the number of properties and attributes is large. Then it will easily be the case that every training example has a unique combination of attributes, which leads to 100% pure

leaf nodes, and every leaf having only a small set of training examples associated with it, maybe singleton sets. Then one has zero misclassifications of the training examples. But, intuitively speaking, the tree has just memorized the training set. If there is some “noise” in the attributes of data points, new examples (not in the training set) will likely have attribute combinations that lead the learnt decision tree on wrong tracks.

We will learn to analyze and fight overfitting later in the course. At this point I only point out that if a learnt tree T overfits, one can typically obtain from it another tree T' which overfits less, by *pruning* T . Pruning a tree means to select some internal nodes and delete all of their children. This makes intuitive sense: the deeper one goes down in a tree learnt by the core algorithm, the more does the branch reflect “individual” attribute combinations found in the training data.

To make this point particularly clear, consider a case where there are many properties, but only one of them carries information relevant for classification, while all others are purely random. For instance, for a classification of patients into the two classes “healthy” and “has_cancer”, the binary property “has_increased_leukocyte_count” carries classification relevant information, while the binary property “given_name_starts_with_A” is entirely unconnected to the clinical status. If there are enough such irrelevant properties to uniquely identify each patient entered into the training sample, the core algorithm will (i) most likely find that the relevant property “has_increased_leukocyte_count” leads to the greatest information gain at the root and thus use it for the root decision, and (ii) subsequently generate tree branches that lead to 100% pure leaf nodes. Zero training error and poor generalization to new patients results. The best tree here would be the one that only expands the root node once, exploiting the only relevant property.

With this insight in mind, there are two strategies to end up with trees that are not too deep.

The first strategy is *early stopping*. One does not carry the core algorithm to its end but at each node which one has created, one decides whether a further expansion would lead to overfitting. A number of statistical criteria are known to base this stopping on; or one can use cross-validation (explained in future chapters in these lecture notes). We do not discuss this further here – the textbook by Duda explains a few of these statistical criteria. A problem with early stopping is that it suffers from the *horizon effect*: if one stops early at some node, a more fully expanded tree might exploit further properties that are, in fact, relevant for classification refinement underneath the stopped node.

The second strategy is *pruning*. The tree is first fully built with the core algorithm, then it is incrementally shortened by cutting away end sections of branches. An advantage of pruning over early stopping is that it avoids the horizon effect. Duda says (I am not a decision tree expert and can’t judge) that pruning should be preferred over early stopping “in small problems” (whatever “small” means).

2.6 Variants and refinements

The landscape of decision tree learning is much richer than what I can explore in a single lecture. Two important topics which I omit (treated briefly in Duda's book):

Missing values. Real-world datasets will typically have missing values, that is, not all training or test examples will have attribute values filled in for all properties. Missing values require adjustments both in training (an adapted version of impurity measures which accounts for missing values) and in testing (if a test example leads to a node with property Q and the example misses the required attribute for Q, the normal classification procedure would abort). The Duda book dedicates a subsection to recovery algorithms.

Numerical properties. Many real-world properties, like for instance "velocity" or "weight", have a continuous numerical range for attribute values. This requires a split of the continuous value range into a finite number (often two) of discrete bins which then serve as discrete attributes. For instance, the range of a "body_weight" property might be split into two ranges " $<50\text{kg}$ " and " $\geq 50\text{kg}$ ", leading to a new property with only two attributes. These ranges can be further split into finer-grained subproperties which would be queried downstream in the tree, for instance querying for " $<45\text{kg}$ " vs. " $45\text{kg} \leq 50\text{kg}$ " underneath the " $<50\text{kg}$ " node. Again, heuristics must be called upon to determine splitting boundaries which lead to well-performing (correct and cheap = not too many nodes) trees. Again – consult Duda.

All in all, there is a large number of design decisions to be made when setting up a decision tree learning algorithm, and a whole universe of design criteria and algorithmic sub-procedures is available in the literature. Some combinations of such design decisions have led to final algorithms which have become branded with names and widely used. Two algorithms which are invariably cited (and which are available ready-made in professional toolboxes) are the ID3 algorithm and its more complex and higher-performant successor, the C4.5 algorithm. They had been introduced by the decision tree pioneer Ross Quinlan in 1986 and 1993, respectively. The Duda book gives brief descriptions of these canonical algorithms.

2.7 Random forests

Decision tree learning is a subtle affair. If one designs a DTLA by choosing a specific combination of the manifold design options, the tree that one gets from a training dataset D will be influenced by the effects of that choice of options. It will not be "the optimal" tree which one might obtain by some other choice of options. Furthermore, if one uses pruning or early stopping to fight overfitting, one will likely end up with trees that, while not overfitting, are underfitting – that is, they do not exploit all the information that is hidden in the training data.

In summary, whatever one does, one is likely to obtain a decision tree that is significantly sub-optimal.

This is a common situation in machine learning. There are only very few machine learning techniques where one has full mathematical control over getting the best possible model from a given training dataset, and decision tree learning is not among them. Fortunately, there is also a common escape strategy for minimizing the quality deficit inherent in most learning designs: *ensemble methods*. The idea is to train a whole collection (called “ensemble”) of models (here: trees), each of which is likely suboptimal (jargon: each model is obtained from a “weak learner”), but if their results on a test example are diligently combined, the merged result is much better than what one gets from each of the models in the ensemble. It’s the idea of crowd intelligence.

“Ensemble methods” is an umbrella term for a wide range of concrete techniques. They differ, obviously, in the kind of the individual models, like for instance decision trees vs. neural networks. Second, they vary in the methods of how one generates diversity in the ensemble. Ensemble methods work well only to the extent that the individual models in the ensemble probe different aspects in the training data – they should look at the data from different angles, so to speak. Thirdly, ensemble methods differ in the way how the different results of the individual models are combined. A general theory for setting up an ensemble learning scheme is not available – we are again thrown back to heuristics. The Wikipedia articles on “Ensemble learning” and “Ensemble averaging (machine learning)” give a condensed overview.

Obviously, ensemble methods can only be used when the computational cost of training a single model is rather low. This is the case for decision tree learning. Because training an individual tree will likely not give a competitive result, but is cheap, it is common practice to train not a single decision tree but an entire ensemble – which in the case of tree learning is called a *random forest*. In fact, random forests can yield competitive results (for instance, compared to neural networks) at moderate cost, and are therefore often used in practical applications.

The definite reference on random forests is Breiman (2001). This paper has been cited more than 50,000 times on Google Scholar. This places it on a par even with the most cited works from the deep learning field. The paper has two parts. In the first part, Breiman gives a mathematical analysis of why combining decision trees does not lead to overfitting, and derives an instructive upper bound on the generalization error (= risk, see Section 2.3). These results were in synchrony with mainstream theory work in other areas of machine learning at the time and made this paper *the* theory anchor for random forests. However, I personally find the math notation used by Breiman opaque and hard to penetrate, and I am not sure how many readers would understand it. The second, larger part of this paper describes several variants and extensions of random forest algorithms, discusses their properties and benchmarks some of them against the leading classification learning algorithms of the time, with favourable outcomes. This part is an easy

read and the presented algorithms are not difficult to implement. My hunch is that it is the second rather than the first part which led to the immense impact of this paper.

Here I give a summary of the paper, in reverse order, starting with the practical algorithm recommended by Breiman. After that I give an account of the most conspicuous theoretic results in transparent, standard probability notation.

In ensemble learning one must construct many *different* models in an automated fashion. One way to achieve this is to employ a *stochastic* learning algorithm. A stochastic learning algorithm can be seen as a learning algorithm which takes two arguments. The first argument is the training data D , the same as in ordinary, non-stochastic learning algorithms. The second argument is a random vector, denoted by Θ in Breiman's paper, which is set to different random values in each run of the algorithm. Θ can be seen as a vector of control parameters in the algorithm; different random settings of these control parameters lead to different outcomes of the learning algorithm although the training data are always the same, namely D .

Breiman proposes two ways in order to make the tree learning stochastic:

Bagging. In each run of the learning algorithm, the training dataset is *resampled with replacement*. That is, from D one creates a new training dataset D' of the same size as D by randomly copying elements from D into D' . This is a general approach for ensemble learning, called *bagging*. The Wikipedia article on "Bootstrap aggregating" gives an easy introduction if you are interested in learning more.

Random feature selection is the term used by Breiman for a randomization technique where, at each node whose query property has to be chosen during tree growing, a small subset of all still unqueried properties is randomly drawn as candidates for the query property of this node. The winning property among them is determined by the information gain criterion.

Combining these two randomness-producing mechanisms with a number of other design decisions not mentioned here, Breiman obtains a stochastic learning algorithm which, with ensembles of size 100, outperformed other methods that were state-of-the-art at that time.

The random vector Θ , which is needed for a concise specification and a mathematical analysis of the resulting stochastic tree learning algorithm, is some essentially arbitrary encoding of the random choices made for bagging and random feature selection.

The main theoretical result in Breiman's paper is his Theorem 2.3:

$$PE^* \leq \bar{\varrho} (1 - s^2) / s^2.$$

This theorem certainly added much to the impact of the paper, because it gives an intuitive guidance for the proper design of random forests, and also because it

connected random decision tree forests to other machine learning methods which were being mathematically explored at the time when the paper appeared. I will now try to give a purely intuitive explanation, and then conclude this section with a clean-math explanation of this theorem.

- The quantity PE^* , called *generalization error* by Breiman, is the probability that a random forest (in the limit of its size going to infinity) makes wrong decisions. Obviously one wants this probability to be as small as possible.
- The quantity $\bar{\rho}$ is a certain statistical correlation measure which measures how similar, on average across the tree population in a forest, the various trees will arrive at different classifications on test data points. This measure is maximal if all trees always yield the same classification results, and it is minimal if, roughly speaking, their classification results are statistically independent.
- The quantity s , which satisfies $0 \leq s \leq 1$, called by Breiman the *strength* of the stochastic tree learning algorithm, measures a certain “discerning power” of the average tree in a forest, that is, how large is the probability gap between an average tree making the right decision and the probability for the tree to make a wrong decision. s ranges in $[0, 1]$ and if the strength is equal to the maximal value of 1, then all trees in the forest always make correct decisions on all test data points. If it is zero, then the trees make more incorrect decisions than correct ones on average across the forest and test data points.

Note that the factor $(1-s^2)/s^2$ ranges between zero and infinity and is monotonously increasing with decreasing s . It is zero with maximal strength $s = 1$, and it is infinite if the strength is zero.

The theorem gives an upper bound on the generalization error observed in (asymptotically infinitely large) random forests. The main message is that this bound is a product of a factor $\bar{\rho}$ which is smaller when the different trees in a forest vary more in their response to test inputs, with a factor $(1-s^2)/s^2$ which is smaller when the trees in the forest place a larger probability gap between correct and the second most common decision across the forest. The suggestive message of all of this is that in designing a stochastic decision tree learning algorithm one should

- aim at maximizing the response variability across the forest, while
- attempting to ensure that trees mostly come out with correct decisions.

If one succeeds to generate trees that always give correct decisions, one has maximal strength $s = 1$ and the generalization error is obviously zero. This will usually be impossible. Instead, the stochastic tree learning algorithm will produce trees that have a residual error probability, that is, $s < 1$. Then, the first factor

implies that one should aim at a stochastic tree generation mechanism which (while fixing the strength) show great variability in their response behavior.

The remainder of this section is only for those of you who are familiar with probability theory, and this material will not be required in the final exam. I will give a mathematically transparent account of Breiman's Theorem 2.3. This boils down to an exercise in clean mathematical formalism. We start by formulating the ensemble learning scenario in rigorous probability terms. There are two underlying probability spaces involved, one for generating data points, and the other for generating the random vectors Θ . I will denote these two spaces by $(\Omega, \mathfrak{A}, P)$ and $(\Omega^*, \mathfrak{A}^*, P^*)$ respectively, with the second one used for generating Θ . The elements of Ω, Ω^* will be denoted by symbols ω, ω^* , respectively.

Let X be the RV which generates attribute vectors,

$$X : \Omega \rightarrow A_1 \times \dots \times A_m,$$

and Y the RV which generates class labels,

$$Y : \Omega \rightarrow C.$$

The random parameter vectors Θ can be vectors of any type, numerical or categorical, depending on the way that one chooses to parametrize the learning algorithm. Let T denote the space from which Θ 's can be picked. For a forest of K trees one needs K random vectors Θ_j , where $j = 1, \dots, K$. Let

$$Z_j : \Omega^* \rightarrow T$$

be the RV which generates the j -th parameter vector, that is, $Z_j(\omega^*) = \Theta_j$. The RVs Z_j ($j = 1, \dots, K$) are independent and identically distributed (iid).

In the remainder of this section, we fix some training dataset D . Let $h(\cdot, \Theta)$ denote the tree that is obtained by running the stochastic tree learning algorithm with parameter vector Θ . This tree is a function which outputs a class decision if the input is an attribute vector, that is,

$$h(\mathbf{x}, \Theta) \in C.$$

When Θ is fixed, we can view h as a function of the RV X , thus

$$h(X, \Theta) : \Omega \rightarrow C$$

is a random variable.

Now the stage is prepared to re-state Breiman's central result (Theorem 2.3) in a transparent notation.

Breiman starts by introducing a function mr , called *margin function for a random forest*,

$$\begin{aligned} mr : (A_1 \times \dots \times A_m) \times C &\rightarrow \mathbb{R} \\ (\mathbf{x}, y) &\mapsto P^*(h(\mathbf{x}, Z) = y) - \max_{\tilde{y} \neq y} \{P^*(h(\mathbf{x}, Z) = \tilde{y})\}, \end{aligned}$$

where Z is a random variable distributed like the Z_j .

If you think about it you will see that $mr(\mathbf{x}, y) > 0$ if and only iff a very large forest (in the limit of its size going to infinity) will classify the data point (\mathbf{x}, y) correctly, and $mr(\mathbf{x}, y) < 0$ if the forest will come to a wrong collective decision.

On the basis of this margin function, Breiman defines the *generalization error* PE^* by

$$PE^* = P(mr(X, Y) < 0).$$

The expectation of mr over data examples is

$$s = E[mr(X, Y)].$$

s is a measure to what extent, averaged over data point examples, the correct answer probability (over all possible decision trees) is larger than the highest probability of deciding for any other answer. Breiman calls s the *strength* of the parametrized stochastic learning algorithm. The stronger the stochastic learning algorithm, the greater the probability margin between correct and wrong classifications on average over data point examples.

Breiman furthermore introduces a *raw margin function* rmg_Θ , which is a function of X and Y parametrized by Θ , through

$$rmg_\Theta(\omega) = \begin{cases} 1, & \text{if } h(X(\omega), \Theta) = Y(\omega), \\ -1, & \text{if } h(X(\omega), \Theta) \text{ gives the maximally probable among the} \\ & \text{wrong answers in an asymptotically infinitely large forest,} \\ 0, & \text{else.} \end{cases} \quad (10)$$

Define $\varrho(\Theta, \Theta')$ to be the correlation (= covariance normalized by division with standard deviations) between rmg_Θ and $rmg_{\Theta'}$. For given Θ, Θ' , this is a number in $[-1, 1]$. Seen as a function of Θ, Θ' , $\varrho(\Theta, \Theta')$ maps each pair Θ, Θ' into $[-1, 1]$, which gives a RV which we denote with the same symbol ϱ for convenience,

$$\varrho(Z, Z') : \Omega^* \rightarrow [-1, 1],$$

where Z, Z' are two independent RVs with the same distribution as the Z_j .

Let

$$\bar{\varrho} = E[\varrho(Z, Z')]$$

be the expected value of $\varrho(Z, Z')$. It measures to what extent, in average across random choices for Θ, Θ' , the resulting two trees $h(\cdot, \Theta)$ and $h(\cdot, \Theta')$ have both the same correct or wrong decision averaged over data examples.

And here is, finally, Breiman's Theorem 2.3:

$$PE^* \leq \bar{\varrho}(1 - s^2)/s^2, \quad (11)$$

whose intuitive message I discussed earlier.

3 Elementary supervised temporal learning

In this section I give an introduction to a set of temporal data modeling methods which combine simplicity with broad practical usefulness: learning temporal tasks by training a linear regression map that transforms input signal windows to output data. In many scenarios, this simple technique is all one needs. It can be programmed and executed in a few minutes (really!) and you should run this technique as a first baseline whenever you start a serious learning task that involves time series data.

This section deals with *numerical* timeseries where the data format for each time point is a real-valued scalar or vector. This includes the majority of all learning tasks that arise in the natural sciences, in engineering, robotics, speech and in image processing.

Methods for dealing with *symbolic* timeseries (in particular texts, but also discrete action sequences of intelligent agents / robots, DNA sequences and more) can be obtained by encoding symbols into numerical vectors and then apply numerical methods. Often however one uses methods that operate on symbol sequence data structures directly (all kinds of discrete-state dynamical systems, deterministic or stochastic, like finite automata, Markov chains, hidden Markov models, dynamical Bayesian networks, and more). I will not consider such methods in this section.

My secret educational agenda in this section is that this will force you to rehearse linear regression - which is such a basic, simple yet widely useful method that everybody should have a totally 100% absolute unshakeable secure firm hold on it.

3.1 Recap: linear regression

Remark on notation: throughout these lecture notes, vectors are column vectors unless otherwise noted. I use $'$ to denote vector or matrix transpose.

I start with a generic rehearsal of linear regression, independent of temporal learning tasks.

The linear regression task is specified as follows:

Given: a collection $(\mathbf{x}_i, y_i)_{i=1,\dots,N}$ of N training data points, where $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$.

Wanted: a linear map from \mathbb{R}^n to \mathbb{R} , represented by a *regression weight vector* \mathbf{w} (a row vector of size n), which solves the minimization problem

$$\mathbf{w} = \underset{\mathbf{w}^*}{\operatorname{argmin}} \sum_{i=1}^N (\mathbf{w}^* \mathbf{x}_i - y_i)^2. \quad (12)$$

In machine learning terminology, this is a supervised learning task, because the training data points include “teacher” outputs y_i . Note that the abstract

data format $(\mathbf{x}_i, y_i)_{i=1,\dots,N}$ is the same as for decision trees, but here the data are numerical and there they were symbolic.

By a small modification, one can make linear regression much more versatile. Note that any weight vector \mathbf{w}^* in (12) will map the zero vector $\mathbf{0} \in \mathbb{R}^n$ on zero. This is often not what one wants to happen. If one enriches (12) by also training a *bias* $b \in \mathbb{R}$, via

$$(\mathbf{w}, b) = \underset{\mathbf{w}^*, b^*}{\operatorname{argmin}} \sum_{i=1}^N (\mathbf{w}^* \mathbf{x}_i + b^* - y_i)^2,$$

one obtains an *affine* linear function (linear plus offset). The common way to set up lineare regression such that affine linear solutions become possible is to pad the original input vectors \mathbf{x}_i with a last component of constant size 1, that is, in (12) on uses $n + 1$ -dimensional vectors $[\mathbf{x}; 1]$ (using Matlab notation for the vertical concatenation of vectors). Then a solution of (12) on the basis of the padded input vectors will be a regression weight vector $[\mathbf{w}, b] \in \mathbb{R}^{n+1}$, with the last component b giving the offset.

We now derive a solution formula for the minimization problem (12). Most textbooks start from the observation that the objective function $\sum_{i=1}^N (\mathbf{w} \mathbf{x}_i - y_i)^2$ is a quadratic function in the weights \mathbf{w} and then one uses calculus to find the minimum of this quadratic function by setting its partial derivatives to zero. I will present another derivation which does not need calculus and better reveals the underlying geometry of the problem.

Let $\mathbf{x}_i = (x_i^1, \dots, x_i^n)'$ be the i th input vector. The key to understand linear regression is to realize that the N values x_1^j, \dots, x_N^j of the j -the component in these vectors ($j = 1, \dots, n$) can be regarded as an N -dimensional vector $\varphi_j = (x_1^j, \dots, x_N^j)' \in \mathbb{R}^N$. Similarly, the N target values y_1, \dots, y_N can be combined into an N -dimensional vector y . Figure 10 **Top** shows a case where there are $N = 10$ input vectors of dimension $n = 4$.

Using these N -dimensional vectors as a point of departure, geometric insight gives us a nice clue how \mathbf{w} should be computed. To admit a visualization, we consider a case where we have only $N = 3$ input vectors which each have $n = 2$ components. This gives two $N = 3$ dimensional vectors φ_1, φ_2 (Figure 10 **Bottom**). The target values y_1, y_2, y_3 are combined in a 3-dimensional vector y .

Notice that in machine learning, one should best have more input vectors than the input vectors have components, that is, $N > m$. In fact, a very coarse rule of thumb – with many exceptions – says that one should aim at $N > 10n$ (if this is not warranted, use unsupervised dimension reduction methods to reduce n). We will thus assume that we have fewer vectors φ_j than training data points. The vectors φ_j thus span an n -dimensional subspace in \mathbb{R}^N (greenish shaded area in Figure 10 **Bottom**).

Notice (easy exercise, do it!) that the minimization problem (12) is equivalent

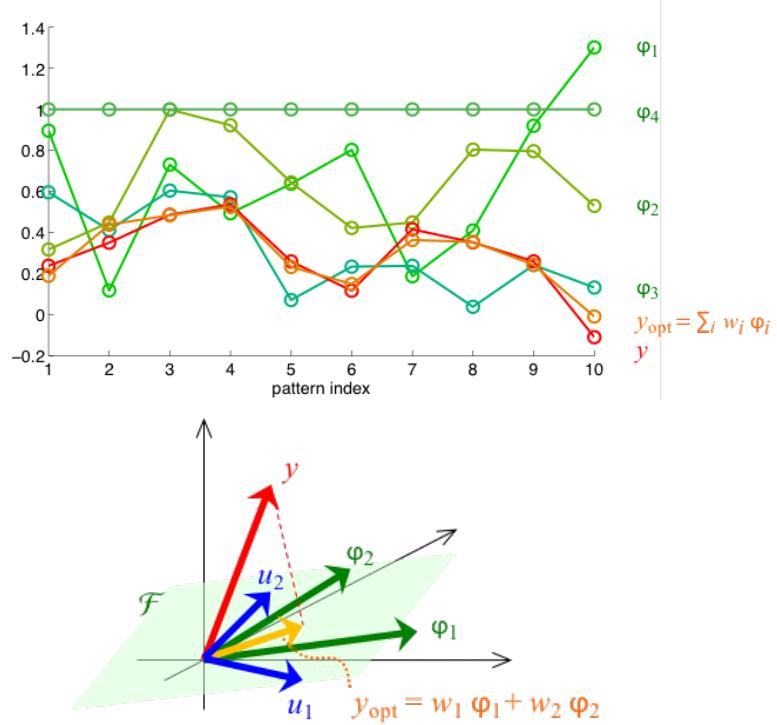


Figure 10: Two visualizations of linear regression. **Top.** This visualization shows a case where there are $N = 10$ input vectors \mathbf{x}_i , each one having $n = 4$ vector components x_i^1, \dots, x_i^4 (green circles). The fourth component is a constant-1 bias. The ten values x_1^j, \dots, x_{10}^j of the j -th component (where $j = 1, \dots, 4$) form a 10-dimensional (row) vector φ_j , indicated by a connecting line. Similarly, the ten target values y_i give a 10-dimensional vector y (shown in red). The linear combination $y_{\text{opt}} = \mathbf{w}[\varphi_1; \varphi_2; \varphi_3; \varphi_4]$ which gives the best approximation to y in the least mean square error sense is shown in orange. **Bottom.** The diagram shows a case where there the input vector dimension is $n = 2$ and there are $N = 3$ input vectors $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ in the training set. The three values x_1^1, x_2^1, x_3^1 of the first component give a three dimensional vector φ_1 , and the three values of the second component give φ_2 (green). These two vectors span a 2-dimensional subspace \mathcal{F} in $\mathbb{R}^N = \mathbb{R}^3$, shown in green shading. The three target values y_1, y_2, y_3 similarly make for a vector y (red). The linear combination $y_{\text{opt}} = w_1 \varphi_1 + w_2 \varphi_2$ which has the smallest distance to y is given by the projection of y on this plane \mathcal{F} (orange). The vectors u_1, u_2 shown in blue is a pair of orthonormal basis vectors which span the same subspace \mathcal{F} .

to

$$\mathbf{w} = \underset{\mathbf{w}^*}{\operatorname{argmin}} \left\| \left(\sum_{j=1}^n w_j^* \varphi_j \right) - y \right\|^2, \quad (13)$$

where $\mathbf{w}^* = (w_1^*, \dots, w_n^*)$. We take another look at Figure 10 **Bottom**. Equation 13 tells us that we have to find the linear combination $\sum_{j=1}^n w_j \varphi_j$ which comes closest to y . Any linear combination $\sum_{j=1}^n w_j^* \varphi_j$ is a vector which lies in the linear subspace \mathcal{F} spanned by $\varphi_1, \dots, \varphi_n$ (shaded area in the figure). The linear combination which is closest to y apparently is the projection of y on that subspace. This is the essence of linear regression!

All that remains is to compute which linear combination of $\varphi_1, \dots, \varphi_n$ is equal to the projection of y on \mathcal{F} . Let us call this projection y_{opt} . We may assume that the vectors $\varphi_1, \dots, \varphi_n$ are linearly independent. If they would be linearly dependent, we could drop as many from them as is needed to reach a linearly independent set.

The rest is mechanical linear algebra.

Let $X = (\varphi_1, \dots, \varphi_n)'$ be the $n \times N$ sized matrix whose rows are formed by the φ_j' (and whose columns are the \mathbf{x}_i). Then $X' X$ is a positive semi-definite matrix of size $N \times N$ with a singular value decomposition $X' X = U_N \Sigma_N U_N'$. Since the rank of X is $n < N$, only the first n singular values in Σ_N are nonzero. Let $U = (u_1, \dots, u_n)$ be the $N \times n$ matrix made from the first n columns in U_N , and let Σ be the $n \times n$ diagonal matrix containing the n nonzero singular values of Σ_N on its diagonal. Then

$$X' X = U \Sigma U'. \quad (14)$$

This is sometimes called the *compact SVD*. Notice that the columns u_j of U form an orthonormal basis of \mathcal{F} (blue arrows in Figure 10 **Bottom**).

Using the coordinate system given by u_1, \dots, u_n , we can rewrite each φ_j as

$$\varphi_j = \sum_{l=1}^n (u_l' \varphi_j) u_l = U U' \varphi_j. \quad (15)$$

Similarly, the projection y_{opt} of y on \mathcal{F} is

$$y_{\text{opt}} = \sum_{l=1}^n (u_l' y) u_l = U U' y, \quad (16)$$

But also $y_{\text{opt}} = X' \mathbf{w}'$. From (15) we get $X' = U U' X'$, which in combination with (16) turns $y_{\text{opt}} = X' \mathbf{w}'$ into

$$U U' y = U U' X' \mathbf{w}'.$$

A weight vector \mathbf{w} solves this equation if it solves

$$U' y = U' X' \mathbf{w}'. \quad (17)$$

It remains to find a weight vector \mathbf{w} which satisfies (17). I claim that $\mathbf{w}' = (XX')^{-1} X y$ does the trick, that is, $U' y = U' X' (XX')^{-1} X y$ holds.

To see this, first observe that XX' is nonsingular, thus $(XX')^{-1}$ is defined. Furthermore, observe that $U' y$ and $U' X' (XX')^{-1} X y$ are n -dimensional vectors, and that the $N \times n$ matrix $U\Sigma$ has rank n . Therefore,

$$U' y = U' X' (XX')^{-1} X y \iff U\Sigma U' y = U\Sigma U' X' (XX')^{-1} X y. \quad (18)$$

Replacing $U\Sigma U'$ by $X' X$ (compare Equation 14) turns the right equation in (18) into $X' X y = X' X X' (XX')^{-1} X y$, which is obviously true. Therefore, $\mathbf{w}' = (XX')^{-1} X y$ solves our optimization problem (13).

We summarize our findings:

Data. A set $(\mathbf{x}_i, y_i)_{i=1,\dots,N}$ of n -dimensional input vectors \mathbf{x}_i and scalar targets y_i .

Wanted. An n -dimensional row weight vector \mathbf{w} which solves the linear regression objective from Equation 12.

Step 1. Sort the input vectors as columns into an $n \times N$ matrix X and the targets into an N -dimensional vector y .

Step 2. Compute the (transpose of the) result by

$$\mathbf{w}' = (XX')^{-1} X y. \quad (19)$$

Some further remarks:

- What we have derived here generalizes easily to cases where the data are of the form $(\mathbf{x}_i, \mathbf{y}_i)_{i=1,\dots,N}$ where $\mathbf{x}_i \in \mathbb{R}^n, \mathbf{y}_i \in \mathbb{R}^k$. That is, the output data are vectors, not scalars. The objective is to find a $k \times n$ regression weight matrix W which solves

$$W = \underset{W^*}{\operatorname{argmin}} \sum_{i=1}^N \|W^* \mathbf{x}_i - \mathbf{y}_i\|^2. \quad (20)$$

The solution is given by

$$W' = (XX')^{-1} X Y,$$

where Y is the $N \times k$ matrix that contains the \mathbf{y}'_i in its rows.

- For an $a \times b$ sized matrix A , where $a \geq b$ and A has rank b , the matrix $A^+ := (A'A)^{-1} A'$ is called the (left) *pseudo-inverse* of A . It satisfies $A^+ A = I_{b \times b}$. It is often also written as A^\dagger .

- Computing the inversion $(XX')^{-1}$ may suffer from numerical instability when XX' is close to singular. Remark: this happens more often than you would think - in fact, XX' matrices obtained from real-world, high-dimensional data are *often* ill-conditioned (= close to singular). You should always feel uneasy when your program code contains a matrix inverse! A quick fix is to always add a small multiple of the $n \times n$ identity matrix before inverting, that is, replace (19) by

$$W'_{\text{opt}} = (XX' + \alpha^2 I_{n \times n})^{-1} X y. \quad (21)$$

This is called *ridge regression*. We will see later in this course that ridge regression not only helps to circumvent numerical issues, but also offers a solution to the problem of overfitting.

- A note on terminology. Here we have described *linear* regression. The word “regression” is used in much more general scenarios. The general setting goes like this:

Given: Training data $(\mathbf{x}_i, \mathbf{y}_i)_{i=1,\dots,N}$, where $\mathbf{x}_i \in \mathbb{R}^n, \mathbf{y}_i \in \mathbb{R}^k$.

Also given: a *search space* H containing candidate functions $h : \mathbb{R}^n \rightarrow \mathbb{R}^k$.

Also given: a loss function $L : \mathbb{R}^k \times \mathbb{R}^k \rightarrow \mathbb{R}^{\geq 0}$.

Wanted: A solution to the optimization problem

$$h_{\text{opt}} = \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^N L(h(\mathbf{x}_i), \mathbf{y}_i)$$

In the case of linear regression, the search space H consists of all linear functions from \mathbb{R}^n to \mathbb{R}^k , that is, it consists of all $k \times n$ matrices. The loss function is the *quadratic loss* which you see in (20). When one speaks of linear regression, the use of the quadratic loss is implied.

Search spaces H can be arbitrarily large and rich in modeling options – for instance, H might be the space of all deep neural networks of a given structure and size.

Classification tasks look similar to regression tasks at first sight: training data there have the format $(\mathbf{x}_i, c_i)_{i=1,\dots,N}$. The difference is that the target values c_i are not numerical but symbolic — they are class labels.

3.2 Temporal learning tasks

There are a number of standard learning tasks which are defined on the basis of timeseries data. Those tasks involve training data consisting of

an input signal $(\mathbf{u}(t))_{t \in \mathcal{T}}$, where \mathcal{T} is an *ordered* set of time points and for every $t \in \mathcal{T}, \mathbf{u}(t) \in \mathbb{R}^k$;

a “teacher” output signal $(\mathbf{y}(t))_{t \in \mathcal{T}}$, where $\mathbf{y}(t) \in \mathbb{R}^m$.

For simplicity we will only consider discrete time with equidistant unit timesteps, that is $\mathcal{T} = \mathbb{N}$ (unbounded time) or $\mathcal{T} = \{0, 1, \dots, T\}$ (finite time).

The learning task consists in training a system which operates in time and, if it is fed with the input signal $\mathbf{u}(t)$, produces an output signal $\hat{\mathbf{y}}(t)$ which approximates the teacher signal $\mathbf{y}(t)$ (in the sense of minimizing a loss $L(\hat{\mathbf{y}}(t) - \mathbf{y}(t))$ in time average over the training data). A few examples for illustration:

- Input signal: an ECG (electro cardiogram) signal; desired output: a signal which is constant zero as long as the ECG is normal, and jumps to 1 if there are irregularities in the ECG.
- Input signal: room temperature measurements from a thermometer. Output: value 0 as long as room temperature input signal is above certain threshold, value 1 if it is below the threshold (this is the input-output behavior of a thermostat, albeit a badly designed one - why?).
- Input signal: a noisy radio signal with lots of statics and echos. Desired output signal: the input signal in a version which has been de-noised and where echos have been cancelled.
- Input signal: a speech signal in English. Desired output: a speech signal in Dutch.

These are quite different sorts tasks. There are many names in use for quick references to temporal tasks. The ECG monitoring task would be called a *temporal classification* or *fault monitoring* task; the 0-1 switching of the badly designed thermostat is too simplistic to have a respectable name; radio engineers would speak of *de-noising* and *equalizing*, and speech translation is too uniquely complex to have a name besides “online speech translation”. Input-output signal transformation tasks are as many and as diverse as there are wonders under the sun.

In many cases, one may assume that the current output data point $\mathbf{y}(t)$ only depends on inputs up to that time point, that is, on the input history $\dots, \mathbf{u}(t-2), \mathbf{u}(t-1), \mathbf{u}(t)$. Specifically, $\mathbf{y}(t)$ does *not* depend on future inputs. Input-output systems where the output does not depend on future inputs are called *causal* systems in the signal processing world.

A causal system can be said to have *memory* if the current output $\mathbf{y}(t)$ is not fully determined by the current input $\mathbf{u}(t)$, but is influenced by earlier inputs as well. I must leave the meaning of “influenced by” vague at this point; we will make it precise in a later section when we investigate stochastic processes in more detail. All examples except the (poorly designed) thermostat example have memory.

Often, the output $\mathbf{y}(t)$ is influenced by long-ago input only to a negligible extent, and it can be explained very well from only the input history extending back

to a certain limited duration. All examples in the list above except the English translation one have such limited relevant memory spans. In causal systems with bounded memory span, the current output $y(t)$ thus depends on an input *window* $\mathbf{u}(t - d + 1), \mathbf{u}(t - d + 2), \dots, \mathbf{u}(t - 1), \mathbf{u}(t)$ of d steps duration. Figure 11 (top) gives an impression.

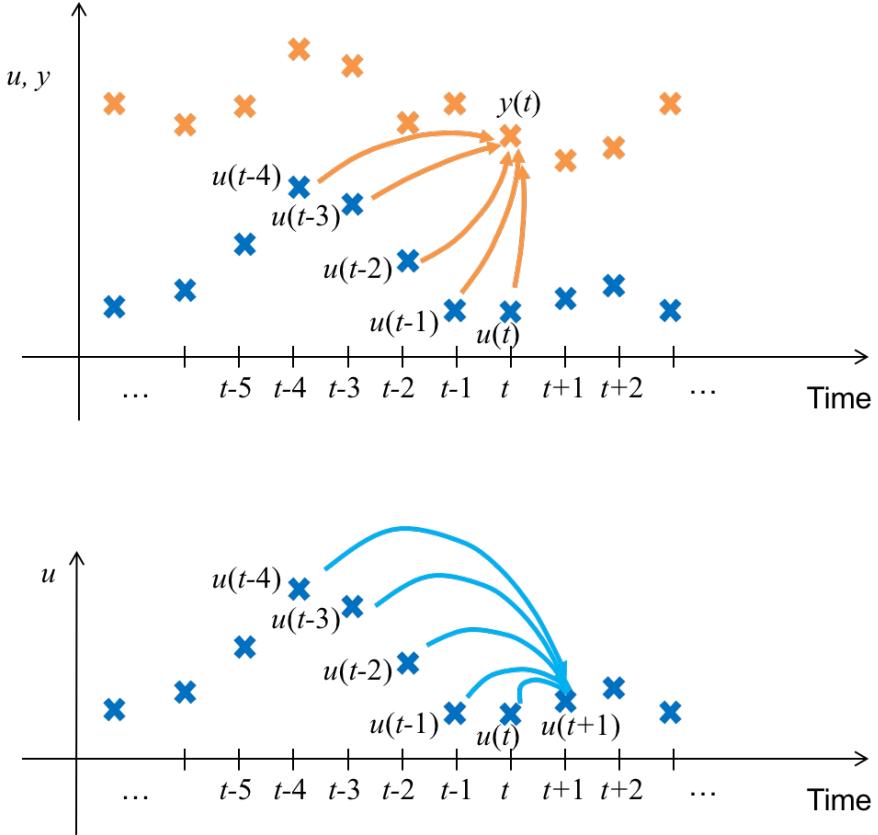


Figure 11: The principle of temporal system learning by window-based regression. The case of scalar input and output signals is shown. **Top:** an input signal $u(t)$ (blue crosses) is transformed to an output signal $y(t)$ (orange). **Bottom:** The special case of timeseries prediction (single step prediction). The window size in both diagrams is $d = 5$.

In this situation, learning an input-output system is a regression learning task. One has to find a regression function f of the kind

$$f : (\mathbb{R}^k)^d \rightarrow \mathbb{R}^m$$

which minimizes a chosen loss function on average over the training data time points.

The most convenient option is to go for the quadratic loss, which turns the fearsome input-output timeseries learning task into a case of linear regression.

Note that while in Section 3.1 I assumed that the input data points for linear regression were scalar, they are now k -dimensional vectors in general. This is not a problem; all one has to do is to flatten the collection of d k -dimensional input vectors which lie in a window into a single $d \cdot k$ dimensional vector, and then apply (20) as before.

Linear regression is often surprisingly accurate, especially when one uses large windows and a careful regularization (to be discussed later in this course) through ridge regression. When confronted with a new supervised temporal learning task, the first thing one should do as a seasoned pro is to run it through the machinery of window-based linear regression. This takes a few minutes programming and gives, at least, a baseline for more sophisticated methods.

But, linear regression only can give linear regression functions. This is not good enough if the dynamical input-output system behavior has significant nonlinear components. Then one must find a nonlinear regression function f .

If that occurs, one can take resort to a simple method which yields nonlinear regression functions while not renouncing the conveniences of the basic linear regression learning formula (19). I discuss this for the case of scalar inputs $u(t) \in \mathbb{R}$. The trick is to add fixed nonlinear transforms to the collection of input arguments $u(t-d+1), u(t-d+2), \dots, u(t)$. A common choice is to add polynomials. To make notation easier, let us rename $u(t-d+1), u(t-d+2), \dots, u(t)$ to u_1, u_2, \dots, u_d . If one adds all polynomials of degree 2, one obtains a collection of $d + d(d+1)/2$ input components for the regression, namely

$$\{u_1, u_2, \dots, u_d\} \cup \{u_i u_j \mid 1 \leq i \leq j \leq d\}.$$

If one wants “even more nonlinearity”, one can venture to add higher-order polynomials. The idea to approximate nonlinear regression functions by linear combinations of polynomial terms is a classical technique in signal processing, where it is treated under the name of *Volterra expansion* or *Volterra series*. Very general classes of nonlinear regression functions can be approximated to arbitrary degrees of precision with Volterra expansions.

Adding increasingly higher-order terms to a Volterra expansion obviously leads to a combinatorial explosion. Thus one will have to call upon some pruning scheme to use only those polynomial terms which lead to an increase of accuracy. There is a substantial literature in signal processing dealing with pruning strategies for Volterra series (google “Volterra pruning”). I personally would never try to use polynomials of degree higher than 2. If that doesn’t give satisfactory results, I would switch to other modeling techniques, using neural networks for instance.

3.3 Time series prediction tasks

Looking into the future can make you rich, or wise, or satisfy your curiosity – these motivations are deeply rooted in human nature and thus it is no wonder that time series prediction is an important kind of task for machine learning.

Time series prediction tasks come in many variants and I will not attempt to draw the large picture but restrict this treatment to timeseries with integer timesteps and vector-valued observations, as in the previous subsections.

Re-using terminology and notation, the input signal in the training data is, as before, $(\mathbf{u}(t))_{t \in \mathcal{T}}$. If one wants to predict this sequence of observations h timesteps ahead (h is called *prediction horizon*), the desired output $\mathbf{y}(t)$ is just the input, shifted by h timesteps:

$$(\mathbf{y}(t))_{t \in \mathcal{T}} = (\mathbf{u}(t + h))_{t \in \mathcal{T}}.$$

A little technical glitch is that due to the timeshift h , the last h data points in the input signal $(\mathbf{u}(t))_{t \in \mathcal{T}}$ cannot be used as training data because their h -step future would lie beyond the maximal time T .

Framed as an $\mathbf{u}(t)$ -to- $\mathbf{y}(t)$ input-output signal transformation task, all methods that can be applied for the latter can be used for timeseries prediction too. Specifically, simple window-based linear regression (as in Figure 11 bottom) is again a highly recommendable first choice for getting a baseline predictor whenever you face a new timeseries prediction problem with numerical timeseries.

3.4 State-based vs. signal-based timeseries modeling

So far, I have described some ways how one can use input observations to compute output signals. If one starts thinking about it, it seems that in some cases this should be impossible.

Many temporal learning tasks concern the input-output modeling of a physical *dynamical system*. The output signal is not (somehow magically) directly transformed from the input signal, but by a complex transformation which resides in the internal mechanisms of the physical system that sits “in the middle” between input and output signals. Just two examples:

- A radio reporter commenting a soccer game. Output: the reporter’s speech signal. Input: what the reporter sees on the soccer field. Dynamical system in the middle: the reporter, especially his brain and vocal tract. – Two different reporters would comment the game differently, even if they see it from the same reporters’ stand. Much of the specific information in the reporter’s radio speech is not explainable by the raw soccer visual input, but arises from the reporter’s specific knowledge and emotional condition. – Fun fact: attempting to developing machine learning methods for the automated commenting of soccer matches had been, for a decade or so around 1985, one of the leading research motifs in Germany’s young AI research landscape.
- Input signal: the radiowaves emitted from a transmitting antenna. Output signal: the radiowaves picked up by a receiving antenna. Physical system “in the middle”: the physical world between the two antennas, inducing all kinds of static noise, echos, distortions, such that the received radio signal is a highly corrupted version of the cleanly transmitted one. Does it not

seem hopeless to model the input-output transformation without including a model of the intervening physical *channel*? – There can hardly be a more classical problem than this one; analysing signal transmission channels gave rise to Shannon’s information theory.

Abstracting from these examples, consider how natural scientists and mathematicians describe dynamical systems. We stay in line with earlier parts of this section and consider only discrete-time models with a time index set is $T = \mathbb{N}$ or $T = \mathbb{Z}$. Three timeseries are considered simultaneously:

- an *input signal* $\mathbf{u}(t)$, where $\mathbf{u}(t) \in \mathbb{R}^k$ (as before),
- an *output signal* $\mathbf{y}(t)$, where $\mathbf{y}(t) \in \mathbb{R}^m$ (as before),
- a *dynamical system state* sequence $\mathbf{x}(t)$, where $\mathbf{x}(t) \in \mathbb{R}^n$ (this is new).

In soccer reporting example, $\mathbf{x}(t)$ would refer to some kind of brain state — for instance, the vector of activations of all the reporter’s brain’s neurons. In the signal transmission example, $\mathbf{x}(t)$ would be the state vector of some model of the physical world stretched out between the sending and receiving antenna.

A note in passing: variable naming conventions are a little confusing. In the machine learning literature, x ’s and y ’s in (\mathbf{x}, \mathbf{y}) pairs usually mean the arguments and values (or inputs and outputs) of classification/regression functions. Both \mathbf{x} and \mathbf{y} are observable. In the dynamical systems literature (mathematics, physics and engineering), the variable name x typically is reserved for the state of a signal-generating dynamical system – this state is normally not fully observable and not part of training data. Only the input and output signals $\mathbf{u}(t), \mathbf{y}(t)$ are observable “data”.

In a discrete-time setting, the temporal evolution of $\mathbf{u}(t), \mathbf{x}(t), \mathbf{y}(t)$ is governed by two functions, the *state update map*

$$\mathbf{x}(t+1) = f(\mathbf{x}(t), \mathbf{u}(t+1)) \tag{22}$$

and the *observation function*

$$\mathbf{y}(t) = g(\mathbf{x}(t)). \tag{23}$$

The input signal $\mathbf{u}(t)$ is not specified by some equation, it is just “given”.

Figure 12 visualizes the structural difference between the signal-based and the state-based input-output transformation models.

There are many other types of state update maps and observation functions, for instance ODEs and PDEs for continuous-time systems, automata models for discrete-state systems, or a host of probabilistic formalisms for random dynamical systems. For our present discussion, considering only discrete-time state update maps is good enough.

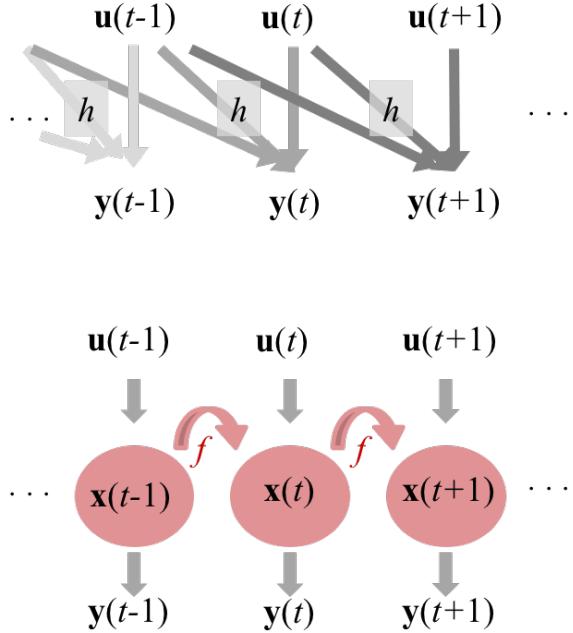


Figure 12: Contrasting signal-based with state-based timeseries transformations. **Top:** Window-based determination of output signal through a regression function h (here the window size is 3). **Bottom:** Output signal generation by driving in intermediary dynamical system with an input signal. The state $\mathbf{x}(t)$ is updated by f .

A core difference between signal-based and state-based timeseries transformations is the achievable memory timespans. In windowed signal transformations through regression functions, the memory depth is bounded by the window length. In contrast, the dynamical system state $\mathbf{x}(t)$ of the intermediary system is potentially co-determined by input that was fed to the dynamical system in an arbitrary deep past – the memory span can be unbounded! This may seem counterintuitive if one looks at Figure 12 because at each time point t , only the input data point $\mathbf{u}(t)$ from that same timepoint is fed to the dynamical system. But $\mathbf{u}(t)$ leaves some trace on the state $\mathbf{x}(t)$, and this effect is forwarded to the next timestep through f , thus $\mathbf{x}(t+1)$ is affected by $\mathbf{u}(t)$, too; and so forth. Thus, if one expects long-range or even unbounded memory effects, using state-based transformation models is often the best way to go.

Machine learning offers a variety of state-based models for timeseries transformations, together with learning algorithms. The most powerful ones are hidden Markov models (which we'll get to know in this course) and other dynamical graphical models (which we will not touch), and recurrent neural networks (which we'll briefly meet I hope).

In some applications it is important that the input-output transformation can be learnt in an *online adaptive* fashion. The input-output transformation is not trained just once, on the basis on a given, fixed training dataset. Instead, training never ends; while the system is being used, it continues to adapt to *changing* input-output relationships in the data that it processes. My favourite example is the online adaptive filtering (denoising, echo cancellation) of the radio signal received by a mobile phone. When the phone is physically moving while it is being used (phonecall in a train, or just while walking up and down in a room), the signal channel from the transmitter antenna to the phone's antenna keeps changing. The denoising, echo-cancelling filter is re-learnt every few milliseconds. This is done with a window-based linear regression (window size several tens of thousands) and ingeniously simplified/accelerated algorithms. Because this is powerful stuff, we machine learners should not leave that only to the electrical engineers (who invented it) but learn to use it ourselves. I will devote a session later in this course to these online adaptive signal processing methods.

3.5 Takens' theorem

From what I just explained it may seem that signal-based and state-based input-output transformations are two quite different things. However, surprising connections exist. In certain ways and for certain systems, the two even coincide. This was discovered by the Dutch mathematician Floris Takens (1940-2010). His celebrated theorem (Takens, 1991), now universally called *Takens Theorem*, is deep and beautiful, and enabled novel data analysis methods which triggered a flood of work in the study of complex system dynamics (in all fields biological, physical, metereological, economical, social, engineering). Also, Takens was a professor in Groningen, and founded a tradition of dynamical systems theory research in our mathematics department. Plus, finally, Takens-like theorems have very recently (not yet published) been used to analyse why/how certain recurrent neural networks of the “reservoir computing” brand function so surprisingly well in machine learning. These are all good enough causes for me to end this section with an intuitive explanation of Takens theorem, although this is not normally a part of a machine learning course.

Since the following contains allusions to mathematical concepts which I cannot assume are known by all course participants, the material in this section will not be tested in exams.

Floris Takens' original theorem was formulated in a context of continuous-time dynamics governed by differential equations. Many variants and extensions of Takens theorem are known. To stay in tune with earlier parts of this section I present a discrete-time version. Consider the input-free dynamical system

$$\begin{aligned}\mathbf{x}(t+1) &= f(\mathbf{x}(t)), \\ y(t) &= g(\mathbf{x}(t)),\end{aligned}\tag{24}$$

where $\mathbf{x}(t) \in \mathbb{R}^n$ and $y(t) \in \mathbb{R}$ (a one-dimensional output). The background intuition behind this set-up is that \mathbf{x} models some physical dynamical system — possibly quite high-dimensional, for instance \mathbf{x} being a brain state vector — while y is a physical quantity that is “measured”, or “observed”, from \mathbf{x} .

If the state update dynamics (24) is run for a long time, in many systems the state vector sequence $\mathbf{x}(t)$ will ultimately become confined to an *attracting* subset $A \subset \mathbb{R}^n$ of the embedding space \mathbb{R}^n . In some cases, this subset will be a low-dimensional manifold of dimension m ; in other cases (“chaotic attractors”) it will have a fractal geometry which is characterized by a *fractal dimension*, a real number which I will likewise call m and which typically is also small compared to the embedding space dimension n .

Now let us look at the observation sequence $y(t)$ recorded when the system has “relaxed” to the attracting subset. It is a scalar timeseries. For any $k > 1$, this scalar timeseries can be turned into a k -dimensional timeseries $\mathbf{y}(t)$ by *delay embedding*, as follows. Choose a delay of δ timesteps. Then set $\mathbf{y}(t) = (y(t), y(t - \delta), y(t - 2\delta), \dots, y(t - (k-1)\delta))'$. That is, just stack a few consecutive observations (spaced in time by δ) on top of each other. This gives a sequence of k -dimensional vectors $\mathbf{y}(t)$. You will recognize them as what we called “observation windows” earlier in this section.

Takens-like theorems now state that, under conditions specific to the particular version of the theorem, the geometry of the dynamics of the delay-embedded vectors is the same as the geometry of the dynamics of the original system, up to a smooth bijective transformation between the two geometries. I think a picture is really helpful here! Figure 13 gives a visual demo of Takens theorem at work.

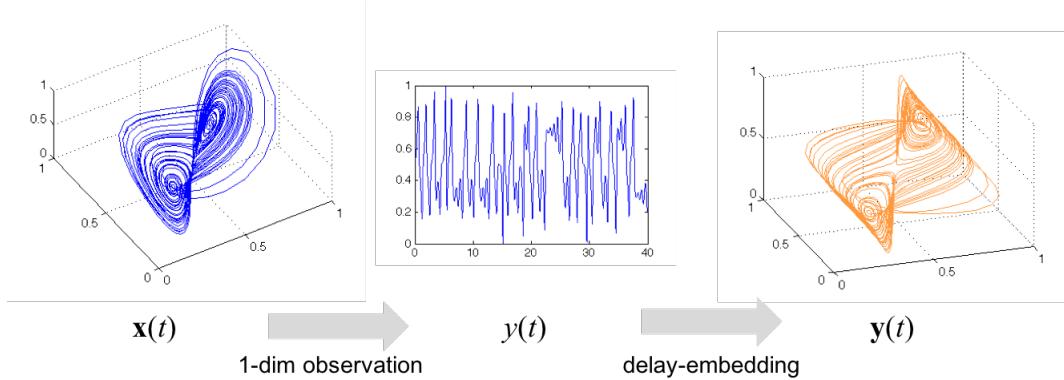


Figure 13: Takens theorem visualized. Left plot shows the Lorenz attractor, a chaotic attractor with a state sequence $\mathbf{x}(t)$ defined in an $n = 3$ dimensional state space. The center plot shows a 1-dimensional observation $y(t)$ thereof. The right plot (orange) shows the state sequence $\mathbf{y}(t)$ obtained from $y(t)$ by three-dimensional delay embedding (I forgot which delay δ I used).

When I just said, “the geometry ... is the same ... up to smooth bijective transformation”, I mean the following. Imagine that in Figure 13 (left) the blue trajectory lines were spun into a transparent 3-dimensional substrate which has rubber-like qualities. Then, by pushing and pulling and shearing this substrate (without rupturing it), the blue lines would take on new positions in 3D space — until they would *exactly* coincide with the orange ones in the right panel.

Even without applying such a “rubber-sheet transformation” (a term used in dynamical systems textbooks), some geometric characteristics are identical between the left and the right panel in the figure. Specifically, the so-called *Lyapunov exponents*, which are indicators for the stability (noise robustness) of the dynamics, and the (possibly fractal) dimension of the attracting set A are identical in the original dynamics and its delay-embedded reconstruction.

The deep, nontrivial message behind Takens-like theorems is that one can trade time for space in certain dynamical systems. If by “space” one means dimensions of state vectors, and by “time” lengths of observation windows of a scalar observable, Takens-like theorems typically state that if the dimension of the original attractor set (manifold or fractal) is m , then using a delay embedding of an integer dimension larger or equal than $2m + 1$ will always recuperate the original geometry; and sometimes smaller delay embedding dimensions suffice. Just for completeness: the fractal dimension of the Lorenz attractor is about $m = 2.06$, thus any delay embedding dimension exceeding $2m + 1 = 5.12$ would be certainly sufficient for reconstruction from a scalar observation; it turns out in Figure 13 that an embedding dimension of 3 is already working.

In timeseries prediction tasks in machine learning, Takens theorem gives a justification why in deterministic, autonomous dynamical systems it should be possible to predict the future of a scalar timeseries $y(t)$ from its past. If the timeseries in question has been produced from a system like (24), Takens theorem establishes that the information contained in the last $2m + 1$ timesteps before the prediction point t comprises the full information about the state of the system that generated $y(t)$ and thus knowledge of the last $2m + 1$ timesteps is enough to predict $y(t + 1)$ with absolute accuracy.

Even if you forget about the Takens theory background, it is good to be familiar with delay-embeddings of scalar timeseries, because they help creating instructive graphics. If you transform a one-dimensional scalar timeseries $y(t)$ into a 2-dimensional one by a 2-element delay embedding, plotting this 2-dimensional trajectory $\mathbf{y}(t) = (y(t - \delta), y(t))'$ in its delay coordinates $(y(t - \delta), y(t))$ will often give revealing insight into the structure of the timeseries — our visual system is optimized for seeing patterns in 2-D images, not in 1-dimensional timeseries plots. See Figure 14 for a demo.

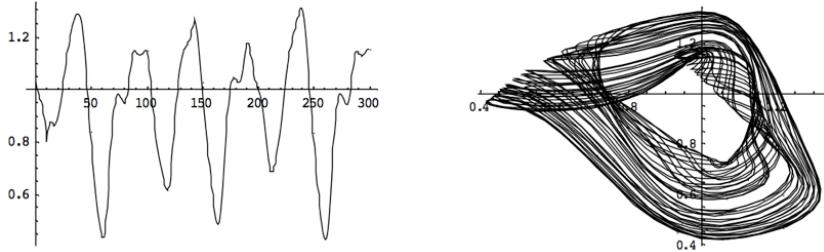


Figure 14: Getting nice graphics from delay embeddings. Left: a timeseries recorded from the “Mackey-Glass” chaotic attractor. Right: plotting the trajectory of a delay-embedded version of the left signal.

4 Basic methods for dimension reduction

One way to take up the fight with the “curse of dimensionality” which I briefly highlighted in Section 1.2.2 is to reduce the dimensionality of the raw input data before they are fed to subsequent learning algorithms. The dimension reduction ratio can be enormous.

In this Section I will introduce three standard methods for dimension reduction: K-means clustering, principal component analysis, and self-organizing feature maps. They all operate on raw vector data which come in the form of points $\mathbf{x} \in \mathbb{R}^n$, that is, this section is only about dimension reduction methods for numerical data. Dimension reduction is the archetypical unsupervised learning task.

4.1 Set-up, terminology, general remarks

We are given a family $(\mathbf{x}_i)_{i=1,\dots,N}$ of “raw” data points, where $\mathbf{x}_i \in \mathbb{R}^n$. The goal of dimension reduction methods is to compute from each (high-dimensional) “raw” data vector \mathbf{x}_i a (low-dimensional) vector $\mathbf{f}(\mathbf{x}_i) \in \mathbb{R}^m$, such that

- $m < n$, that is, we indeed reduce the number of dimensions — maybe even dramatically;
- the low-dimensional vectors $\mathbf{f}(\mathbf{x}_i)$ should preserve from the raw data the specific information that is needed to solve the learning task that comes after this dimension-reducing data “preprocessing”.

Terminology: The m component functions which constitute $\mathbf{f}(\mathbf{x})$ are called *features*. A feature is just a (any) function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ which computes a scalar characteristic of input vectors $\mathbf{x} \in \mathbb{R}^n$. If one bundles together m such features f_1, \dots, f_m one obtains a *feature map* $(f_1, \dots, f_m)' =: \mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ which maps input vectors to *feature vectors*.

It is very typical, almost universal, for ML systems to include an initial data processing stage where raw, high-dimensional input patterns are first projected

from their original pattern space \mathbb{R}^n to a lower-dimensional feature space. In TICS, for example, a neural network was trained in a clever way to reduce the 1,440,000-dimensional raw input patterns to a 4,096-dimensional feature vector.

The ultimate quality of the learning system clearly depends on a good choice of features. Unfortunately there does not exist a unique or universal method to identify “good” features. Depending on the learning task and the nature of the data, different kinds of features work best. Accordingly, ML research has come up with a rich repertoire of *feature extraction* methods.

On an intuitive level, a “good” set of features $\{f_1, \dots, f_m\}$ should satisfy some natural conditions:

- The number m of features should be small — after all, one of the reasons for using features is dimension reduction.
- Each feature f_i should be *relevant* for the task at hand. For example, when the task is to distinguish helicopter images from winged aircraft photos (a 2-class classification task), the brightness of the background sky would be an irrelevant feature; but the binary feature “has wings” would be extremely relevant.
- There should be *little redundancy* — each used feature should contribute some information that the others don’t.
- A general intuition about features is that they should be rather cheap and easy to compute at the front end where the ML system meets the raw data. The “has wings” feature for helicopter vs. winged aircraft classification more or less amounts to actually solving the classification task and presumably is neither cheap nor easy to compute. Such highly informative, complex features are sometimes called *high-level features*; they are usually computed on the basis of more elementary, *low-level* features. Often features are computed stage-wise, low-level features first (directly from data), then stage by stage more complex, more directly task-solving, more “high-level cognition” features are built by combining the lower-level ones. *Feature hierarchies* are often found in ML systems. Example: in face recognition from photos, low-level features might extract coordinates of isolated black dots from the photo (candidates for the pupils of the person’s eyes); intermediate features might give distance ratios between eyes, nose-tip, center-of-mouth; high-level features might indicate gender or age.

To sharpen our intuitions about features, let us hand-design some features for use in a simple image classification task. Consider a training dataset which consists of grayscale, low-resolution pictures of handwritten digits (Figure 15). Here and further in this section I will illustrate dimension reduction techniques with the simple “Digits” benchmark dataset which was first described in Kittler et al. (1998). This dataset contains 2000 grayscale images of handwritten digits,

200 from each class. The images are 15×16 sized, making for $n = 240$ dimensional image vectors \mathbf{x} . We assume they have been normalized to a real-valued pixel value range in $[0, 1]$ with 0 = white and 1 = black. The Digits benchmark task is to train a classifier which classifies such images into the ten digit classes.

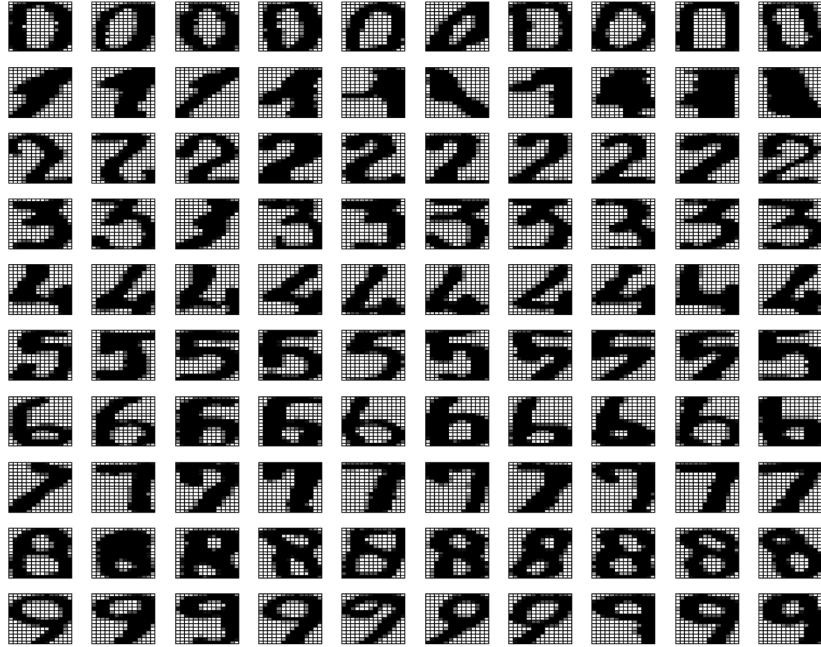


Figure 15: Some examples from the Digits dataset.

Here are some candidates for features that might be useful for this task.

Mean brightness. $f_1(\mathbf{x}) = \mathbf{1}'_n \mathbf{x} / n$ ($\mathbf{1}_n$ is the vector of n ones). This is just the mean brightness of all pixels. Might be useful e.g. for distinguishing “1” images from “8” images because we might suspect that for drawing an “8” one needs more black ink than for drawing a “1”. Cheap to compute but not very class-differentiating.

Radiality. An image \mathbf{x} is assigned a value $f_2(\mathbf{x}) = 1$ if and only if two conditions are met: (i) the center horizontal pixel line crossing the image from left to right has a sequence of pixels that changes from black to white to black; (ii) same for the center vertical pixel line. If this double condition is not met, the image is assigned a feature value of $f_2(\mathbf{x}) = 0$. f_2 thus has only two possible values; it is called a *binary feature*. We might suspect that only the “0” images have this property. This would be a slightly less cheap-to-compute feature compared to f_1 but more informative about classes.

Prototype matching. For each of the 10 classes c_j ($j = 1, \dots, 10$) define a *prototype vector* π_j as the mean image vector of all examples of that class contained in the training dataset: $\pi_j = 1/N_j \sum_{\mathbf{x} \text{ is training image of class } j} \mathbf{x}$.

Then define 10 features f_3^j by match with these prototype vectors: $f_3^j(\mathbf{x}) = \pi_j' \mathbf{x}$. We might hope that f_3^j has a high value for patterns of class j and low values for other patterns.

Hand-designing features can be quite effective. Generally speaking, human insight on the side of the data engineer is a success factor for ML systems that can hardly be overrated. In fact, the classical ML approach to speech recognition was for two decades relying on low-level acoustic features that had been hand-designed by insightful phonologists. The MP3 sound coding format is based on features that reflect characteristics of the human auditory system. Many of the first functional computer vision and optical character recognition systems relied heavily on visual feature hierarchies which grew from the joint efforts of signal processing engineers and cognitive neuroscience experts.

However, since hand-designing good features means good insight on the side of the engineer, and good engineers are rare and have little time, the practice of ML today relies much more on features that are obtained from learning algorithms. Numerous methods exist. In the following subsections we will inspect three such methods.

4.2 K-means clustering

K-means clustering is an algorithm which allows one to split a collection of training data points $(\mathbf{x}_i)_{i=1,\dots,N} \in \mathbb{R}^n$ into K clusters C_1, \dots, C_K , such that points from the same cluster lie close to each other while being further away from points in other clusters. Each cluster C_j is represented by its *codebook* vector c_j , which is the vector pointing to the mean of all vectors in the cluster — that is, to the cluster's center of gravity.

The codebook vectors can be used in various ways to compress n -dimensional test data points \mathbf{x}^{test} into lower-dimensional formats. The classical method, which also explains the naming of the c_j as “codebook” vectors, is to represent \mathbf{x}^{test} simply by the index j of the codebook vector c_j which lies closest to \mathbf{x}^{test} , that is, which has the minimal distance $\alpha_j = \|\mathbf{x}^{\text{test}} - c_j\|$. The high-dimensional point \mathbf{x}^{test} becomes represented by a single number, thus here we would have $m = 1$. This method is clearly very economical and it is widely used for data compression. But it is also clear that relevant information contained in vectors \mathbf{x}^{test} may be lost — maybe too much information for many ML applications.

Another, less lossy, way to make use of the codebook vectors for dimension reduction is to compute the distances α_j between \mathbf{x}^{test} and the codebook vectors, and reduce \mathbf{x}^{test} to the K -dimensional distance vector $(\alpha_1, \dots, \alpha_K)'$. When $K \ll n$, the dimension reduction is substantial. The vector $(\alpha_1, \dots, \alpha_K)'$ can be considered a feature vector.

We can be brief when explaining the K -means clustering algorithm, because it is almost self-explaining. The rationale for defining clusters is that points within a

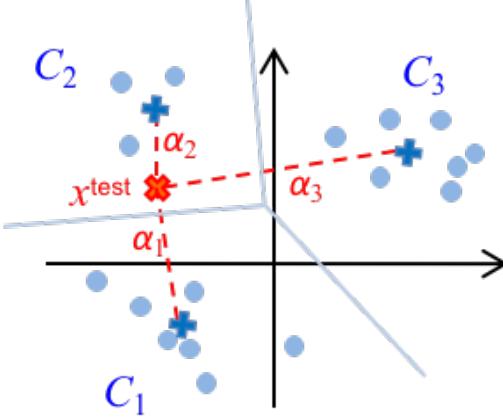


Figure 16: Clusters obtained from K-means clustering (schematic): For a training set of data points (light blue dots), a spatial grouping into clusters C_j is determined by the K-means algorithm. Each cluster becomes represented by a codebook vector (dark blue crosses). The figure shows three clusters. The light blue straight lines mark the cluster boundaries. A test data point \mathbf{x}^{test} (red cross) may then become coded in terms of the distances α_j of that point to the codebook vectors. Since this \mathbf{x}^{test} falls into the second cluster C_2 , it could also be compressed into the codebook index “2” of this cluster.

cluster should have small metric distance to each other, points in different clusters should have large distance from each other. The procedure runs like this:

Given: a training data set $(\mathbf{x}_i)_{i=1,\dots,N} \in \mathbb{R}^n$, and a number K of clusters that one maximally wishes to obtain.

Initialization: randomly assign the training points to K sets S_j ($j = 1, \dots, K$).

Repeat: For each set S_j , compute the mean $\mu_j = |S_j|^{-1} \sum_{\mathbf{x} \in S_j} \mathbf{x}$. This mean vector μ_j is the “center of gravity” of the vector cluster S_j . Create new sets S'_j by putting each data point \mathbf{x}_i into that set S'_j where $\|\mathbf{x}_i - \mu_j\|$ is minimal. If some S'_j remains empty, dismiss it and reduce K to K' by subtracting the number of dismissed empty sets (this happens rarely). Put $S_j = S'_j$ (for the nonempty sets) and $K = K'$.

Termination: Stop when in one iteration the sets remain unchanged.

It can be shown that at each iteration, the error quantity

$$J = \sum_{j=1}^K \sum_{\mathbf{x} \in S_j} \|\mathbf{x} - \mu_j\|^2 \quad (25)$$

will not increase. The algorithm typically converges quickly and works well in

practice. It finds a local minimum or saddle point of J . The final clusters S_j may depend on the random initialization. The clusters are bounded by straight-line boundaries; each cluster forms a *Voronoi cell*. K -means cannot find clusters defined by curved boundaries. Figure 17 shows an example of a clustering run using K -means.

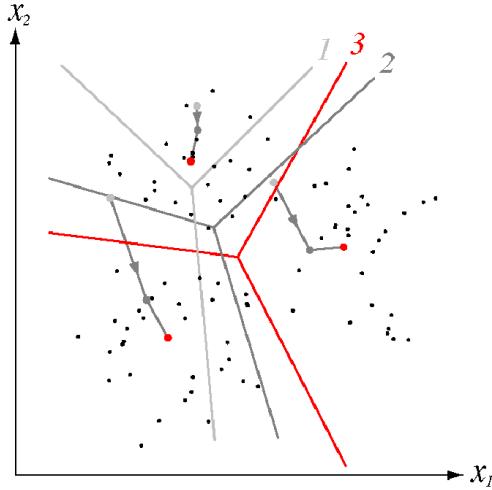


Figure 17: Running K -means with $K = 3$ on two-dimensional training points. Thick dots mark cluster means μ_j , lines mark cluster boundaries. The algorithm terminates after three iterations, whose boundaries are shown in light gray, dark gray, red. (Picture taken from Chapter 10 of the textbook Duda et al. (2001)).

K -means clustering and other clustering methods have many uses besides dimension reduction. Clustering can also be seen as a stand-alone technique of unsupervised learning. The detected clusters and their corresponding codebook vectors are of interest in their own right. They reveal a basic structuring of a set of patterns $\{\mathbf{x}_i\}$ into subsets of mutually similar patterns. These clusters may be further analyzed individually, given meaningful names and helping a human data analyst to make useful sense of the original unstructured data cloud. For instance, when the patterns $\{\mathbf{x}_i\}$ are customer profiles, finding a good grouping into subgroups may help to design targetted marketing strategies.

4.3 Principal component analysis

Like clustering, principal component analysis (PCA) is a basic data analysis technique that has many uses besides dimension reduction. But here we will focus on this use.

Generally speaking, a good dimension reduction method, that is, a good feature map $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, should preserve much information contained in the high-dimensional patterns $\mathbf{x} \in \mathbb{R}^n$ and encode it robustly in the feature vectors $y = \mathbf{f}(\mathbf{x})$. But, what does it mean to “preserve information”? A clever answer is this: a feature representation $\mathbf{f}(\mathbf{x})$ preserves information about \mathbf{x} to the extent that \mathbf{x} can be *reconstructed* from $\mathbf{f}(\mathbf{x})$. That is, we wish to have a *decoding function* $\mathbf{d} : \mathbb{R}^m \rightarrow \mathbb{R}^n$ which leads back from the feature vector encoding $\mathbf{f}(\mathbf{x})$ to \mathbf{x} , that is we wish to achieve

$$\mathbf{x} \approx \mathbf{d} \circ \mathbf{f}(\mathbf{x})$$

And here comes a fact of empowerment: when \mathbf{f} and \mathbf{d} are confined to *linear* functions and when the similarity $\mathbf{x} \approx \mathbf{d} \circ \mathbf{f}(\mathbf{x})$ is measured by mean square error, the optimal solution for \mathbf{f} and \mathbf{d} can be easily and cheaply computed by a method that is known since the early days of statistics, *principal component analysis* (PCA). It was first found, in 1901, by Karl Pearson, one of the fathers of modern mathematical statistics. The same idea has been independently re-discovered under many other names in other fields and for a variety of purposes (check out https://en.wikipedia.org/wiki/Principal_component_analysis for the history). Because of its simplicity, analytical transparency, modest computational cost, and numerical robustness PCA is widely used — it is the first-choice default method for dimension reduction that is tried almost by reflex, before more elaborate methods are maybe considered.

PCA is best explained alongside with a visualization (Figure 18). Assume the patterns are 3-dimensional vectors, and assume we are given a sample of $N = 200$ raw patterns $\mathbf{x}_1, \dots, \mathbf{x}_{200}$. We will go through the steps of a PCA to reduce the dimension from $n = 3$ to $m = 2$.

The first step in PCA is to *center* the training patterns \mathbf{x}_i , that is, subtract their mean $\mu = 1/N \sum_i \mathbf{x}_i$ from each pattern, obtaining centered patterns $\bar{\mathbf{x}}_i = \mathbf{x}_i - \mu$. The centered patterns form a point cloud in \mathbb{R}^n whose center of gravity is the origin (see Figure 18A).

This point cloud will usually not be perfectly spherically shaped, but instead extend in some directions more than in others. “Directions” in \mathbb{R}^n are characterized by unit-norm “direction” vectors $u \in \mathbb{R}^n$. The distance of a point $\bar{\mathbf{x}}_i$ from the origin *in the direction of* u is given by the projection of $\bar{\mathbf{x}}_i$ on u , that is, the inner product $u' \bar{\mathbf{x}}_i$ (see Figure 19).

The “extension” of a centered point cloud $\{\bar{\mathbf{x}}_i\}$ in a direction u is defined to be the mean squared distance to the origin of the points $\bar{\mathbf{x}}_i$ in the direction of u . The direction of the largest extension of the point cloud is hence the direction vector given by

$$u_1 = \underset{u, \|u\|=1}{\operatorname{argmax}} 1/N \sum_i (u' \bar{\mathbf{x}}_i)^2. \quad (26)$$

Notice that since the cloud $\bar{\mathbf{x}}_i$ is centered, the mean of all $u' \bar{\mathbf{x}}_i$ is zero, and hence the number $1/N \sum_i (u' \bar{\mathbf{x}}_i)^2$ is the variance of the numbers $u' \bar{\mathbf{x}}_i$.

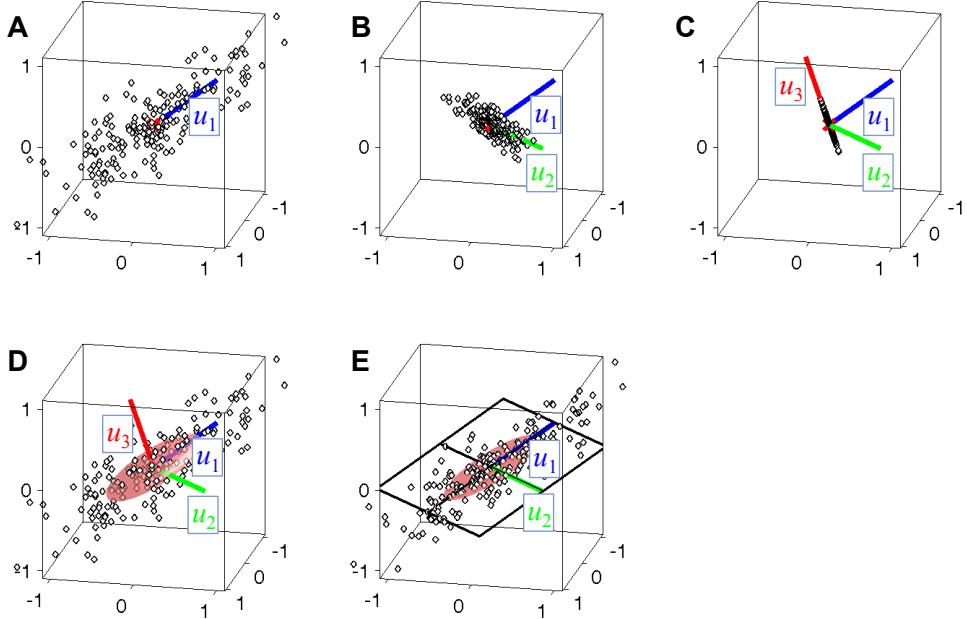


Figure 18: Visualization of PCA. **A.** Centered data points and the first principal component vector u_1 (blue). The origin of \mathbb{R}^3 is marked by a red cross. **B.** Projecting all points to the orthogonal subspace of u_1 and computing the second PC u_2 (green). **C.** Situation after all three PCs have been determined. **D.** Summary visualization: the original data cloud with the three PCs and an ellipsoid aligned with the PCs whose main axes are scaled to the standard deviations of the data points in the respective axis direction. **E.** A new dimension-reduced coordinate system obtained by the projection of data on the subspace U_m spanned by the m first PCs (here: the first two).

Inspecting Figure 18A, one sees how u_1 points in the “longest” direction of the pattern cloud. The vector u_1 is called the first *principal component* (PC) of the centered point cloud.

Next step: project patterns on the $(n - 1)$ -dimensional linear subspace of \mathbb{R}^n that is orthogonal to u_1 (Figure 18B). That is, map pattern points $\bar{\mathbf{x}}$ to $\bar{\mathbf{x}}^* = \bar{\mathbf{x}} - (u'_1 \bar{\mathbf{x}}) \cdot u_1$. Within this “flattened” pattern cloud, again find the direction vector of greatest variance

$$u_2 = \underset{u, \|u\|=1}{\operatorname{argmax}} \frac{1}{N} \sum_i (u' \bar{\mathbf{x}}_i^*)^2$$

and call it the second PC of the centered pattern sample. From this procedure it is clear that u_1 and u_2 are orthogonal, because u_2 lies in the orthogonal subspace of u_1 .

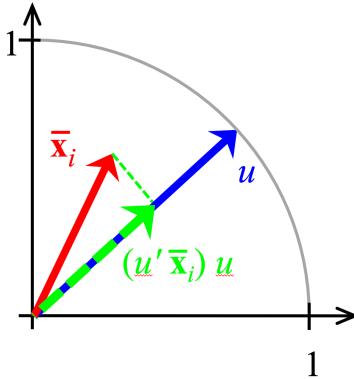


Figure 19: Projecting a point $\bar{\mathbf{x}}_i$ on a direction vector u : the inner product $u' \bar{\mathbf{x}}_i$ is the distance of $\bar{\mathbf{x}}_i$ from the origin along the direction given by u .

Now repeat this procedure: In iteration k , the k -th PC u_k is constructed by projecting pattern points to the linear subspace that is orthogonal to the already computed PCs u_1, \dots, u_{k-1} , and u_k is obtained as the unit-length vector pointing in the “longest” direction of the current $(n - k + 1)$ -dimensional pattern point distribution. This can be repeated until n PCs u_1, \dots, u_n have been determined. They form an orthonormal coordinate system of \mathbb{R}^n . Figure 18C shows this situation, and Figure 18D visualizes the PCs plotted into the original data cloud.

Now define features f_k (where $1 \leq k \leq n$) by

$$f_k : \mathbb{R}^n \rightarrow \mathbb{R}, \quad \mathbf{x} \mapsto u'_k \bar{\mathbf{x}}, \quad (27)$$

that is, $f_k(\bar{\mathbf{x}})$ is the projection component of $\bar{\mathbf{x}}$ on u_k . Since the n PCs form an orthonormal coordinate system, any point $\mathbf{x} \in \mathbb{R}^n$ can be perfectly reconstructed from its feature values by

$$\mathbf{x} = \mu + \sum_{k=1,\dots,n} f_k(\mathbf{x}) u_k. \quad (28)$$

The PCs and the corresponding features f_k can be used for dimension reduction as follows. We select the first (“leading”) PCs u_1, \dots, u_m up to some index m . Then we obtain a feature map

$$\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad \mathbf{x} \mapsto (f_1(\mathbf{x}), \dots, f_m(\mathbf{x}))'. \quad (29)$$

At the beginning of this section I spoke of a decoding function $\mathbf{d} : \mathbb{R}^m \rightarrow \mathbb{R}^n$ which should recover the original patterns \mathbf{x} from their feature vectors $\mathbf{f}(\mathbf{x})$. In our PCA story, this decoding function is given by

$$\mathbf{d} : (f_1(\mathbf{x}), \dots, f_m(\mathbf{x}))' \mapsto \mu + \sum_{k=1}^m f_k(\mathbf{x}) u_k. \quad (30)$$

How “good” is this dimension reduction, that is, how similar are the original patterns \mathbf{x}_i to their reconstructions $\mathbf{d} \circ \mathbf{f}(\mathbf{x}_i)$?

If dissimilarity of two patterns $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ is measured in the square error sense by

$$\delta(\mathbf{x}_1, \mathbf{x}_2) := \|\mathbf{x}_1 - \mathbf{x}_2\|^2,$$

a full answer can be given. Let

$$\sigma_k^2 = 1/N \sum_i f_k(\mathbf{x}_i)^2$$

denote the variance of the feature values $f_k(\mathbf{x}_i)$ (notice that the mean of the $f_k(\mathbf{x}_i)$, taken over all patterns, is zero, so σ_k^2 is indeed their variance). Then the mean square distance between patterns and their reconstructions is

$$1/N \sum_i \|\mathbf{x}_i - \mathbf{d} \circ \mathbf{f}(\mathbf{x}_i)\|^2 = \sum_{k=m+1}^n \sigma_k^2. \quad (31)$$

A derivation of this result is given in Appendix E.

Equation (31) gives an absolute value for dissimilarity. For applications however the relative amount of dissimilarity compared to the mean variance of patterns is more instructive. It is given by

$$\frac{1/N \sum_i \|\mathbf{x}_i - \mathbf{d} \circ \mathbf{f}(\mathbf{x}_i)\|^2}{1/N \sum_i \|\bar{\mathbf{x}}_i\|^2} = \frac{\sum_{k=m+1}^n \sigma_k^2}{\sum_{k=1}^m \sigma_k^2}. \quad (32)$$

Real-world sets of patterns often exhibit a rapid (roughly exponential) decay of the feature variances as their index k grows. The ratio (32) then is very small compared to the mean size $E[\|\bar{\mathbf{X}}\|^2]$ of patterns, that is, only very little information is lost by reducing the dimension from n to m via PCA. Our visualization from Figure 18 is not doing justice to the amount of compression savings that is often possible. Typical real-world, high-dimensional, centered data clouds in \mathbb{R}^n are often very “flat” in the vast majority of directions in \mathbb{R}^n and these directions can all be zeroed without much damage by the PCA projection.

Note. Using the word “variance” above is not mathematically correct. Variance is the *expected* squared deviation from a *population* mean, a statistical property which is different from the *averaged* squared deviation from a *sample* mean. It would have been more proper to speak of an “empirical variance” above, or an “estimated variance”. Check out Appendix D!

4.4 Mathematical properties of PCA and an algorithm to compute PCs

The key to analysing and computing PCA is the $(n \times n)$ -dimensional covariance matrix $C = 1/N \sum_i \bar{\mathbf{x}}_i \bar{\mathbf{x}}_i'$, which can be easily obtained from the centered training data matrix $\bar{\mathbf{X}} = [\bar{\mathbf{x}}_1, \dots, \bar{\mathbf{x}}_N]$ by $C = 1/N \bar{\mathbf{X}} \bar{\mathbf{X}}'$. C is very directly related to PCA by the following facts:

1. The PCs u_1, \dots, u_n form a set of orthonormal, real eigenvectors of C .
2. The feature variances $\sigma_1^2, \dots, \sigma_n^2$ are the eigenvalues of these eigenvectors.

A derivation of these facts can be found in my legacy ML lecture notes, Section 4.4.1 (http://minds.jacobs-university.de/uploads/teaching/lectureNotes/LN_ML_Fall11.pdf). Thus, the principal component vectors u_k and their associated data variances σ_k^2 can be directly gleaned from C .

Computing a set of unit-norm eigenvectors and eigenvalues from C can be most conveniently done by computing the *singular value decomposition* (SVD) of C . Algorithms for computing SVDs of arbitrary matrices are shipped with all numerical or statistical mathematics software packages, like Matlab, R, or Python with numpy. At this point let it suffice to say that every covariance matrix C is a so-called *positive semi-definite* matrix. These matrices have many nice properties. Specifically, their eigenvectors are orthogonal and real, and their eigenvalues are real and nonnegative.

In general, when an SVD algorithm is run on an n -dimensional positive semi-definite matrix C , it returns a factorization

$$C = U \Sigma U',$$

where U is an $n \times n$ matrix whose columns are the normed orthogonal eigenvectors u_1, \dots, u_n of C and where Σ is an $n \times n$ diagonal matrix which has the eigenvalues $\lambda_1, \dots, \lambda_n$ on its diagonal. They are usually arranged in descending order. Thus, computing the SVD of $C = U \Sigma U'$ directly gives us the desired PC vectors u_k , lined up in U , and the variances σ_k^2 , which appear as the eigenvalues of C , collected in Σ .

This enables a convenient control of the goodness of similarity that one wants to ensure. For example, if one wishes to preserve 98% of the variance information from the original patterns, one can use the r.h.s. of (32) to determine the “cutoff” m such that the ratio in this equation is about 0.02.

4.5 Summary of PCA based dimension reduction procedure

Data. A set $(\mathbf{x}_i)_{i=1,\dots,N}$ of n -dimensional pattern vectors.

Result. An n dimensional mean pattern vector μ and m principal component vectors arranged column-wise in an $n \times m$ sized matrix U_m .

Procedure.

Step 1. Compute the pattern mean μ and center the patterns to obtain a centered pattern matrix $\bar{X} = [\bar{\mathbf{x}}_1, \dots, \bar{\mathbf{x}}_N]$.

Step 2. Compute the SVD $U \Sigma U'$ of $C = 1/N \bar{X} \bar{X}'$ and keep from U only the first m columns, making for a $n \times m$ sized matrix U_m .

Usage for compression. In order to compress a new n -dimensional pattern to a m dimensional feature vector $\mathbf{f}(\mathbf{x})$, compute $\mathbf{f}(\mathbf{x}) = U'_m \bar{\mathbf{x}}$.

Usage for uncompression (decoding). In order to approximately restore \mathbf{x} from its feature vector $\mathbf{f}(\mathbf{x})$, compute $\mathbf{x}_{\text{restored}} = \mu + U_m \mathbf{f}(\mathbf{x})$.

4.6 Eigendigits

For a demonstration of dimension reduction by PCA, consider the “3” digit images. After reshaping the images into 240-dimensional grayscale vectors and centering and computing the PCA on the basis of $N = 100$ training examples, we obtain 240 PCs u_k associated with variances σ_k^2 . Only the first 99 of these variances are nonzero (because the 100 image vectors \mathbf{x}_i span a 100-dimensional subspace in \mathbb{R}^{240} ; after centering the $\bar{\mathbf{x}}_i$ however span only a 99-dimensional subspace – *why? homework exercise!* – thus the matrix $C = 1/N \bar{X} \bar{X}'$ has rank at most the rank of \bar{X} , which is 99), thus only the first 99 PCs are useable. Figure 20 **A** shows some of these eigenvectors u_i rendered as 15×16 grayscale images. It is customary to call such PC re-visualizations *eigenimages*, in our case “eigendigits”. (If you have some spare time, do a Google image search for “eigenfaces” and you will find weird-looking visualizations of PC vectors obtained from PCA carried out on face pictures.)

Figure 20 **B** shows the variances σ_i^2 of the first 99 PCs. You can see the rapid (roughly exponential) decay. Aiming for a dissimilarity ratio (Equation 32) of 0.1 gives a value of $m = 32$. Figure 20 **C** shows the reconstructions of some “3” patterns from the first m PC features using (30).

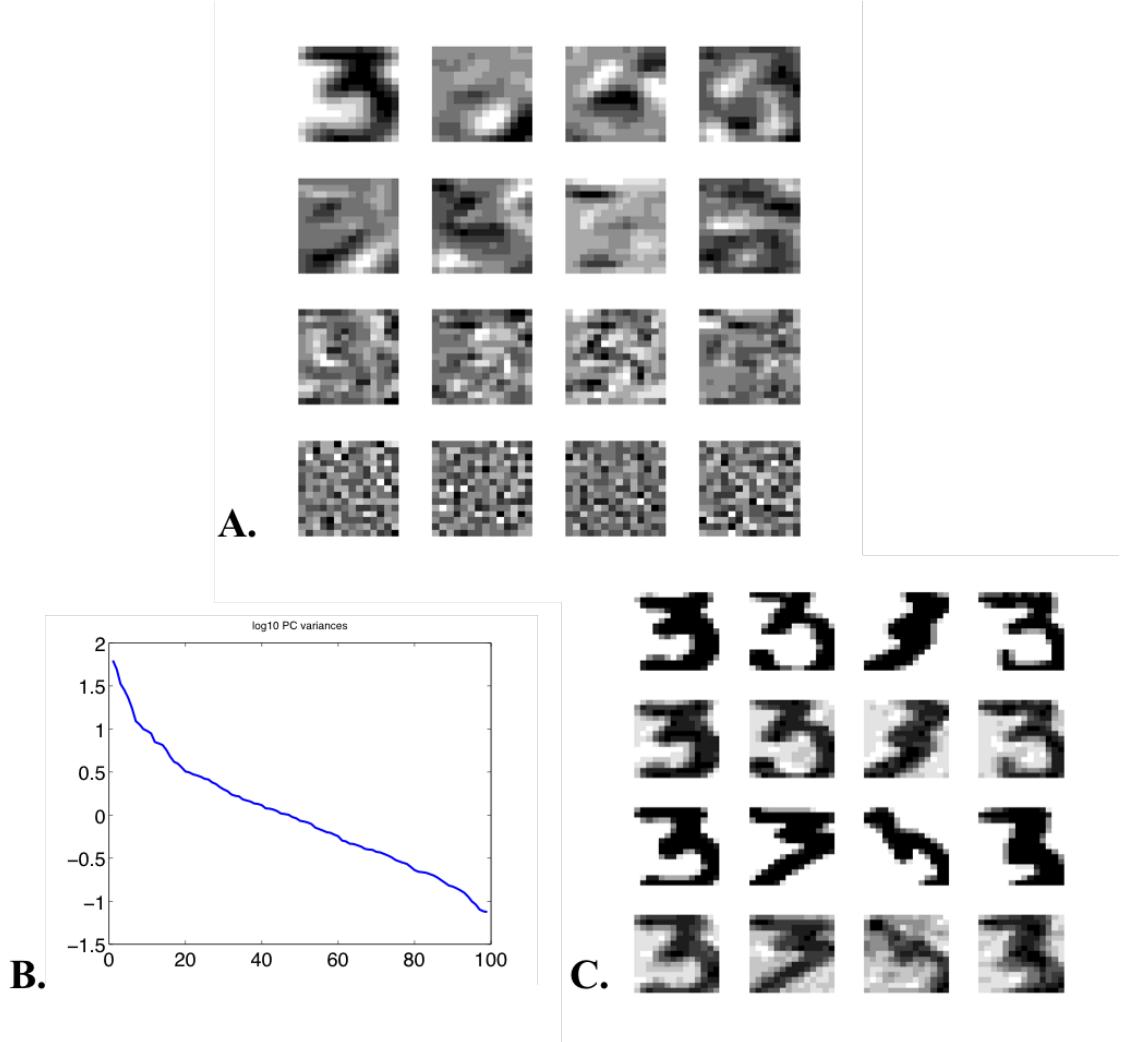


Figure 20: **A.** Visualization of a PCA computed from the “3” training images. Top left panel shows the mean μ , the next 7 panels (row-wise) show the first 7 PCs. Third row shows PCs 20–23, last row PCs 96–99. Grayscale values have been automatically scaled per panel such that they spread from pure white to pure black; they do not indicate absolute values of the components of PC vectors. **B.** The (log10 of) variances of the PC features on the “3” training examples. **C.** Reconstructions of digit “3” images from the first $m = 32$ features, corresponding to a reconstitution of 90% of the original image dataset variance. First row: 4 original images from the training set. Second row: their reconstructions. Third row: 4 original images from the test set. Last row: their reconstruction.

4.7 Self-organizing maps

Nature herself has been facing the problem of dimension reduction when She was using evolution as a learning algorithm to obtain well-performing brains. Specifically, in human visual processing the input dimension n is in the order of a million (judging from the number of axons in the optical nerve), but the “space” where the incoming n -dimensional raw visual signal is processed has only $m = 2$ dimensions: it is the walnut-kernel-curly surface of the brain, the *cortical sheet*. Now this is certainly an over-simplified view of biological neural information processing, but at least in some brain areas and for some kinds of input projecting into those areas, it seems fairly established in neuroscience that an incoming high-dimensional signal becomes mapped on an $m = 2$ dimensional representation on the cortical sheet.

In the 1980 decade, Teuvo Kohonen developed an artificial neural network model which (i) could explain how this neural dimension reduction could be learnt in biological brains; and (ii) which could be used as a practical machine learning algorithm for dimension reduction. His model is today known as *Kohonen Network* or *Self-Organizing Map (SOM)*. The SOM model is one the very few and precious instances of a neural information processing model which is anchored in, and can bridge between, both neuroscience and machine learning. It made a strong impact in both communities. In the machine learning / data analysis world SOMs have been superseded in later years by other methods, and in my perception they are only rarely used today. But this may again change when non-digital neuromorphic microchip technologies will mature. In neuroscience research, SOM-inspired models continue to be explored. Furthermore, the SOM model is simple and intuitive. I think all of this is reason enough to include them in this lecture.

The learning task for SOMs is defined as follows (for the case of $m = 2$):

Given: a pattern collection $\mathcal{P} = (\mathbf{x}_i)_{i=1,\dots,N}$ of points in \mathbb{R}^n .

Given: a 2-dimensional grid of “neurons” $\mathcal{V} = (v_{kl})_{1 \leq k \leq K, 1 \leq l \leq L}$, where k, l are the grid coordinates of neuron v_{kl} .

Learning objective: find a map $\kappa : \mathcal{P} \rightarrow \mathcal{V}$ which

1. preserves input space topology (the metric topology of \mathbb{R}^n), that is, points \mathbf{x}_j in the neighborhood of a point \mathbf{x}_i should be mapped to $\kappa(\mathbf{x}_i)$ or to grid neighbors of $\kappa(\mathbf{x}_i)$, and which
2. covers the pattern collection \mathcal{P} evenly, that is, all grid neurons v_{kl} should have approximately the same number of κ -preimages.

Two comments: (i) Here I use a grid with a rectangular neighborhood structure. Classical SOM papers and many applications of SOMs use a hexagonal neighborhood structure instead, where each neuron has 6 neighbors, all at the same distance. (ii) The “learning objective” sounds vague. It is. At the time

when Kohonen introduced SOMs, tailoring learning algorithms along well-defined loss functions was not standard. Kohonen's modeling attitude was biological modeling oriented and a mathematical analysis of the SOM algorithm was not part of his agenda. In fact the problem to find a loss function which is minimized by the original SOM algorithm was still unsolved in the year 2008 (Yin, 2008). I don't know what the current state of research in this respect is — I would guess not much has happened since.

In a trained SOM, each grid neuron v_{kl} is connected to the input pattern space \mathbb{R}^n by an n -dimensional weight (row) vector $\mathbf{w}(v_{kl})$. If a new test pattern $\mathbf{x} \in \mathbb{R}^n$ arrives, its κ -image is computed by

$$\kappa(\mathbf{x}) = \operatorname{argmax}_{v_{kl} \in \mathcal{V}} \mathbf{w}(v_{kl})\mathbf{x}. \quad (33)$$

In words, the neuron v whose weight vector $\mathbf{w}(v)$ best matches the input pattern \mathbf{x} is chosen. In the SOM literature this neuron is called the *best matching unit* (BMU). Clearly the map κ is determined by the weight vectors $\mathbf{w}(v)$. In order to train them on the training data set \mathcal{P} , a basic SOM learning algorithm works as follows:

Initialization: The weights $\mathbf{w}(v_{kl})$ are set to small random values.

- Iterate until convergence:**
1. Randomly select a training pattern $\mathbf{x} \in \mathcal{P}$.
 2. Determine the BMU v_{BMU} for this pattern, based on the current weight vectors $\mathbf{w}(v)$.
 3. Update all the grid's weight vectors $\mathbf{w}(v_{kl})$ according to the formula

$$\mathbf{w}(v_{kl}) \leftarrow \mathbf{w}(v_{kl}) + \lambda f_r(d(v_{kl}, v_{\text{BMU}})) (\mathbf{x} - \mathbf{w}(v_{kl})). \quad (34)$$

4. Make r a little smaller before entering the next iteration.

In this sketch of an algorithm, λ is a *learning rate* — a small number, for instance $\lambda = 0.01$, plugged in for numerical stability. The function d is the Euclidean distance (between two neurons positions on the grid). $f_r : \mathbb{R}^{\geq 0} \rightarrow \mathbb{R}^{\geq 0}$ is a non-negative, monotonously decreasing function defined on the nonnegative reals which satisfies $f_r(0) = 1$ and which goes to zero as its argument grows. The function is parametrized by a radius parameter $r > 0$, where greater values of r spread out the range of f_r . A common choice is to use a Gaussian-like function $f_r(x) = \exp(-x^2/r^2)$.

Here is an intuitive explanation of this learning algorithm. When a new training pattern \mathbf{x} has been presented and the BMU has been determined, each weight vector $\mathbf{w}(v_{kl})$ is adapted a little bit. The weight update formula (34) adds to $\mathbf{w}(v_{kl})$ a small multiple of $\mathbf{x} - \mathbf{w}(v_{kl})$. This pulls $\mathbf{w}(v_{kl})$ into the direction of \mathbf{x} . How strong this pull is depends on how close v_{kl} is to v_{BMU} — the closer, the stronger. This is regulated by the distance-dependent factor $f_r(d(v_{kl}, v_{\text{BMU}}))$. Obviously the adaptation is strongest for the BMU. When (in the early phases of the

algorithm) r is large, neurons in the wider vicinity of the BMU will likewise receive rather strong weight adjustments, while only far-away neurons remain more or less unaffected.

This mechanics has the effect that after convergence, the training dataset \mathcal{P} will be covered by grid neurons rather evenly (see objective nr. 2 stated above). To get an intuition why this is so, let us consider a specific scenario. Assume that the pattern set \mathcal{P} contains a dense cluster of mutually quite similar patterns \mathbf{x} , besides a number of other, dissimilar patterns. Furthermore assume that we are in an early stage of the learning process, where the radius r is still rather large, and also assume that at this early stadium of learning, each pattern from the cluster yields the same BMU v_0 . Due to the large number of members in the cluster, patterns from that cluster will be drawn for the learning algorithm rather often. With r large, this will have the effect that neurons in the wider neighborhood of the BMU v_0 will grow their weight vectors toward $\mathbf{w}(v_0)$. After some time, v_0 will be surrounded by grid neurons whose weight vectors are all similar to $\mathbf{w}(v_0)$, and $\mathbf{w}(v_0)$ will roughly be the mean of all patterns \mathbf{x} in the cluster. Now, some patterns \mathbf{x}' in the cluster will start to have as their BMU not v_0 any longer, but some of its surrounding neighbors (why?). As a consequence, increasingly many patterns in the cluster will best-match the weight vectors of an increasing number of neighbors of v_0 : the subpopulation of grid neurons which respond to cluster patterns has grown from the singleton population $\{v_0\}$ to a larger one. This growth will continue until the population of grid neurons responding to cluster patterns has become so large that each member's BMU-response-rate has become too low to further drive this population's expansion.

The radius r is set to large values initially in order to let all (or most) patterns in \mathcal{P} compete with each other, leading to a coarse global organization of the emerging map κ . In later iterations, increasingly smaller r leads to a fine-balancing of the BMU responses of patterns that are similar to each other.

SOM learning algorithms come in many variations. I sketched an arbitrary exemplar. The core idea is always the same. Setting up a SOM learning algorithm and tuning it is not always easy – the weight initialization, the decrease schedule for r , the learning rate, the random sampling strategy of training patterns from \mathcal{P} , the grid dimension (2 or 3 or even more... 2 is the most common choice), or the pattern preprocessing (for instance, normalizing all patterns to the same norm) are all design decisions that can have a strong impact on the convergence properties and final result quality of the learning process.

Yin (2008) includes a brief survey of SOM algorithm variants. I mention in passing an algorithm which has its roots in SOMs but is quite significantly different: The *Neural Gas* algorithm (good brief intro in https://en.wikipedia.org/wiki/Neural_gas), like SOMs, leads to a collection of neurons which respond to patterns from a training set through trainable weight vectors. The main difference is that the neurons are not spatially arranged on a grid but are spatially uncoupled from each other (hence, neural “gas”). The spatially defined distance d appearing

in the adaptation efficacy term $f_r(d(v_{kl}, v_{\text{BMU}}))$ is replaced by a rank ordering: the neuron with the best response to training pattern \mathbf{x} (i.e., the BMU) is adapted most, the unit v with the second best response (i.e., second largest value of $\mathbf{w}(v) \mathbf{x}$) is adapted second most strongly, etc.

For a quick SOM demo I used a version of a legacy Matlab toolbox published by Kohonen's own research group (I downloaded it almost 20 years ago). As a pattern dataset \mathcal{P} I used the Digits dataset that I also used before in this section. I used 100 examples from each digit class. Figure 21 shows the result of training an 8×8 neuron grid on this dataset. As expected, the ten digit classes become represented each by approximately the same number of grid neurons, reflecting the fact that the classes were represented in \mathcal{P} in equal shares.

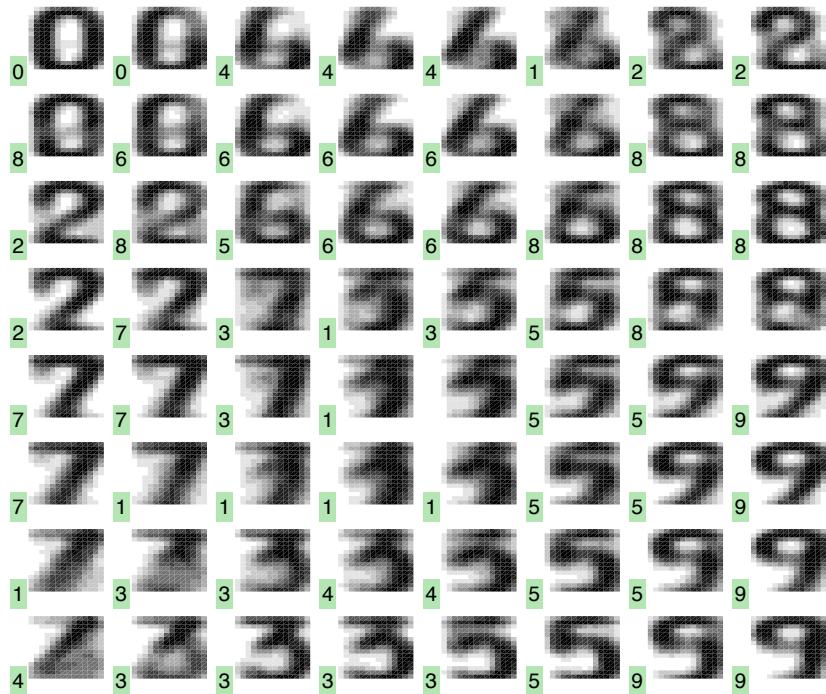


Figure 21: An 8×8 SOM representation of the Digits training dataset. The grayscale images show the weight vectors $\mathbf{w}(v_{kl})$, reshaped back to the rectangular pixel image format. The numbers in the small green insets give the most frequent class label of all training patterns which had the respective grid neuron as BMU. If this number is missing, the grid neuron never fired as BMU for any training pattern.

Just for the fun of it, I also generated an unbalanced pattern set that had 180 examples of class "5" and 20 examples of every other class. The result is shown in Figure 22. As it should be, roughly half of the SOM neurons are covering the "5" class examples, while the other SOM neurons reach out their weight vectors into the remaining classes.

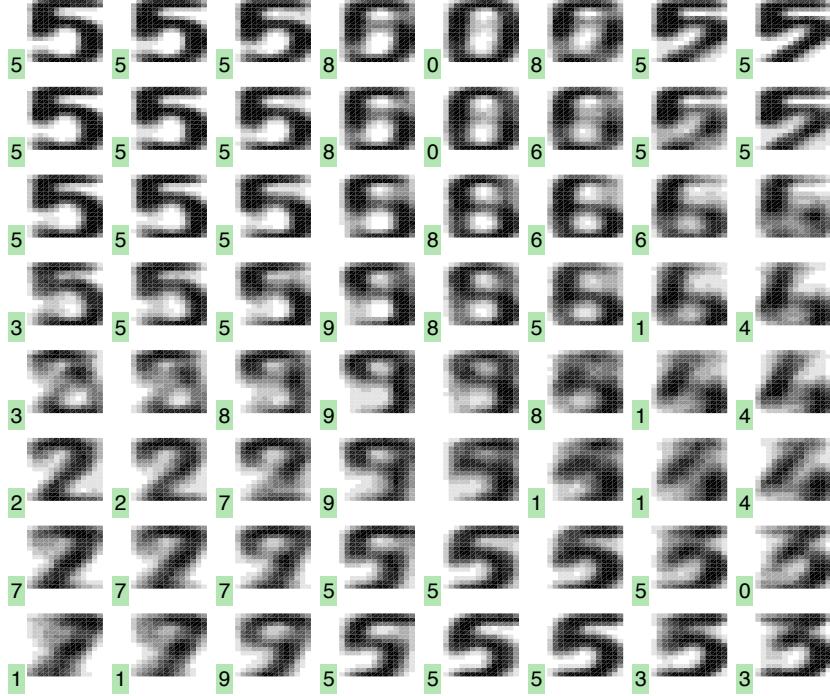


Figure 22: Similar to previous figure, but with an unbalanced pattern set where the number of “5” examples was the same as the total number of all other classes.

Practical uses of SOMs appear to have been mostly as visualization tools for exploring (labelled) high-dimensional datasets in two-dimensional graphics. Specifically, such SOM visualizations can give insight into metric similarities between pattern classes. For instance, inspecting the bottom right 3×3 panels in Figure 21, one finds “morphable” similarities between certain versions of “5” and “9” patterns.

What I find more relevant and interesting about SOMs is their use in neurosciences. Learning mechanisms similar to the SOM learning algorithm have been (and are being) invoked to explain the 2-dimensional spatial organization of *cortical maps*. They can explain how neurons on a surface patch of the cortical sheet align their specific responsiveness to high-dimensional input (from sensors or other brain areas) with their local 2-dimensional metric neighborhood. Pictures which put synthetic SOM grids side to side with images recorded from small patches of cortical surface have variously been published. Figure 23 gives an example from a study by Swindale and Bauer (1998).

If you are interested in such themes, I can recommend the recent review Bednar and Wilson (2016) on cortical maps as a starting point.

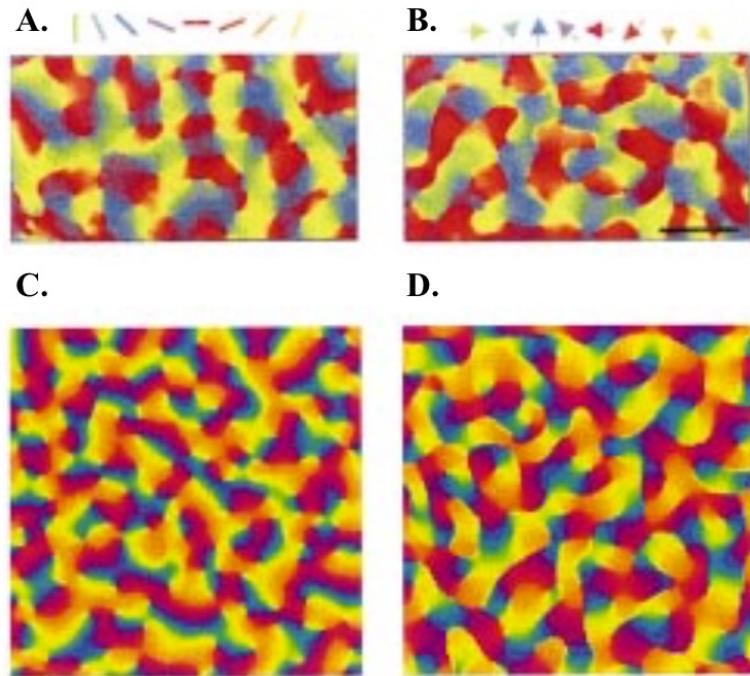


Figure 23: The SOM algorithm reproducing biological cortical response patterns. The scenario: an anaesthetized but eye-open ferret is shown moving images of a bar with varying orientation and direction of movement, while response activity from neurons on a patch (about 10 square mm) of visual cortex is recorded. **A.** A color-coded recording of neural response activity depending on the orientation of the visually presented bar. For instance, if the bar was shown in horizontal orientation, the neurons who responded to this orientation with maximal activity are rendered in red. **B.** Like in panel A., but for the motion direction of the bar (same cortical patch). **C., D.** Artificial similes of A., B. generated with a SOM algorithm. Figures taken from Swindale and Bauer (1998), who in turn took the panels A. and B from Weliky et al. (1996).

4.8 Summary discussion. Model reduction, data compression, dimension reduction

Reducing the dimensionality of patterns which are represented by vectors is important not only in machine learning but everywhere in science and engineering, where you find it explored under different headline names and for different purposes:

Dimension reduction is a term used in machine learning and statistics. The core motivation is first, that statistical and machine learning models typically depend on an estimate of a probability distribution over some “pattern” space (this distribution estimate may remain implicit); second, that the curse of dimensionality makes it intrinsically hard to estimate distributions from training data points in high-dimensional vector spaces, such that third, a key to success is to reduce the dimension of the “raw” patterns. Typically, the low-dimensional target format is again real-valued vectors (“feature vectors”).

Data compression is a term used in signal processing and communication technology. The practical and economical importance of being able to compress and uncompress bulky data is obvious. The compressed data may have a different type than the raw data. For instance, in vector quantization compression, a real-valued vector will become encoded in a natural number, the index of its associated codebook vector. Again there is a close connection with probability, established through Shannon information theory.

Model reduction is a term used when it comes to trim down not just static “data points” but entire dynamical “system models”. All branches of science and engineering today deal with models of complex physical dynamical systems which are instantiated as systems of coupled differential equations — often millions, sometimes billions ... or more ... of them. One obtains such gargantuan systems of coupled ODEs almost immediately when one discretizes system models expressed in PDEs. Such systems cannot be numerically solved on today’s computing hardware and need to be dramatically shrunk before a simulation can be attempted. I am not familiar with this field. The mathematical tools and leading intuitions are different from the ones that guide dimension reduction in machine learning. I mention this field for completeness and because the name “model reduction” invites misleading analogies with dimension reduction. Antoulas and Sorensen (2001) give a tutorial overview with instructive examples.

In this section we took a look at three methods for dimension reduction of high-dimensional “raw” data vectors, namely K-means clustering, PCA, and SOMs. While at first sight these methods appear quite different from each other, there is a unifying view which connects them. In all three of them, the reduction was done

by introducing a comparatively simple kind of geometric object in the original high-dimensional pattern space \mathbb{R}^n , which was then used to re-express raw data points in a lightweight encoding:

1. In K-means-clustering, this object is the set of codebook vectors, which can be used to compress a test data point to the mere natural number index of its associated codebook vector; or which can be used to give a reduced K -dimensional vector comprised of the distances α_j to the codebook vectors.
2. In PCA, this object is the m -dimensional linear (affine) hyperplane spanned by the first m eigenvectors of the data covariance matrix. An n -dimensional test point is represented by its m coordinates in this hyperplane.
3. In SOMs, this object is the grid of SOM neurons v , including their associated weight vectors $\mathbf{w}(v)$. A new test data point can be compressed to the natural number index of its BMU. This is entirely analog to how the codebook vectors in K-means clustering can be used. Furthermore, if an m -dimensional neuron grid is used (we considered only $m = 2$ above), the grid plane is nonlinearly “folded” into the original dataspace \mathbb{R}^n , in that every grid point v becomes projected to $\mathbf{w}(v)$. It thus becomes an m -dimensional manifold embedded in \mathbb{R}^n , which is characterized by codebook vectors. This can be seen as a nonlinear generalization of the m -dimensional linear affine hyperplanes embedded in \mathbb{R}^n in PCA.

Thus, the SOM shares properties with both K-means clustering and PCA. In fact, one can systematically explore a whole spectrum of dimension reduction / data compression algorithms which are located between K-means clustering and PCA, in the sense that they describe m -dimensional manifolds of different degrees of nonlinearity through codebook vectors. K-means clustering is the extreme case that uses only codebook vectors and no manifolds; PCA is the other extreme with only manifolds and no codebook vectors. The extensive Preface to the collection volume *Principal Manifolds for Data Visualization and Dimension Reduction* (Gorban et al., 2008) gives a readable intro to this interesting field.

In today’s deep learning practice one often ignores the traditional methods treated in this section. Instead one immediately fires the big cannon, training a deep neural network wired up in an *auto-encoder* architecture. An autoencoder network is a multilayer feedforward network whose output layer has the same large dimension n as the input layer. It is trained in a supervised way, using training data $(\mathbf{x}_i, \mathbf{y}_i)$ to approximate the identity function: the training output data \mathbf{y}_i are identical to the input patterns \mathbf{x}_i (possibly up to some noise added to the inputs). The trick is to insert a “bottleneck” layer with only $m \ll n$ neurons into the layer sequence of the network. In order to achieve a good approximation of the n -dimensional identity map on the training data, the network has to discover an $n \rightarrow m$ -dimensional compression mapping which preserves most of the information that is needed to describe the training data points. I will not give an introduction

to autoencoder networks in this course (it's a topic for Marco Wiering's "Deep Learning" course). The Deep Learning standard reference Goodfellow et al. (2016) has an entire section on autoencoders.

5 Discrete symbolic versus continuous real-valued

I hope this section will be as useful as it will be short and simple. Underneath it, however, lurks a mysterious riddle of mathematics, philosophy and the neurosciences.

Some data are given in symbolic form, for instance

- texts of all sorts,
- yes/no or rating scale questionnaire items,
- DNA and protein sequence data,
- records of chess or Go matches,
- large parts in public administration databases.

Others are real-valued (scalar, vector, matrix or array formatted) in principle, notwithstanding the circumstance that on digital machines real numbers must be approximated by finite-length binary strings. Examples of data that are best understood as “continuous” are

- images, video, speech,
- measurement data from technical systems, e.g. in production process control or engine monitoring,
- financial data,
- environmental and weather data,
- signals used in robot motion control.

Many mathematical formalisms can be sorted in two categories: “discrete” and “continuous”. Mathematicians, in fact, come in two kinds – ask one, and he/she will be able to tell you whether he/she feels more like a discrete or continuous mathematician. Discrete sorts of mathematics and discrete mathematicians typically operate in areas of number theory, set theory, logic, algebra, graph theory, automata theory. Continuous maths / mathematicians, that is linear algebra, functional analysis, and calculus.

Computer scientists are almost always discrete-math-minded: 0 and 1 and algorithms and Boolean circuits and AI knowledge representation formalisms are eminently discrete.

Physicists almost always continuous-math minded, because their world is made of continuous space, time, matter, forces and fields.

There are some mathematical domains that are neither, or open to both, especially topology and probability. Some of the most advanced and ingenious modern

fields of mathematics arise from crossover formalisms between the Discrete and the Continuous. The fundamental difference between the two is not dissolved in these theories, but the tension between the Discrete and the Continuous sets free new forms of mathematical energy. Sadly, these lines of research are beyond what I understand and what I can explain (or even name), and certainly beyond what is currently used in machine learning.

The hiatus (an educated word of latin origin, meaning “dividing gap”) between the Discrete and the Continuous is also the source of one of the great unresolved riddles in the neurosciences, cognitive science and AI: how can symbolic reasoning (utterly discrete) emerge from the continuous matter and signal processing in our material brains (very physical, very continuous)? This question has kept AI researchers and philosophers busy (and sometimes aggressively angry with one another) for 5 decades now and is not resolved; if you are interested, you can get a first flavor in the Wikipedia article on “Physical symbol system” or by reading up on the overview articles listed in <http://www.neural-symbolic.org/>.

Back to our down-to-earth business. Machine learning formalisms and algorithms likewise are often either discrete-flavored or continuous-flavored. The former feed on symbolic data and create symbolic results, using tools like decision trees, Bayesian networks and graphical models (including hidden Markov models), inductive logic, and certain sorts of neural networks where neurons have 0-1-valued activations (Hopfield networks, Boltzmann machines). The latter digest vector data and generate vector output, with deep neural network methods currently being the dominating workhorse which overshadow more traditional ones like support vector machines and various sorts of regression learning “machines”.

The great built-in advantage of discrete formalisms is that they lend themselves well to *explainable*, human-understandable solutions. Their typical disadvantage is that learning or inference algorithms are often based on combinatorial search, which quickly lets computing times explode. In contrast, continuous formalisms typically lead to results that cannot be intuitively interpreted – vectors don’t talk – but lend themselves to nicely, smoothly converging optimization algorithms.

When one speaks of “machine learning” today, one mostly has vector processing methods in mind. Also this RUG course is focussing on vector data. The discrete strands of machine learning are more associated with what one often calls “data mining”. This terminology is however not clearly defined. I might mention that I have a brother, Manfred Jaeger (<http://people.cs.aau.dk/~jaeger/>), who is also a “machine learning” prof, but he is from the discrete quarters while I am more on the continuous side. We often meet and talk, but not about science, because we don’t understand what the respective other researches; we publish in different journals and go to different conferences.

Sometimes one has vector data but wants to exploit benefits that come with discrete methods, or conversely, one has symbolic data and wants to use a neural network (because everybody else seems to be using them, or because one doesn’t want to fight with combinatorial explosions). Furthermore, many an interesting

dataset comes as a mix of symbolic-discrete and numerical-continuous data – for instance, data originating from questionnaires or financial/business/admin data often are mixed-sort.

Then, one way to go is to *convert* discrete data to vectors or vice versa. It is a highly empowering professional skill to know about basic methods of discrete \leftrightarrow continuous conversions.

Here are some discrete-to-continuous transformations:

One-hot encodings. Given: data points a_ν that are symbols from a finite “alphabet” $A = \{a_1, \dots, a_k\}$. Examples: yes/no answers in a questionnaire; words from a vocabulary; nucleic acids A, C, T, T occurring in DNA. Turn each a_ν into the k -dimensional binary vector $v_\nu \in \{0, 1\}^k$ which is zero everywhere except at position ν . This is a very common way to present symbolic input to neural networks. On the output side of a neural network (or any other regression learning machine), one-hot encodings are also often used to give vector teacher data in classification tasks: if (\mathbf{x}_i, c_i) is a classification-task training dataset, where c_i is a symbolic class label from a class set $C = \{c_1, \dots, c_k\}$, transform each c_i to its k -dimensional one-hot vector v_i and get a purely vector-type training dataset (\mathbf{x}_i, v_i) .

Binary pattern encodings. If the symbol alphabet A is of a large size k , one might shy away from one-hot encoding because it gives large vectors. Instead, encode each a_ν into a binary vector of length $\lceil \log_2 k \rceil$, where $\lceil \log_2 k \rceil$ is the smallest integer larger or equal to $\log_2 k$. Advantage: small vectors. Disadvantage: subsequent vector-processing algorithms must invest substantial nonlinear effort into decoding this essentially arbitrary encoding. This will mostly be crippling, and binary pattern encodings should only be considered if there is some intuitive logic in the encoding.

Linear scale encoding. If there is some natural ordering in the symbols $a_\nu \in A$, encode every a_ν by the index number ν . Makes sense, for instance, when the a_ν come from a Likert-scale questionnaire item, as in

$$A = \{\text{certainly not, rather not, don't know, rather yes, certainly yes}\}$$

Semantic word vectors. This is a technique which enabled breakthroughs in deep learning for text processing (I mentioned it in the Introduction; it is used in the TICS demo). By a nontrivial machine learning algorithm pioneered by Mikolov et al. (2013), convert words a_ν from a (large) vocabulary A into vectors of some reasonably large size, say 300-dimensional, such that semantically related words become mapped to vectors that are metrically close to each other.

For instance, code vectors v, v' for words $w = \text{airplane}$ and $w' = \text{aircraft}$ should have small distance, whereas the code vector v'' for $w'' = \text{rose}$ should lie at a great distance to both v and v' . So the goal is to find vectors v, v', v'' in this example that have small distance $\|v - v'\|$ and large distances $\|v - v''\|, \|v' - v''\|$. This is achieved by measuring the similarity of two words w, w' through counting how often they occur in similar locations in training texts. A large collection of English texts is processed, collecting statistics about similar sub-phrases in those texts that differed only in the two words whose similarity one wished to assess (plus, there was another trick: can you think of an important improvement of this basic idea?). – Mikolov's paper has been (Google-scholar) cited more than 16,000 times in merely six years!

And here are some continuous-to-discrete transformations:

One-dimensional discretization, also known as binning. Given: a real number dataset $\{x_i\}$ where $x_i \in \mathbb{R}$. Divide the real line into k segments

$$A_1, A_2, \dots, A_{k-1}, A_k = (-\infty, a_1], (a_1, a_2], \dots, (a_{k-2}, a_{k-1}], (a_{k-1}, \infty)$$

. Then transform each x_i to that symbol A_ν for which $x_i \in A_\nu$. Comments:

- The segments $(a_\nu, a_{\nu+1}]$ are called *bins*.
- In the simplest case, the range $[\min\{x_i\}, \max\{x_i\}]$ of the data points is split into k bins of equal length.
- Alternatively, another way of splitting the range $[\min\{x_i\}, \max\{x_i\}]$ is to define the bins such that each bin will contain an approximately equal number of data points from the set $\{x_i\}$.
- Often the best way of splitting the data range into bins depends on the task. This can turn out to be a delicate optimization problem. For instance, there is a sizeable literature in deep learning which is concerned with the question of how one can reduce the numerical precision of weight parameters in neural networks. One first trains the neural network using standard high-precision learning algorithms, obtaining weight matrices in floating-point precision. In order to speed up computing, use small microchips on hand-held devices, or reduce energy consumption when using the trained network, one wishes to reduce the bit precision of the weight matrices. In extreme cases, one wants to end up with only three weight values $\{-1, 0, 1\}$, which would lead to very cheap neural network computations at use time. This amounts to the splitting the range of floating-point precision weights obtained after training into just three bins. The goal is to ensure that the reduced-precision network performs approximately as well as the high-precision original. This is a nontrivial problem. Check out the (randomly chosen) reference Indiveri (2015) if you want to get a taste of the flavor.

Multi-dimensional discretization by hierarchical refinement. If one wants

to discretize a set $\{\mathbf{x}_i\}$ of n -dimensional vectors, one has to split the n -dimensional volume which contains the points $\{\mathbf{x}_i\}$ into a finite set of discrete regions R_ν . A common approach is to let these regions be n -dimensional hypercubes. By a process of hierarchical refinement one constructs these regions R_ν such that in areas where there is a higher point density, or in areas where there is much fine-grained information encoded in the local point distribution, one uses smaller hypercubes to increase resolution. This leads to a tree structure, which in the case $n = 2$ is called a *quadtree* (because every non-leaf node has four children) and in the case $n = 3$ an *octree* of hierarchically nested hypercubes. This tree structure enables a computationally efficient indexing of the hypercubes. The left panel in Figure 24 shows an example. The Wikipedia article on quadtrees is quite instructive. One may also opt for regions R_ν of other polygonal shape (see right panel in Figure 24). There are many such *mesh refinement* methods, with task-specific optimization criteria. They are not typically used in machine learning but rather in methods for simulating spatiotemporal (fluid) dynamics by numerically solving PDEs. Still, it is good to know that such methods exist.

Vector quantization. Using K-means or other clustering algorithms, the vector set $\{\mathbf{x}_i\}$ is partitioned into k cells whose center of gravity vectors are indexed and the indices are used as symbolic encodings of the $\{\mathbf{x}_i\}$. We have seen this in Section 4.2. This *is* a typical machine learning method for discretization.

Turning neural dynamics into symbol sequences. When we (I really mean “we”, = us humans!) speak or write, the continuous-time, continuous-valued neural brain dynamics leads to a discrete sequence of words. Somehow, this symbolic sequence is encoded in the “subsymbolic”, analog brain dynamics. It is unknown *how*, exactly, this encoding is realized. Numerous proposals based on nonlinear dynamical systems theory have been made. This is an area of research in which I am personally engaged. If you are interested: partial selections of some approaches are listed in Durstewitz et al. (2000), Pascanu and Jaeger (2011), Fusi and Wang (2016). In machine learning, the problem of transforming continuous-valued neural state sequences to sequences of words (or letters) arises in applications like speech recognition (“speech-to-text”) or gesture recognition. Here the most common solution (which is not necessarily biologically plausible) is to use the core neural network to generate a sequence of continuous-valued *hypothesis* output vectors with as many components as there are possible target symbols. At each point in time, the numerical value of each vector component reflects a current “degree of belief” about which symbol should be generated. With some postprocessing mechanism (not easy to set up), this hypothesis stream is denoised and turned into a symbol sequence, for instance by selecting at each point in time the symbol that has the largest degree of belief.

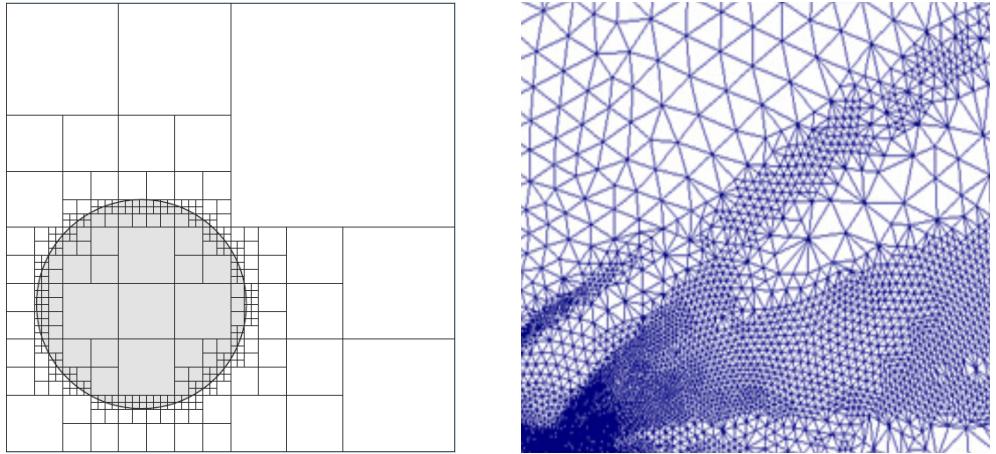


Figure 24: Left: A hierarchical hypercube mesh for a 2-dimensional dataset consisting of points homogeneously distributed inside a circle. Right: adaptive non-orthogonal meshing, here used in an airflow simulation. Sources: left panel: <http://www.questinygroup.com/tag/quad-tree/>; right panel: Luchinsky and et al. (2012).

6 The bias-variance dilemma and how to cope with it

Reality is rich in detail and surprises. Measuring reality gives a finite number of data points – and the reality between and beyond these data points remains unmeasured, a source of unforeseeable surprises. Furthermore, measuring is almost always “noisy” – imprecise or prone to errors. And finally, each space-time event point of reality can be measured in innumerable ways, of which only a small choice is actually measured. For instance, the textbook measurement of “tossing a coin” events is just to record “heads” or “tail”. But one *could* also measure the weight and color of coin, the wind speed, the falling altitude, the surface structure of the ground where the coin falls, and ... everything else. In summary, any dataset will capture only a supremely scarce coverage of space-time events whose unfathomable qualitative richness has been reduced to a ridiculously small number of “observables”. From this impoverished, punctuated information basis, called “training data”, a machine learning algorithm will generate a “model” of a part of reality. This model will then be used to predict properties of *new* space-time events, called “test inputs”. How should it be ever possible for a model to “know” anything about those parts of reality that lie between the pinpoints hit by the training data?

This would be a good point to give up.

But ... quoting Einstein: “*Subtle is the Lord, but malicious He is not*” (https://en.wikiquote.org/wiki/Albert_Einstein). Reality is co-operative. Between

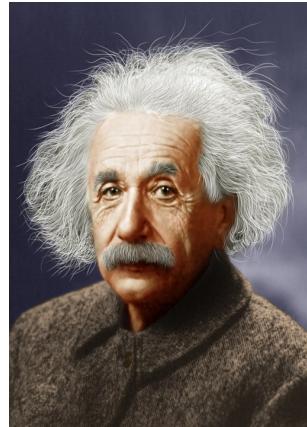


Figure 25: He gave the reason why machine learning works (from https://commons.wikimedia.org/wiki/File:Albert_Einstein--Colorized.jpg)

measurement points, reality changes not arbitrarily but in a lawful manner. The question is, *which* law. In machine learning, this question is known as the problem of *overfitting*, or in more educated terms, the *bias-variance dilemma*. Welcome to this chapter which is all about this question. For all practical exploits of machine learning in your professional future, this is the most helpful chapter in this course.

6.1 Training and testing errors

Let us take a closer look at the archetypical machine learning task of classifying handwritten digits. We use the simple Digits benchmark data which you know from Section 4.1. The task is specified as follows:

Training data. $(\mathbf{x}_i^{\text{train}}, c_i^{\text{train}})_{i=1,\dots,1000}$ where the $\mathbf{x}_i^{\text{train}} \in [0, 1]^{240}$ are 240-dimensional image vectors, whose components correspond to the 15×16 pixels which have normalized grayscale values ranging in $[0, 1]$; and where the c_i^{train} are class labels from the set $\{0, 1, \dots, 9\}$. There are 100 examples from each class in the training data set.

Test data. Another set $(\mathbf{x}_i^{\text{test}}, c_i^{\text{test}})_{i=1,\dots,1000}$, also having 100 exemplars of each digit.

Task specification. This is a benchmark dataset with a standardized task specification: train a classifier using (only) the training data, then test it on the test data and report the rate of misclassifications. The loss function is thus the count loss given in Equation 4.

An elementary but professionally structured learning pipeline which uses methods that we have already described in this course goes as follows (it may be a

welcome rehearsal to give a step-by-step instruction):

- First stage: dimension reduction by PCA:**
1. Center the set of image vectors $(\mathbf{x}_i^{\text{train}})_{i=1,\dots,1000}$ by subtracting the mean vector μ , obtaining $(\bar{\mathbf{x}}_i^{\text{train}})_{i=1,\dots,1000}$.
 2. Assemble the centered image vectors column-wise in a 240×1000 matrix \bar{X} . Compute the covariance matrix $C = 1/1000 X X'$ and factorize C into its SVD $C = U \Sigma U'$. The columns of U are the 240 principal component vectors of C .
 3. Decide how strongly you want to reduce the dimension, shrinking it from $n = 240$ to $m < n$. Let U_m be the matrix made from the first m columns from U .
 4. Project the centered patterns $\bar{\mathbf{x}}_i^{\text{train}}$ on the m first principal components, obtaining m -dimensional feature vectors $\mathbf{f}_i^{\text{train}} = U_m' \bar{\mathbf{x}}_i^{\text{train}}$.

Vectorize the class labels by one-hot encoding: Each c_i^{train} is re-written as a binary 10-dimensional vector v_i^{train} which has a 1 entry in the corresponding class c_i . Assemble these vectors column-wise in a 10×1000 matrix V .

Compute a linear regression classifier: Assemble the 1000 m -dimensional feature vectors $\mathbf{f}_i^{\text{train}}$ into a $m \times 1000$ dimensional matrix F and obtain a $10 \times m$ dimensional regression weight matrix W by

$$W' = (F F')^{-1} F V'.$$

Compute the training MSE and training error rate: The training mean square error (MSE) is given by

$$\text{MSE}^{\text{train}} = 1/1000 \sum_{i=1}^{1000} \|v_i^{\text{train}} - W(\mathbf{f}_i^{\text{train}})\|^2.$$

The training misclassification rate is

$$\varrho^{\text{train}} = 1/1000 |\{i \mid \text{maxInd } W \mathbf{f}_i^{\text{train}} - 1 \neq c_i^{\text{train}}\}|,$$

where `maxInd` picks the index of the maximal element in a vector. (Note: the “ -1 is owed to the fact that the vector indices range from 1 to 10, while the class labels go from 0 to 9).

Similarly, compute the test MSE and error rates by

$$\text{MSE}^{\text{test}} = 1/1000 \sum_{i=1}^{1000} \|v_i^{\text{test}} - W(\mathbf{f}_i^{\text{test}})\|^2$$

and

$$\varrho^{\text{test}} = 1/1000 |\{i \mid \text{maxInd } W \mathbf{f}_i^{\text{test}} - 1 \neq c_i^{\text{test}}\}|,$$

where $\mathbf{f}_i^{\text{test}} = U_m' (\mathbf{x}_i^{\text{test}} - \mu)$. Note: for centering the test data, use the mean μ obtained from the training data!

This is a procedure made from basic linear operations that you should command even when sleepwalking; with some practice the entire thing should not take you more than 30 minutes for programming and running. Altogether a handy quick-and-not-very-dirty routine that you should consider carrying out in every classification task, in order to get a baseline before you start exploring more sophisticated methods.

And now - let us draw what is probably the most helpful and worth burning into your subconscious graphics in these lecture notes. Figure 26 shows these diagnostics for all possible choices of the number $m = 1, \dots, 240$ of PC features used.

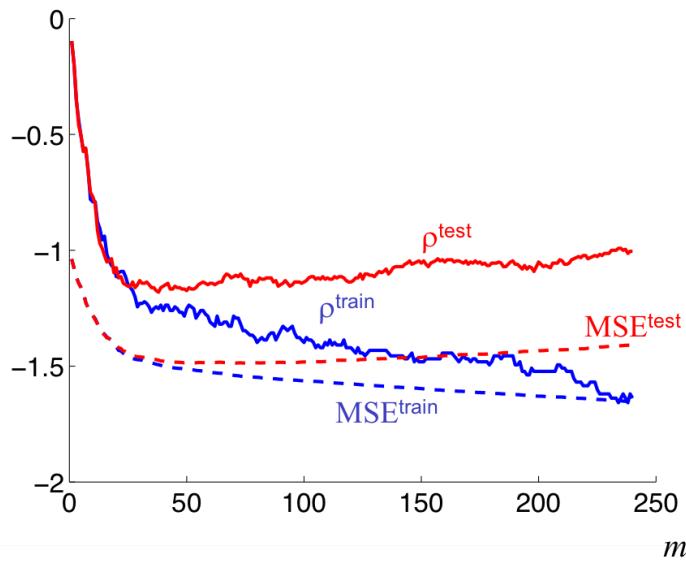


Figure 26: Dashed: Train (blue) and test (red) MSE obtained for $m = 1, \dots, 240$ PCs. Solid: Train (blue) and test (red) misclassification rates. The y -axis is logarithmic base 10.

This plot visualizes one of the most important issues in (supervised) machine learning and deserves a number of comments.

- As m increases (the x -axis in the plot), the number of parameters in the corresponding linear regression weight matrices W grows by $10 \cdot m$. More model parameters means more degree of freedoms, more “flexible” models. With greater m , models can increasingly better solve the linear regression learning equation (20). This is evident from the monotonous decrease of the train MSE curve. The training misclassification rate also decreases persistently except for a jitter that is due to the fact that we optimized models only indirectly for low misclassification.

- The analog performance curves for the testing MSE and misclassification first exhibit a decrease, followed by an increasing tail. The testing misclassification rate is minimal for $m = 34$.
- This “first decrease, then increase” behavior of testing MSE (or classification rate) is *always* observed in supervised learning tasks when models are compared which have growing degrees of data fitting flexibility. In our digit example, this increase in flexibility was afforded by growing numbers of PC features, which in turn gave the final linear regression a richer repertoire of feature values to combine into the hypothesis vectors.
- The increasing tail of testing MSE (or classification rate) is the hallmark of *overfitting*. When the learning algorithm admits too much flexibility, the resulting model can fit itself not only to what is “lawful” in the training data, but also to the random fluctuations in the training data. Intuitively and geometrically speaking, a learning algorithm that can shape many degrees of freedom in its learnt models allows the models to “fold in curls and wiggles to accomodate the random whims of the training data”. But then, the random curls and wiggles of the learnt model will be at odds with fresh testing data.

6.2 The menace of overfitting – it’s real, it’s everywhere

ML research has invented a great variety of model types for an even larger variety of supervised and unsupervised learning tasks. How exactly overfitting (mis-)functions in each of these model types will need a separate geometrical analysis in each case. Because overfitting is such a fundamental challenge in machine learning, I illustrate its geometrical manifestations with four examples.

6.2.1 Example 1: polynomial curve-fitting

This example is *the* standard textbook example for demonstrating overfitting. Let us consider a one-dimensional input, one-dimensional output regression task of the kind where the training data are of form $(x_i, y_i) \in \mathbb{R} \times \mathbb{R}$. Assume that there is some systematic relationship $y = h(x)$ that we want to recover from the training data. We consider a simple artificial case where the x_i range in $[0, 1]$ and the to-be-discovered true functional relationship is $y = \sin(2\pi x)$. The training data, however, contain a noise component, that is, $y_i = \sin(2\pi x_i) + \nu_i$, where ν_i is drawn from a normal distribution with zero mean and standard deviation σ . Figure 27 shows a training sample $(x_i, y_i)_{i=1,\dots,11}$, where $N = 11$ training points x_i are chosen equidistantly.

We now want to solve the task of learning a good approximation for h from the training data (x_i, y_i) by applying polynomial curve fitting, an elementary technique you might be surprised to meet here as a case of machine learning. Consider an

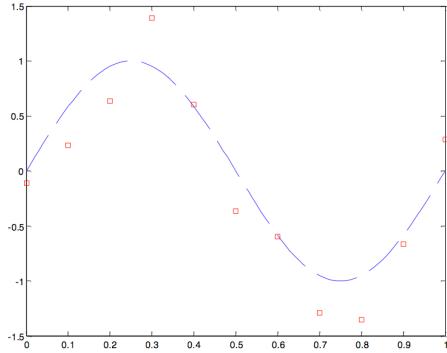


Figure 27: An example of training data (red squares) obtained from a noisy observation of an underlying “correct” function $\sin(2\pi x)$ (dashed blue line).

m -th order polynomial

$$p(x) = w_0 + w_1 x + \cdots + w_m x^m. \quad (35)$$

We want to approximate the function given to us via the training sample by a polynomial, that is, we want to find (“learn”) a polynomial $p(x)$ such that $p(x_i) \approx y_i$. More precisely, we want to minimize the mean square error on the training data

$$\text{MSE}^{\text{train}} = \frac{1}{N} \sum_{i=1}^N (p(x_i) - y_i)^2.$$

At this moment we don’t bother how this task is solved computationally but simply rely on the Matlab function *polyfit* which does exactly this job for us: given data points (x_i, y_i) and polynomial order m , find the coefficients w_j which minimize this MSE. Figure 28 shows the polynomials found in this way for $m = 1, 3, 10$.

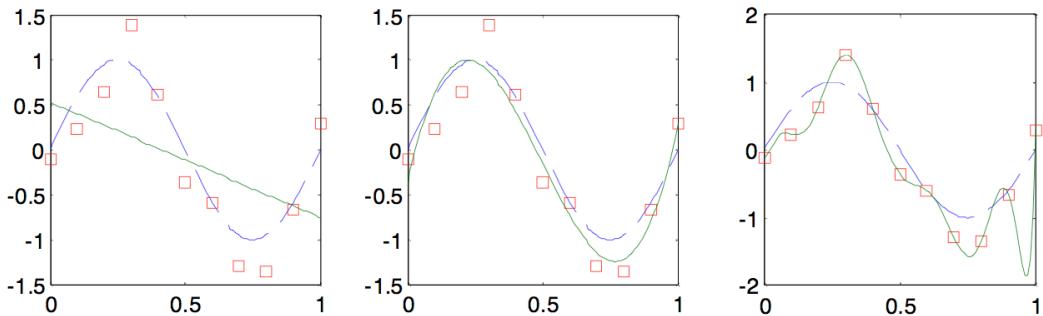


Figure 28: Fitting polynomials (green lines) for polynomial orders 1, 3, 10 (from left to right).

If we compute the MSE's for the three orders $m = 1, 3, 10$, we get $\text{MSE}^{\text{train}} = 0.4852, 0.0703, 0.0000$ respectively. Some observations:

- If we increase the order m , we get increasingly lower $\text{MSE}^{\text{train}}$.
- For $m = 1$, we get a linear polynomial, which apparently does not represent our original sine function well (underfitting).
- For $m = 3$, we get a polynomial that hits our target sine apparently quite well.
- For $m = 10$, we get a polynomial that perfectly matches the training data, but apparently misses the target sine function (overfitting).

The modelling flexibility is here defined through the polynomial order m . If it is too small, the models are too inflexible and *underfit*; if it is too large, we see overfitting.

However, please switch now into your most critical thinking mode and reconsider what I have just said. *Why*, indeed, should we judge the linear fit “underfitting”, the order-3 fit “seems ok”, and the order-10 fit “overfitting”? There is no other ground for justifying these judgements than our visual intuition! In fact, the order-10 fit might be the right one if the data contain no noise! the order-1 fit might be the best one if the data contain a lot of noise! We don't know!

6.2.2 Example 2: pdf estimation

Let us consider the task of estimating a 2-dimensional pdf over the unit square from 6 given training data points $\{x_i\}_{i=1,\dots,6}$, where each x_i is in $[0, 1] \times [0, 1]$. This is an elementary unsupervised learning task, the likes of which frequently occur as a subtask in more involved learning tasks, but which is of interest in its own right too. Figure 29 shows three pdfs which were obtained from three different learning runs with models of increasing flexibility (I don't explain the modeling method here — for the ones who know about it: simple Gaussian Parzen-window models where the degree of admitted flexibility was tuned by kernel width). Again we encounter the fingerprints of under/overfitting: the low-flexibility model seems too “unbending” to resolve any structure in the training point cloud (underfitting), the high-flexibility model is so volatile that it can accommodate each individual training point (presumably overfitting).

But again, we don't really know...

6.2.3 Example 3: learning a decision boundary

Figure 30 shows a schematic of a classification learning task where the training patterns are points in \mathbb{R}^2 and come in two classes. When the trained model is too inflexible (left panel), the decision boundary is confined to be a straight line,

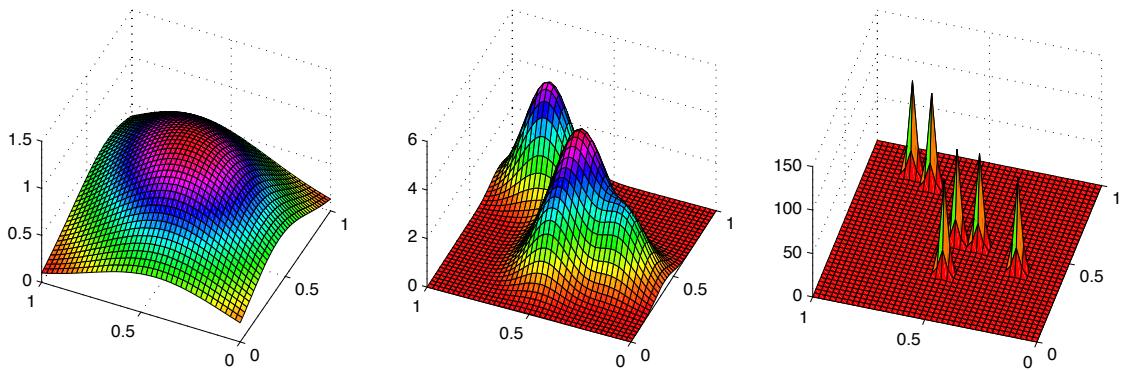


Figure 29: Estimating a pdf from 6 data points. Model flexibility grows from left to right. Note the different scalings of the z -axis: the integral of the pdf is 1 in each of the three cases.

presumably underfitting. When the flexibility is too large, each individual training point can be “lasso-ed” by a sling of the decision boundary, presumably overfitting.

Do I need to repeat that while these graphics *seem* to indicate under- or overfitting, we do not actually *know*?

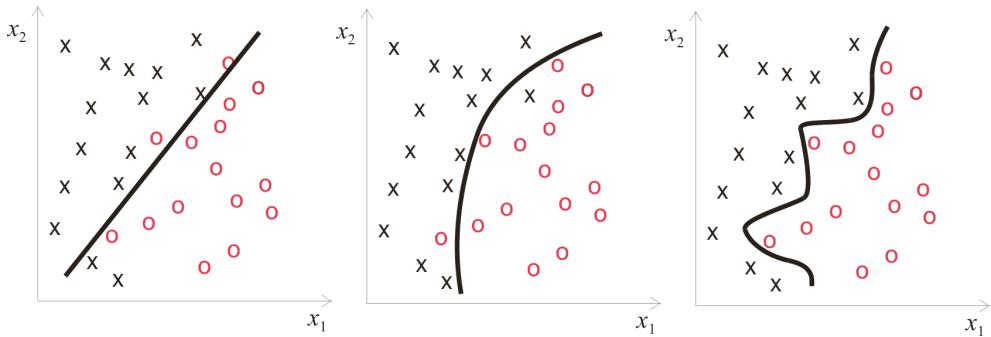


Figure 30: Learning a decision boundary for a 2-class classification task of 2-dimensional patterns (marked by black “ x ” and red “ o ”).

6.2.4 Example 4: furniture design

Overfitting can also hit you in your sleep, see Figure 31.

Again, while this looks like drastic overfitting, it would be just right if all humans sleep in the same folded way as the person whose sleep shape was used for training the mattress model did.



Figure 31: A nightmare case of overfitting. Picture spotted by Yasin Cibuk (2018 ML course participant) on <http://dominicwilcox.com/portfolio/bed/>, designed and crafted by artist Dominic Wilcox, 1999. Quote from the artist's description of this object: "I used my own body as a template for the mattress". From a ML point of view, this means a size $N = 1$ training data set.

6.2.5 Interim summary

The four examples stem from different learning tasks (function approximation, pdf learning, classification learning, furniture optimization), and correspondingly the overfitting problem manifests itself in different geometrical ways. But the flavor is the same in all cases. The model flexibility determines how “wiggly” the geometry of the learnt model can become. Very large flexibility ultimately admits to adapt the model to each individual training point. This leads to small, even zero, training error; but it is likely disastrous for generalization to new test data points. Very low flexibility can hardly adapt to the structure of the training data at all, likewise leading to poor test performance (and poor training performance too). Some intermediate flexibility is likely to strike the best balance.

Properly speaking, flexibility is not a characteristic of the final model that one obtains after learning, but of the learning algorithm. If one uses the term precisely, one should speak, for example, of “the flexibility of the procedure to train a third-order polynomial function”, or of “the flexibility of the PCA-based linear regression learning scheme which uses m PCA features”.

I introduced this way of speaking about “flexibility” ad hoc. In statistics and machine learning textbooks you will hardly find this term. Instead, specific methods to measure and tune the flexibility of a learning algorithm have their specific names, and it is these names that you will find in the literature. The

most famous among them is *model capacity*. This concept has been developed in a field now called *statistical learning theory*, and (not only) I consider it a highlight of modern ML theory. We will however not treat it in this lecture, since it is not an easy concept and it needs to be spelled out in different versions for different types of learning tasks and algorithms. Check out en.wikipedia.org/wiki/Vapnik-Chervonenkis_theory if you want to get an impression. Instead, in Sections 6.4.1 and 6.4.2 I will present two simpler methods for handling modeling flexibility which, while they lack the analytical beauty and depth of the model capacity concept, are immensely useful in practice.

I emphasize that finding the right flexibility for a learning algorithm is ab-so-lute-ly crucial for good performance of ML algorithms. Our little visual examples do not do justice to the dismal effects that overfitting may have in real-life learning tasks where a high dimension of patterns is combined with a small number of training examples — which is a situation faced very often by ML engineers in practical applications.

6.3 An abstract view on supervised learning

Before I continue with the discussion of modeling flexibility, it is helpful to introduce some standard theoretical concepts and terminology. Before you start reading this section, make sure that you understand the difference between the *expectation* of a random variable and the *sample mean* (explained in Appendix D).

In supervised learning scenarios, one starts from a training sample of the form $(x_i, y_i)_{i=1,\dots,N}$, which is drawn from a joint distribution $P_{X,Y}$ of two random variables X and Y . So far, we have focussed on tasks where the x_i were vectors and the y_i were class labels or numbers or vectors, but supervised learning tasks can be defined for any kind variables. In this subsection we will take an abstract view and just consider any kind of supervised learning based on a training sample $(x_i, y_i)_{i=1,\dots,N}$. We then call the x_i the *arguments* and the y_i the *targets* of the learning task. The target RV Y is also sometimes referred to as the *teacher* variable.

The argument and target variables each come with their specific *data value space* – a set of possible values that the RVs X and Y may take. For instance, in our digit classification example, the data value space for X was \mathbb{R}^{240} (or more constrained, $[0, 1]^{240}$) and the data value space for Y was the set of class labels $\{0, 1, \dots, 9\}$. After the extraction of m features and turning the class labels to class indicator vectors, the original picture–label pairs (x_i, y_i) turned into pairs (\mathbf{f}_i, v_i) with data value spaces $\mathbb{R}^m, \{0, 1\}^k$ respectively. In English-French sentence translation tasks, the data value spaces of the random variables X and Y would be a (mathematical representation of a) set of English and French sentences. We now abstract away from such concrete data value spaces and write $\mathcal{E}_X, \mathcal{E}_Y$ for the data value spaces of X and Y .

Generally speaking, the aim of a supervised learning task is to derive from the training sample a function $h : \mathcal{E}_X \rightarrow \mathcal{E}_Y$. We called this function a *decision function* earlier in these lecture notes, and indeed that is the term which is used in abstract statistical learning theory.

Some of the following I already explained before (Section 2.3) but it will be helpful if I repeat this here, with a little more detail.

The decision function h obtained by a learning algorithm should be optimized toward some objective. One introduces the concept of a *loss function*. A loss function is a function

$$L : \mathcal{E}_Y \times \mathcal{E}_Y \rightarrow \mathbb{R}^{\geq 0}. \quad (36)$$

The idea is that a loss function measures the “cost” of a mismatch between the target values y and the values $h(x)$ returned by a decision function. Higher cost means lower quality of h . We have met two concrete loss functions so far:

- A loss that counts misclassifications in pattern classification: when the decision function returns a class label, define

$$L(h(x), y) = \begin{cases} 0, & \text{if } h(x) = y \\ 1, & \text{if } h(x) \neq y \end{cases} \quad (37)$$

- A loss that penalizes quadratic errors of vector-valued targets:

$$L(h(x), y) = \|h(x) - y\|^2. \quad (38)$$

This loss is often just called “quadratic loss”. We used it as a basis for deriving the algorithm for linear regression in Section 3.1.

The decision function h is the outcome of a learning algorithm, which in turn is informed by a sample $(x_i, y_i)_{i=1,\dots,N}$. Learning algorithms should minimize the *expected* loss, that is, a good learning algorithm should yield a decision function D whose *risk*

$$R(h) = E[L(h(X), Y)] \quad (39)$$

is small. The expectation here is taken with respect to the true underlying joint distribution $P_{X,Y}$. For example, in a case where X and Y are numerical RVs and their joint distribution is described by a pdf f , the risk of a decision function h would be given by

$$R(h) = \int_{\mathcal{E}_X \times \mathcal{E}_Y} L(h(x), y) f(x, y) d(x, y).$$

However, the true distribution f is unknown. The mission to find a decision function h which minimizes (39) is, in fact, hopeless. The only access to $P_{X,Y}$ that the learning algorithm affords is the scattered reflection of $P_{X,Y}$ in the training sample $(x_i, y_i)_{i=1,\dots,N}$.

A natural escape from this impasse is to tune a learning algorithm such that instead of attempting to minimize the risk (39) it tries to minimize the *empirical risk*

$$R^{\text{emp}}(h) = 1/N \sum_{i=1}^N L(h(x_i), y_i), \quad (40)$$

which is just the mean loss calculated over the training examples. Minimizing this empirical risk is an achievable goal, and a host of optimization algorithms for all kinds of supervised learning tasks exist which do exactly this, that is, they find

$$h_{\text{opt}} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^N L(h(x_i), y_i). \quad (41)$$

The set \mathcal{H} is the *hypothesis space* – the search space within which a learning may look for an optimal h .

It is important to realize that every learning algorithm comes with a specific hypothesis space. For instance, in decision tree learning \mathcal{H} is the set of all decision trees that use a given set of properties and attributes. Or, in linear regression, \mathcal{H} is the set of all affine linear functions from \mathbb{R}^n to \mathbb{R}^k . Or, if one sets up a neural network learning algorithm, \mathcal{H} is typically the set of all neural networks that have a specific connection structure (number of neuron layers, number of neurons per layer); the networks in \mathcal{H} then differ from each other by the weights associated with the synaptic connections.

The empirical risk is often – especially in numerical function approximation tasks – also referred to as *training error*.

While minimizing the empirical loss is a natural way of coping with the impossibility of minimizing the risk, it may lead to decision functions which combine a low empirical risk with a high risk. This is the ugly face of overfitting which I highlighted in the previous subsection. In extreme cases, one may learn a decision function which has zero empirical risk and yet has a extremely large expected testing error which makes it absolutely useless.

There is no easy or general solution for this conundrum. It has spurred statisticians and mathematicians to develop a rich body of theories which analyze the relationships between risk and empirical risk, and suggest insightful strategies to manage as well as one can in order to keep the risk within provable bounds. These theories, sometimes referred to as *statistical learning theory* (or better, *theories*), are beyond the scope of this lecture.

If you are in a hardcore mood and if you have some background in probability theory, you can inspect Section 18 of lecture notes of my legacy “Principles of Statistical Modeling” course (online at <http://minds.jacobs-university.de/teaching/ln/>). You will find that the definitions of loss and risk given there in the spirit of mathematical statistics are a bit more involved than what I presented above, but the definitions of loss and risk that I gave here are used in textbooks on machine learning.

6.4 Tuning model flexibility

In order to deal with the overfitting problem, one must have ways to tune the flexibility of the learning algorithm and set it to the “right” value. There are many, quite different, ways of tuning flexibility. In this section I outline the most common ones.

6.4.1 Tuning learning flexibility through model class size

In the Digits classification demo from Section 6.1, we tuned flexibility by changing the dimension of the PCA features. In the polynomial curve fitting demo in Section 6.2.1 we changed the order of polynomials. Let us re-consider these two examples to get a clearer picture:

- In the Digits example (Figure 26) we found that the number m of principal component features that were extracted from the raw image vectors was decisive for the testing error. When m was too small, the resulting models were too simple to distinguish properly between different digit image classes (underfitting). When m was too large, overfitting resulted. Fixing a particular m determines the class \mathcal{H} of candidate decision functions within which the empirical risk (41) is minimized. Specifically, using a fixed m meant that the optimial decision function h_{opt} was selected from the set \mathcal{H}_m which contains all decision functions which first extract m principal component features before carrying out the linear regression. It is clear that \mathcal{H}_{m-1} is contained in \mathcal{H}_m , because decision functions that only combine the first $m - 1$ principal component features into the hypothesis vector can be regarded as special cases of decision functions that combine m principal component features into the hypothesis vector, namely those whose linear combination weight for the m -th feature is zero.
- In the polynomial curve fitting example from Section 6.2.1, the model parameters were the monomial coefficients w_0, \dots, w_m (compare Equation 35). After fixing the polynomial order m , the optimal decision function $p(x)$ was selected from the set $\mathcal{H}_m = \{p : \mathbb{R} \rightarrow \mathbb{R} \mid p(x) = \sum_{j=0}^m w_j x^j\}$. Again it is clear that \mathcal{H}_{m-1} is contained in \mathcal{H}_m .

A note on terminology: I use the words “decision function” and “model” as synonyms, meaning the (classification) algorithm h which results from a learning procedure. The word “decision function” is standard in theoretical statistics, the word “model” is more common in machine learning.

Generalizing from these two examples, we are now in a position to draw a precise picture of what it may mean to consider learning algorithms of “increasing flexibility”. A *model class inclusion sequence* is a sequence $\mathcal{H}_1 \subset \mathcal{H}_2 \subset \dots \subset \mathcal{H}_L$ of sets of candidate models. Since there are more candidate models in classes that appear later in the sequence, “higher” model classes have more possibilities to

fit the training data, thus the optimal model within class \mathcal{H}_{m+1} can achieve an empirical risk that is at least as low as the optimal model in class \mathcal{H}_m , — but gets you closer to overfitting.

There are many ways how one can set up a sequence of learning algorithms which pick their respective optimal models from such a model class inclusion sequence. In most cases – such as in our two examples – this will just mean to admit larger models with more tuneable parameters for “higher” classes.

From now on we assume that a class inclusion sequence $\mathcal{H}_1 \subset \dots \subset \mathcal{H}_L$ is given. We furthermore assume we have a loss function L and are in possession of a learning algorithm which for every class \mathcal{H}_m can solve the optimization problem of minimizing the empirical risk

$$h_{\text{opt } m} = \operatorname{argmin}_{h \in \mathcal{H}_m} 1/N \sum_{i=1}^N L(h(x_i), y_i). \quad (42)$$

So... how can we find the best model class m_{opt} which gives us the best *risk* – note: *not* the best *empirical* risk? Or stated in more basic terms, which model class will give us the smallest expected test error? Expressed formally, how can we find

$$m_{\text{opt}} = \operatorname{argmin}_m R(h_{\text{opt } m})? \quad (43)$$

6.4.2 Using regularization for tuning modeling flexibility

The flexibility tuning mechanism explained in this subsection is simple, practical, and in widespread use. It is called *model regularization*.

When one uses regularization to vary the modeling flexibility, one does not vary the model class \mathcal{H} at all. Instead, one varies the optimization objective (41) for minimizing the training error.

The basic geometric intuition behind modeling flexibility is that low-flexibility models should be “smooth”, “more linear”, “flatter”, admitting only “soft curvatures” in fitting data; whereas high-flexibility models can yield “peaky”, “rugged”, “sharply twisted” curves (see again Figures 28, 29, 30).

When one uses model regularization, one fixes a single model structure and size with a fixed number of trainable parameters, that is, one fixes \mathcal{H} . Structure and size of the considered model should be rich and large enough *to be able to overfit* (!) the available training data. Thus one can be sure that the “right” model is contained in the search space \mathcal{H} . For instance, in our evergreen digit classification task one would altogether dismiss the dimension reduction through PCA (which has the same effect as using the maximum number $m = 240$ of PC components) and directly use the raw picture vectors (padded by a constant 1 component to enable affine linear maps) as argument vectors for a training a linear regression decision function. Or, in polynomial curve-fitting, one would fix a polynomial order that clearly is too large for the expected kind of true curve.

The models in \mathcal{H} are typically characterized by a set of trainable parameters. In our example of digit classification through linear regression from raw images, these trainable parameters are the elements of the regression weight matrix; in polynomial curve fitting these parameters are the monomial coefficients. Following the traditional notation in the machine learning literature we denote this collection of trainable parameters by θ . This is a vector that has as many components as there are trainable parameters in the chosen kind of model. We assume that we have M tuneable parameters, that is $\theta \in \mathbb{R}^M$.

Such a high-flexibility model type would inevitably lead to overfitting when an “optimal” model would be learnt using the basic learning equation (41) which I repeat here for convenience:

$$h_{\text{opt}} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^N L(h(x_i), y_i).$$

In order to dampen the exaggerated flexibility of this baseline learning algorithm, one adds a *regularization term* (also known as *penalty term*, or simply *regularizer*) to the loss function. A regularization term is a cost function $R : \mathbb{R}^M \rightarrow \mathbb{R}^{\geq 0}$ which penalizes model parameters θ that code models with a high degree of geometrical “wigginess”.

The learning algorithm then is constructed such that it solves, instead of (41), the regularized optimization problem

$$h_{\text{opt}} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^N L(h(x_i), y_i) + \alpha^2 R(\theta_h). \quad (44)$$

where θ_h is the collection of parameter values in the candidate model h .

The design of a useful penalty term is up to your ingenuity. A good penalty term should, of course, assign high penalty values to parameter vectors θ which represent “wiggly” models; but furthermore it should be easy to compute and blend well with the algorithm used for empirical risk minimization.

Two examples of such regularizers:

1. In the polynomial fit task from Section 6.2.1 one might consider for \mathcal{H} all 10th order polynomials, but penalize the “oscillations” seen in the right panel of Figure 28, that is, penalize such 10th order polynomials that exhibit strong oscillations. The degree of “oscillativity” can be measured, for instance, by the integral over the (square of the) second derivative of the polynomial p ,

$$R(\theta) = R((w_0, \dots, w_{10})) = \int_0^1 \left(\frac{d^2 p(x)}{dx^2} \right)^2 dx.$$

Investing a little calculus (good exercise! not too difficult), it can be seen that this integral resolves to a quadratic form $R(\theta) = \theta' C \theta$ where C is an 11×11 sized positive semi-definite matrix. That format is more convenient to use than the original integral version.

2. A popular regularizer that often works well is just the squared sum of all model parameters,

$$R(\theta) = \sum_{w \in \theta} w^2.$$

This regularizer favors models with small absolute parameters, which often amounts to “geometrically soft” models. This regularizer is popular among other reasons because it supports simple algorithmic solutions for minimizing risk functions that contain it. It is called the *L_2 -norm regularizer* because it measures the (squared) L_2 -norm of the parameter vector θ .

Computing a solution to the minimization task (44) means to find a set of parameters which simultaneously minimizes the original risk and the penalty term. The factor α^2 in (44) controls how strongly one wishes the regularization to “soften” the solution. Increasing α means downregulating the model flexibility. For $\alpha^2 = 0$ one returns to the original un-regularized empirical risk (which would likely mean overfitting). For $\alpha^2 \rightarrow \infty$ the regularization term entirely dominates the model optimization and one gets a model which does not care anymore about the training data but instead only is tuned to have minimal regularization penalty. In case of the L_2 norm regularizer this means that all model parameters are zero – the ultimate wiggle-free model; one should indeed say the model is dead.

Let us briefly return to the overfitting diagram in Figure 26. In that demo, the flexibility regulation (moving left or right on the x -axis) was effected by moving in a model class inclusion hierarchy. But similar diagrams would be obtained in any other ML exercise where other methods for navigating on the flexibility axis might be used. Because the essential message of Figure 26 is universal across all machine learning tasks, I redraw that figure and annotate it generically (Figure 32).

Summarizing and emphasizing the main messages of this figure:

- Increasing the model flexibility leads to monotonous decrease of the empirical risk - because training data can be fitted better and better.
- Increasing the model flexibility from very low to very high will lead to a risk that first decreases (less and less underfitting) and then rises again (more and more overfitting).
- The best model flexibility is where the risk curve reaches its minimum. The problem is that the risk is defined on the distribution of “test” data which one does not have at training time.

When regularization is used to steer the degree of model flexibility, the x -axis in Figure 32 would be labelled by α^2 (highest α^2 on the left, lowest at the right end of the x -axis).

Using regularizers to vary model flexibility is often computationally more convenient than using different model sizes, because one does not have to tamper

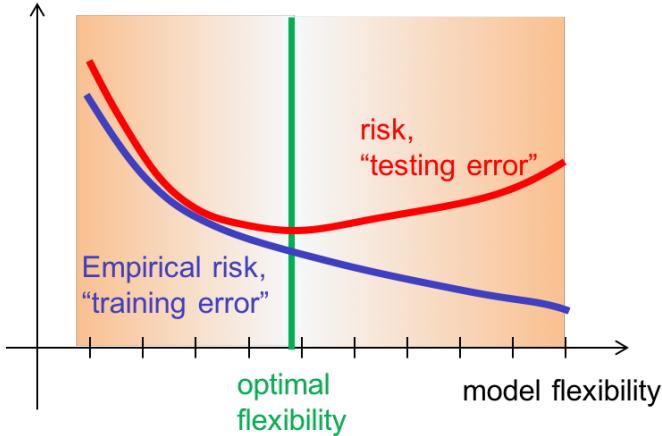


Figure 32: The generic, universal, core challenge of machine learning: finding the right model flexibility which gives the minimal risk.

with differently structured models. One selects a model type with a very large (unregularized) flexibility, which typically means to select a big model with many parameters (maybe hundreds of thousands). Then all one has to do (and one *must* do it – it is a *most crucial* part of any professionally done machine learning project) is to find the optimal degree of flexibility which minimizes the risk. At this moment we don't know how to do that – the secret will be revealed later in this section.

I introduced the word "regularization" here for the specific flexibility-tuning method of adding a penalty term to the loss function. The same word is however often used in a generalized fashion to denote *any* method used for steering model flexibility.

6.4.3 Ridge regression

Let us briefly take a fresh look at linear regression, now in the light of general supervised learning and regularization. Linear regression should always be used in conjunction with regularization. Because this is such a helpful thing to know, I devote this separate subsection to this "trick". Recall from Section 3.1 that the learning task solved by linear regression is to find

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w} \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^N (\mathbf{w} \mathbf{x}_i - y_i)^2, \quad (45)$$

where $(\mathbf{x}_i, y_i)_{i=1,\dots,N}$ is a set of training data with $\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \mathbb{R}$. Like any other supervised learning algorithm, linear regression may lead to overfitting solutions \mathbf{w}_{opt} . It is *always* advisable to control the flexibility of linear regression with an

L_2 norm regularizer, that is, instead of solving (45) go for

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w} \in \mathbb{R}^n}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^N (\mathbf{w} \mathbf{x}_i - y_i)^2 + \alpha^2 \|\mathbf{w}\|^2 \quad (46)$$

and find the best regularization coefficient α^2 . The optimization problem (46) admits a closed-form solution, namely the *ridge regression* formula that we have already met in Equation 21. Rewriting it a little to make it match with the current general scenario, here it is again:

$$\mathbf{w}'_{\text{opt}} = \left(\frac{1}{N} X X' + \alpha^2 I_{n \times n} \right)^{-1} \frac{1}{N} X Y = (X X' + \alpha^2 I_{n \times n})^{-1} X Y, \quad (47)$$

where $X = [x_1, \dots, x_N]$ and $Y = (y_1, \dots, y_N)'$.

In Section 3.1 I motivated to use the ridge regression formula because it warrants numerical stability. Now we see that a more fundamental reason to prefer ridge regression over the basic kind of regression (45) is that it implements L_2 norm regularization. The usefulness of ridge regression as an allround simple baseline tool for supervised learning tasks can hardly be overrated.

6.4.4 Tuning model flexibility through adding noise

Another way to tune model flexibility is to add noise. Noise can be added at different places. Here I discuss three scenarios.

Adding noise to the training input data. If we have a supervised training dataset $(\mathbf{x}_i, y_i)_{i=1, \dots, N}$ with vector patterns \mathbf{x}_i , one can enlarge the training dataset by adding more patterns obtained from the original patterns $(\mathbf{x}_i, y_i)_{i=1, \dots, N}$ by adding some noise vectors to each input pattern \mathbf{x}_i : for each \mathbf{x}_i , add l variants $\mathbf{x}_i + \nu_{i1}, \dots, \mathbf{x}_i + \nu_{il}$ of this pattern to the training data, where the ν_{ij} are i.i.d. random vectors (for instance, uniform or Gaussian noise). This increases the number of training patterns from N to $(l+1)N$. The more such noisy variants are added and the stronger the noise, the more difficult will it be for the learning algorithm to fit all data points, and the smoother the optimal solution becomes – that is, the more one steers to the left (underfitting) side of Figure 32. Adding noise to training patterns is a very common strategy.

Adding noise while the optimization algorithm runs. If the optimization algorithm used for minimizing the empirical risk is iterative, one can “noisify” it by jittering the intermediate results with additive noise. We have not seen an iterative algorithm in this course so far, therefore I cannot demonstrate this by example here. The “dropout regularization” trick which is widely used in deep learning is of this kind. The effect is that the stronger the algorithm is noisified, the stronger the regularization, that is the further one steers to the left end of Figure 32.

Use stochastic ensemble learning. We have seen this strategy in the presentation of random forests. A stochastic learning algorithm is repeatedly executed with different random seeds (called Θ in Section 2.7). The stronger the randomness of the stochastic learning algorithm, and the more members are included in the ensemble, the stronger the regularization effect.

6.5 Finding the right modeling flexibility by cross-validation

Statistical learning theory has come up with a few analytical methods to approximately solve (43). But these methods are based on additional assumptions which are neither easy to verify nor always granted. By far the most widely used method to determine an (almost) optimal model flexibility (that is, determine the position of the green line in Figure 32) is a rather simple scheme called *cross-validation*. Cross-validation is a generic method which does not need analytical insight into the particulars of the learning task at hand. Its main disadvantage is that it may be computationally expensive.

Here is the basic idea of cross-validation.

In order to determine whether a given model is under- or overfitting, one would need to run it on test data that are “new” and not contained in the training data. This would allow one to get a hold on the red curve in Figure 32.

However, at training time only the training data are available.

The idea of cross-validation is to artificially split the training data set $D = (x_i, y_i)_{i=1, \dots, N}$ into two subsets $T = (x_i, y_i)_{i \in I}$ and $V = (x'_i, y'_i)_{i \in I'}$. These two subsets are then pretended to be a “training” and a “testing” dataset. In the context of cross-validation, the second set is called a *validation* set.

A bit more formally, let the flexibility axis in Figure 32 be parametrized by m , where small m means “strong regularization”, that is, “go left” on the flexibility axis. Let the range of m be $m = 1, \dots, L$.

For each setting of regularization strength m , the data in T is used to train an optimal model $h_{\text{opt } m}$. The test generalization performance on “new” data is then tested on the validation set, for each $m = 1, \dots, L$. It is determined which model $h_{\text{opt } m}$ performs best on the validation data. Its regularization strength m is then taken to be the sought solution m_{opt} to (43). After this screening of model classes for the best test performance, a model within the found optimal regularization strength is then finally trained on the original complete training data set D .

This whole procedure is called cross-validation. Notice that nothing has been said so far about how to split D into T and V . This is not a trivial question: how should D be best partitioned?

A clever way to answer this question is to split D into many subsets D_j of equal size ($j = 1, \dots, K$). Then carry out K complete screening runs via cross validation, where in the j -th run the subset D_j is withheld as a validation set, and the remaining $K - 1$ sets joined together make for a training set. After these K runs, average the validation errors in order to find m_{opt} . This is called *K-fold*

cross-validation. Here is the procedure in detail:

Given: A set $(x_i, y_i)_{i=1,\dots,N}$ of training data, and a loss function L .

Also given: Some method which allows one to steer the model flexibility along a regularization strength parameter m . The weakest regularization should be weak enough to allow overfitting.

Step 1. Split the training data into K disjoint subsets $D_j = (x_i, y_i)_{i \in I_j}$ of roughly equal size $N' = N/K$.

Step 2. Repeat for $m = 1, \dots, L$:

Step 2.1 Repeat for $j = 1, \dots, K$:

Step 2.2.1 Designate D_j as validation set V_j and the union of the other $D_{j'}$ as training set T_j .

Step 2.2.1 Compute the model with minimal training error on T_j

$$h_{\text{opt } m j} = \underset{h \in \mathcal{H}_m}{\operatorname{argmin}} 1/|T_j| \sum_{(x_i, y_i) \in T_j} L(h(x_i), y_i),$$

where \mathcal{H}_m is the model search space for the regularization strength m .

Step 2.2.2 Test $h_{\text{opt } m j}$ on the current validation set V_j by computing the validation risk

$$R_{m j}^{\text{val}} = 1/|V_j| \sum_{(x_i, y_i) \in V_j} L(h_{\text{opt } m j}(x_i), y_i).$$

Step 2.2 Average the K validation risks $R_{m j}^{\text{val}}$ obtained from the “folds” carried out for this m , obtaining

$$R_m^{\text{val}} = 1/K \sum_{j=1,\dots,K} R_{m j}^{\text{val}}.$$

Step 3. Find the optimal class by looking for that m which minimizes the averaged validation risk:

$$m_{\text{opt}} = \underset{m}{\operatorname{argmin}} R_m^{\text{val}}.$$

Step 4. Compute $h_{m_{\text{opt}}}$ using the complete original training data set:

$$h_{m_{\text{opt}}} = \underset{h \in \mathcal{H}_{m_{\text{opt}}}}{\operatorname{argmin}} 1/N \sum_{i=1,\dots,N} L(h(x_i), y_i).$$

This procedure contains two nested loops and looks expensive. For economy,

one starts with the low-end m and increases it stepwise, assessing the generalization quality through cross-validation for each regularization strength m , until the validation risk starts to rise. The strength m_{opt} reached at that point is likely to be about the right one.

The best assessment of the optimal class is achieved when the original training data set is split into singleton subsets – that is, each D_j contains just a single training example. This is called *leave-one-out cross-validation*. It looks like a horribly expensive procedure, but yet it may be advisable when one has only a small training data set, which incurs a particularly large danger of ending up with poorly generalizing models when a wrong model flexibility was used.

K -fold cross validation is widely used – it is a factual standard procedure in supervised learning tasks when the computational cost of learning a model is affordable.

6.6 Why it is called the bias-variance dilemma

We have seen that a careful adjustment of the flexibility of a supervised learning algorithm is needed to find the sweet spot between underfitting and overfitting. A more educated way to express this condition is to speak of the *bias-variance tradeoff*, also known as *bias-variance dilemma*. In this subsection I want to unravel the root cause of the under/overfitting phenomenon in a little more mathematical detail. We will find that it can be explained in terms of a bias and a variance term in the expected error of estimated models.

Again I start from a training data set $(x_i, y_i)_{i=1,\dots,N}$ drawn from a joint distribution $P_{X,Y}$, with $x_i \in \mathbb{R}^n, y_i \in \mathbb{R}$. Based on these training data I consider the learning task to find a decision function $h : \mathbb{R}^n \rightarrow \mathbb{R}$ which has a low quadratic risk

$$R(h) = E_{X,Y}[(h(X) - Y)^2], \quad (48)$$

where the notation $E_{X,Y}$ indicates that the expectation is taken with respect to the joint distribution of X and Y . We assume that we are using some fixed learning algorithm \mathcal{A} which, if it is given a training sample $(x_i, y_i)_{i=1,\dots,N}$, estimates a model \hat{h} . The learning algorithm \mathcal{A} can be anything, good or bad, clever or stupid, overfitting or underfitting; it may be close to perfect or just be always returning the same model without taking the training sample into account – we don't make any assumptions about it.

The next consideration leads us to the heart of the matter, but it is not trivial. In mathematical terms, the learning algorithm \mathcal{A} is just a *function* which takes a training sample $(x_i, y_i)_{i=1,\dots,N}$ as input and returns a model \hat{h} . Importantly, if we would run \mathcal{A} repeatedly, but using freshly sampled training data $(x_i, y_i)_{i=1,\dots,N}$ in each run, then the returned models \hat{h} would be varying from trial to trial – because the input samples $(x_i, y_i)_{i=1,\dots,N}$ are different in different trials. Applying these varying \hat{h} to some fixed pattern $x \in \mathbb{R}^n$, the resulting values $\hat{h}(x)$ would show a random behavior too. The distribution of these variable values $\hat{h}(x)$ is

a distribution over \mathbb{R} . This distribution is determined by the distribution $P_{X,Y}$ and the chosen learning algorithm \mathcal{A} and the training sample size, and a good statistician could give us a formal derivation of the distribution of $\hat{h}(x)$ from $P_{X,Y}$ and knowledge of \mathcal{A} , but we don't need that for our purposes here. The only insight that we need to take home at this point is that for a fixed x , $\hat{h}(x)$ is a random variable whose value is determined by the drawn training sample $(x_i, y_i)_{i=1,\dots,N}$, and which has an expectation which we write as $E_{\text{retrain}}[\hat{h}(x)]$ to indicate that the expectation is taken over all possible training runs with freshly drawn training data.

Understanding this point is the key to understanding the inner nature of under/overfitting.

If you feel that you have made friends with this $E_{\text{retrain}}[\hat{h}(x)]$ object, we can proceed. The rest is easy compared to this first conceptual clarification.

Without proof I note the following, intuitively plausible fact. Among *all* decision functions (from any candidate space \mathcal{H}), the quadratic risk (48) is minimized by the function

$$\Delta(x) = E_{Y|X=x}[Y|X=x], \quad (49)$$

that is, by the expectation of Y given x . This function $\Delta : \mathbb{R}^n \rightarrow \mathbb{R}, x \mapsto E[Y|X=x]$ is the gold standard for minimizing the quadratic risk; no learning algorithm can give a better result than this. Unfortunately, of course, Δ remains unknown because the underlying true distribution $P_{X,Y}$ cannot be exactly known.

Now fix some x and ask by how much $\hat{h}(x)$ deviates, on average and in the squared error sense, from the optimal value $\Delta(x)$. This expected squared error is

$$E_{\text{retrain}}[(\hat{h}(x) - \Delta(x))^2].$$

We can learn more about this error if we re-write $(\hat{h}(x) - \Delta(x))^2$ as follows:

$$\begin{aligned} (\hat{h}(x) - \Delta(x))^2 &= (\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)] + E_{\text{retrain}}[\hat{h}(x)] - \Delta(x))^2 \\ &= (\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)])^2 + (E_{\text{retrain}}[\hat{h}(x)] - \Delta(x))^2 \\ &\quad + 2(\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)])(E_{\text{retrain}}[\hat{h}(x)] - \Delta(x)). \end{aligned} \quad (50)$$

Now observe that

$$E_{\text{retrain}} \left[(\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)]) (E_{\text{retrain}}[\hat{h}(x)] - \Delta(x)) \right] = 0, \quad (51)$$

because the second factor $(E_{\text{retrain}}[\hat{h}(x)] - \Delta(x))$ is a constant, hence

$$\begin{aligned} E_{\text{retrain}} \left[(\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)]) (E_{\text{retrain}}[\hat{h}(x)] - \Delta(x)) \right] &= \\ &= E \left[\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)] \right] (E_{\text{retrain}}[\hat{h}(x)] - \Delta(x)), \end{aligned}$$

and

$$\begin{aligned} E_{\text{retrain}} \left[\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)] \right] &= E_{\text{retrain}}[\hat{h}(x)] - E_{\text{retrain}}[E_{\text{retrain}}[\hat{h}(x)]] \\ &= E_{\text{retrain}}[\hat{h}(x)] - E_{\text{retrain}}[\hat{h}(x)] \\ &= 0. \end{aligned}$$

Inserting (51) into (50) and taking the expectation on both sides of (50) of finally gives us

$$E_{\text{retrain}}[(\hat{h}(x) - \Delta(x))^2] = \underbrace{\left(E_{\text{retrain}}[\hat{h}(x)] - \Delta(x) \right)^2}_{\text{(squared) bias}} + \underbrace{E_{\text{retrain}} \left[\left(\hat{h}(x) - E_{\text{retrain}}[\hat{h}(x)] \right)^2 \right]}_{\text{variance}}. \quad (52)$$

The two components of this error are conventionally named the *bias* and the *variance* contribution to the expected squared mismatch $E_{\text{retrain}}[(\hat{h}(x) - \Delta(x))^2]$ between the learnt-model decision values $\hat{h}(x)$ and the optimal decision value $\Delta(x)$. The bias measures how strongly the average learning result deviates from the optimal value; thus it indicates a systematic error component. The variance measures how strongly the learning results $\hat{h}(x)$ vary around their expected value $E_{\text{retrain}}[\hat{h}(x)]$; this is an indication of how strongly the particular training data sets induce variations on the learning result.

When the model flexibility is too low (underfitting), the bias term dominates the expected modeling error; when the flexibility is too high (overfitting), the variance term is the main source of mismatch. This is why the underfitting versus overfitting challenge is also called the bias-variance tradeoff (or dilemma).

7 Representing and learning distributions

Almost all machine learning tasks are based on training data that have some random component. Having completely noise-free data from deterministic sources observed with high-precision measurements is a rare exception. Thus, machine learning algorithms are almost all designed to cope with stochastic data. Their ultimate functionality (classification, prediction, control, ...) will be served well or poorly to the extent that the probability distribution of the training data has been properly accounted for. We have seen this in the previous section. If one (wrongly) believes that there is little randomness in the training data, one will take the given training points as almost correct – and end up with an overfitted model. Conversely, if one (wrongly) thinks the training data are almost completely “white noise randomness”, the learnt model will under-exploit the information in the training data and underfit. Altogether it is fair to say that machine learning is the art of probability distribution modeling (plus subsequent use of that learnt distribution for classification etc.)

In many machine learning algorithms, the distribution model remains implicit – there is no place or data structure in the algorithm which explicitly models the data distribution. In some algorithms however, and for some tasks, the probability distribution of the training data is explicitly modeled. We will encounter some of these algorithms later in this course (hidden Markov models, Bayesian networks, Boltzmann machines are of that kind). At any rate, it is part of the standard knowledge of a machine learning professional to know some ways to represent probability distributions and to estimate these representations from data.

7.1 Optimal classification

The grand role which is played by modeling a data distribution accurately can be demonstrated very nicely in classification learning. We have met this basic task several times now – and in your professional future you will meet it again, dozens of times if you will seriously work in machine learning. We consider again a scenario where the input patterns are vectors $\mathbf{x} \in \mathbf{P} \subseteq \mathbb{R}^n$, which belong to one of k classes which we represent by their class labels $C = \{c_1, \dots, c_k\}$. The set \mathbf{P} is the “possible pattern space”, usually a bounded subset of \mathbb{R}^n (for instance, when the patterns \mathbf{x} are photographic pixel vectors and the pixel color values are normalized to a range between 0 and 1, \mathbf{P} would be the n -dimensional unit hypercube). We furthermore introduce a random variable (RV) X for the patterns and a RV Y for the class labels.

Many different decision functions $h : \mathbf{P} \rightarrow C$ exist, some of which can be learnt from training data using some known learning algorithm. We will not deal with the learning problem in this subsection, but ask (and answer!) the fundamental question:

Among all possible decision functions $h : \mathbf{P} \rightarrow C$, which one has the lowest risk in the sense of giving the lowest possible rate of misclassifications on test data?

It turns out that the minimal-risk decision function is in fact well-defined and unique, and it can (and must) be expressed in terms of the distribution of our data-generating RVs X and Y .

Our starting point is the true joint distribution $P_{X,Y}$ of patterns and labels. This joint distribution is given by all the probabilities of the kind

$$P(X \in A, Y = c), \quad (53)$$

where A is some subvolume of \mathbf{P} and $c \in \mathbf{C}$. The subvolumes A can be n -dimensional hypercubes within \mathbf{P} , but they also can be arbitrarily shaped “volume bodies”, for instance balls or donuts or whatever. Note that the probabilities $P(X \in A, Y = c)$ are numbers between 0 and 1, while the distribution $P_{X,Y}$ is the *function* which assigns to every choice of $A \subseteq \mathbf{P}, c \in C$ the number $P(X \in A, Y = c)$ (probability theory gives us a rigorous formal way to define and handle this strange object, $P_{X,Y}$, which I will explain in depth, step by step, in the tutorial.)

The joint distribution $P_{X,Y}$ is the “ground truth” – it is the real-world statistical distribution of pattern-label pairs of the kind we are interested in. In the Digits example, it would be the distribution of pairs made of (i) a handwritten digit and (ii) a human-expert provided class label. Test digit images and their class labels would be randomly “drawn” from this distribution.

A decision function $h : \mathbf{P} \rightarrow \{c_1, \dots, c_k\}$ partitions the pattern space \mathbf{P} into k disjoint *decision regions* R_1, \dots, R_k by

$$R_i = \{\mathbf{x} \in \mathbf{P} \mid h(\mathbf{x}) = c_i\}. \quad (54)$$

A test pattern \mathbf{x}^{test} is classified by h as class i if and only if it falls into the decision region R_i .

Now we are prepared to analyze and answer our ambitious question, namely which decision functions yield the lowest possible rate of misclassifications. Since two decision functions yield identical classifications if and only if their decision regions are the same, we will focus our attention on these regions and reformulate our question: which decision regions yield the lowest rate of misclassifications, or expressed in its mirror version, which decision regions give the highest probability of correct classifications?

Let f_i be the pdf for the conditional distribution $P_{X|Y=c_i}$. It is called the *class-conditional* distribution.

The probability to obtain a correct classification for a random test pattern, when the decision regions are R_i , is equal to $\sum_{i=1}^k P(X \in R_i, Y = c_i)$. Rewriting this expression using the pdfs of the class conditional distributions gives

$$\begin{aligned}
& \sum_{i=1}^k P(X \in R_i, Y = c_i) = \\
&= \sum_{i=1}^k P(X \in R_i | Y = c_i) P(Y = c_i) \\
&= \sum_{i=1}^k P(Y = c_i) \int_{R_i} f_i(\mathbf{x}) d\mathbf{x} \\
&= \sum_{i=1}^k \int_{R_i} P(Y = c_i) f_i(\mathbf{x}) d\mathbf{x}.
\end{aligned} \tag{55}$$

Note that the integral is taken over a region that possibly has curved boundaries, and the integration variable \mathbf{x} is a vector. The boundaries between the decision regions are called *decision boundaries*. For patterns \mathbf{x} that lie exactly on such boundaries, two or more classifications are equally probable. For instance, the digit pattern shown in the last but third column in the second row in Figure 15 would likely be classified by humans as a “1” or “4” class pattern with roughly the same probability; this pattern would lie close to a decision boundary.

The expression (55) obviously becomes maximal if the decision regions are given by

$$R_i = \{\mathbf{x} \in \mathbf{P} \mid i = \operatorname{argmax}_j P(Y = c_j) f_j(\mathbf{x})\}. \tag{56}$$

Thus we have found the decision function which is optimal in the sense that it maximizes the probability of correct classifications: namely

$$h_{\text{opt}} : \mathbf{P} \rightarrow C, \mathbf{x} \mapsto c_{\operatorname{argmax}_j P(Y=c_j) f_j(\mathbf{x})}. \tag{57}$$

A learning algorithm that finds the optimal decision function (or some function approximating it) *must* learn (implicitly or explicitly) estimates of the the class-conditional distributions $P_{X|Y=c_i}$ and the class probabilities $P(Y = c_i)$.

The class probabilities are also called the class *priors*. Figure 33 visualizes optimal decision regions and decision boundaries. In higher dimensions, the geometric shapes of decision regions can become exceedingly complex, fragmented and “folded into one another” — disentangling them during a learning process is one of the eternal challenges of ML.

7.2 Representing and learning distributions

The optimal classifier described in the previous subsection is optimal because it is shaped along the true distribution of pattern and class label random variables.

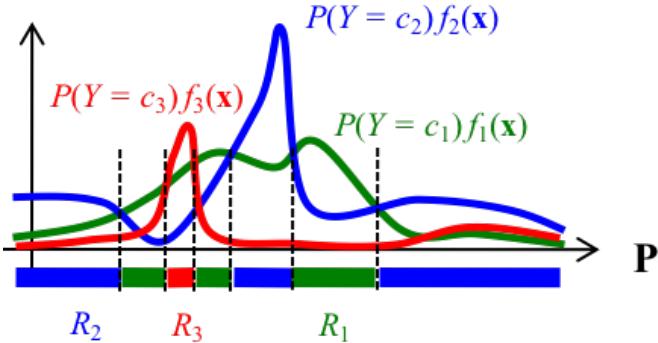


Figure 33: Optimal decision regions R_i . A case with a one-dimensional pattern space \mathbf{P} and $k = 3$ classes is shown. Broken lines indicate decision boundaries. Decision regions need not be connected!

Quite generally, *any* machine learning task can be solved “optimally” (in terms of minimizing some risk) only if the solution takes the true distribution of all task-relevant RVs into account. As I mentioned before, many learning algorithms estimate a model of the underlying data distribution only implicitly. But some ML algorithms generate explicit models of probability distributions, and in the wider fields of statistical modeling, explicit models of probability distributions are often the final modeling target.

Representing a probability distribution in mathematical formalism or by an algorithm is not always easy. Real-world probability distributions can be utterly complex and high-dimensional objects which one cannot just “write down” in a formula. Over the last 60 or so decades, ML and statistics research has developed a wide range of formalisms and algorithms for representing and estimating (“learning”) probability distributions. A machine learning professional should know about some basic kinds of such formalisms and algorithms. In this section I present a choice, ranging from elementary to supremely complex, powerful — and computationally costly.

7.2.1 Some classical probability distributions

In this section we describe a few distributions which arise commonly in application scenarios. They have standard names and one should just know them, and under which conditions they arise.

We start with distributions that are defined over discrete sample spaces $S = \{s_1, \dots, s_k\}$ (finite sample space) or $S = \{s_1, s_2, s_3, \dots\}$ (countable infinite sample space). These distributions can all be represented by their *probability mass function* (pmf):

Definition 7.1 Given a discrete sample space S (finite or countable infinite), a probability mass function on S is a function $p : S \rightarrow [0, 1]$ whose total mass is 1, that is, which satisfies $\sum_{s \in S} p(s) = 1$.

Bernoulli distribution. The Bernoulli distribution arises when one deals with observations that have only two possible outcomes, like tail – head, female – male, pass – fail, 0 – 1. This means a two-element sample space $S = \{s_1, s_2\}$ equipped with the power set σ -field, on which a Bernoulli distribution is defined by its pmf, which in this case has its own standard terminology:

Definition 7.2 The distribution of a random variable which takes values in $S = \{s_1, s_2\}$ with probability mass function

$$p(s_i) = \begin{cases} 1 - q & \text{for } i = 1 \\ q & \text{for } i = 2 \end{cases}$$

is called a Bernoulli distribution with success parameter q , where $0 \leq q \leq 1$.

A Bernoulli distribution is thus specified by a single parameter, q .
Learning / estimation: Given a set of training data $\{x_1, \dots, x_N\}$ where $x_i \in \{s_1, s_2\}$, the parameter q can be estimated by $\hat{q} = (1/N) |\{i | x_i = s_2\}|$.

Binomial distribution. The binomial distribution describes the counts of successes if a binary-outcome “Bernoulli Experiment” is repeated N times. For a simple example, consider a gamble where you toss a coin N times, and every time the head comes up, you earn a Dollar (not Euro; gambling is done in Las Vegas). What is the distribution of Dollar earnings from such N -repetition games, if the coin comes up with head ($= s_2$; outcome tail is s_1) with a success probability of q ? Clearly the range of possible earnings goes from 0 to N Dollars. These earnings are distributed according to the *binomial distribution*:

Definition 7.3 Let N be the number of trials of independent Bernoulli experiments with success probability q in each trial. The distribution of the number of successes is called the binomial distribution with parameters N and q and its pmf is given by

$$p(s) = \binom{N}{s} q^s (1 - q)^{N-s} = \frac{N!}{s!(N-s)!} q^s (1 - q)^{N-s}, \quad s = 0, 1, 2, \dots, N.$$

We write $Bi(N, q)$ to denote the binomial distribution with parameters N and q .

The factor $\binom{N}{s}$ is called *binomial coefficient*. Figure 34 shows some binomial pmf's.

A note on notation: it is customary to write

$$X \sim Bi(10, 0.25)$$

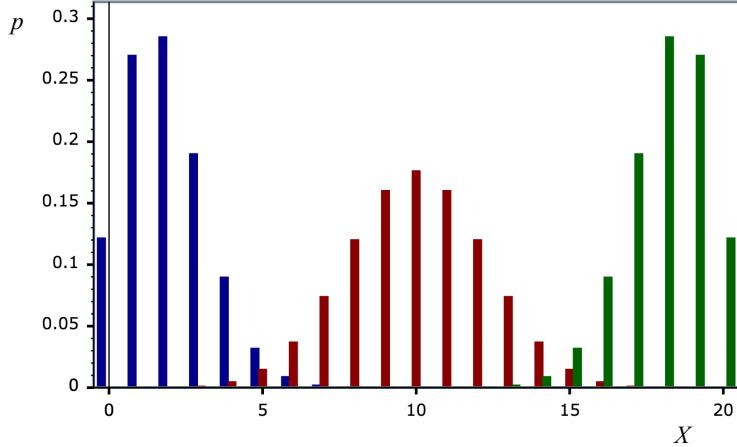


Figure 34: The pmf's of $Bi(20, 0.1)$ (blue), $Bi(20, 0.5)$ (red), and $Bi(20, 0.9)$ (green). Figure taken from www.boost.org.

as a shorthand for the statement “ X is distributed according to $Bi(10, 0.25)$ ”.

Learning / estimation: Depends on the format of available training data. In most cases the data will be of the same form as in the Bernoulli distribution, which gives rise to an estimate \hat{q} of the success parameter. The number N of repetitions will usually be given by the context in which the modeling task arises and need not be estimated.

Poisson distribution. This distribution is defined for $S = \{0, 1, 2, \dots\}$. $p(s)$ is the probability that a particular kind of event occurs s times within a given time interval. Examples (except the last one taken from https://en.wikipedia.org/wiki/Poisson_distribution): $p(s)$ might be the probability that

- s meteorites impact on the earth within 100 years,
- a call center receives s calls in an hour,
- a block of uranium emits s alpha particles in a second,
- s patients arrive in an emergency ward between 10 and 11 pm,
- a piece of brain tissue sends s neural spike signals within a second.

Similarly, instead of referring to a time interval, s may count spatially or otherwise circumscribed events, for instance

- the number of dust particles found in a milliliter of air,
- the number of diamonds found in a ton of ore.

The expected number of events $E[X]$ is called the *rate* of the Poisson distribution, and is commonly denoted by λ . The pmf of a Poisson distribution with rate λ is given by

$$p(s) = \frac{\lambda^s e^{-\lambda}}{s!}. \quad (58)$$

Figure 35 depicts the pmf's for three different rates.

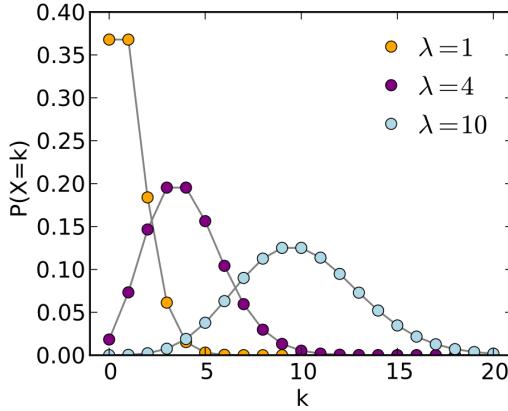


Figure 35: The pmf of the Poisson distribution for various values of the parameter λ . The connecting lines between the dots are drawn only for better visual appearance (image source: https://commons.wikimedia.org/wiki/File:Poisson_pmf.svg).

Learning / estimation: For concreteness consider the call center example. Training data would be N 1-hour protocols of incoming calls, where k_i ($i = 1, \dots, N$) is the number of calls received in the respective hour. Then λ is estimated by $\hat{\lambda} = (1/N) \sum_i k_i$.

We continue with a few distributions on continuous sample spaces (\mathbb{R}^n or subsets thereof). These distributions are most often characterized by their *probability density functions* (pdf's).

Definition 7.4 Let X be a RV which takes values in $S \subseteq \mathbb{R}^n$. A pdf for the distribution of X is a function $f : S \rightarrow \mathbb{R}^{\geq 0}$ which satisfies the following condition: For every subvolume $A \subseteq S$ of S , the probability $P(X \in A)$ that X gives a value in A is equal to

$$P(X \in A) = \int_A f(\mathbf{x}) d\mathbf{x}. \quad (59)$$

Point distributions. There exist continuous-valued distributions for which a description through a pdf is impossible. One kind of such “non-pdf-able” continuous distributions occurs quite often: *point distributions*. Here, the probability

mass is concentrated in a few (finitely many or countably infinitely many) points in \mathbb{R}^n . The simplest point distribution is defined on the real line $S = \mathbb{R}$ and has the defining property that for any subset $A \subseteq \mathbb{R}$ it holds that

$$P(X \in A) = \begin{cases} 1 & \text{if } 0 \in A \\ 0 & \text{if } 0 \notin A. \end{cases}$$

There are several ways to write down such a distribution in mathematical formalism. In the machine learning literature (and throughout the natural sciences, especially physics), the above point distribution would be represented by a weird kind of pdf-like function, called the *Dirac delta function* δ (the Wikipedia article https://en.wikipedia.org/wiki/Dirac_delta_function is recommendable if you want to understand this function better). The Dirac delta function is used inside an integral just like a normal pdf is used in (59). Thus, for a subset $A \subseteq \mathbb{R}$ one has

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

The Dirac delta is also used in \mathbb{R}^n , where likewise it is a “pdf” which describes a probability distribution concentrated in a single point, the origin.

If one wants to have a multi-point distribution one can combine Dirac deltas. For example, if you want to create a probability measure on the real line that places a probability of 1/2 on the point 1.0, and probabilities 1/4 each on the points 2.0 and 3.0, you can do this by a linear combination of shifted Dirac deltas:

$$P(X \in A) = \int_A 1/2 \delta(x - 1) + 1/4 \delta(x - 2) + 1/4 \delta(x - 3) dx.$$

Point distributions arise frequently in Bayesian machine learning, where they are needed to express and compute certain “hyperdistributions”. We will meet them later in this course.

We now take a look at a small choice of common continuous distributions which can be characterized by pdfs.

The uniform distribution. We don’t need to make a big fuzz about this. If $I = [a_1, b_1] \times \dots \times [a_n, b_n]$ is a n -dimensional interval in \mathbb{R}^n , the uniform distribution on I is given by the pdf

$$p(x) = \begin{cases} \frac{1}{(b_1 - a_1) \dots (b_n - a_n)} & \text{if } x \in I \\ 0 & \text{if } x \notin I. \end{cases} \quad (60)$$

Learning / estimation: This distribution is not normally “learnt”. There is also not in about it that could be learnt: the uniform distribution has no “shape” that would be specified by learnable parameters.

The exponential distribution. This distribution is defined for $S = [0, \infty)$ and could be paraphrased as “the distribution of waiting times until the next of these things happens”. Consider any of the kinds of temporal events listed for the Poisson distribution, for instance the event “meteorite hits earth”. The exponential distribution characterizes how long you have to wait for the next impact, given that one impact has just happened. Like in the Poisson distribution, such random event processes have an average *rate events / unit reference time interval*. For instance, meteorites of a certain minimum size hit the earth with a rate of 2.34 per year (just guessing). This rate is again denoted by λ . The pdf of the exponential distribution is given by

$$p(x) = \lambda e^{-\lambda x} \quad (\text{note that } x \geq 0). \quad (61)$$

It is (almost) self-explaining that the expectation of an exponential distribution is the reciprocal of the rate, $E(X) = 1/\lambda$. Figure 36 shows pdf’s for some rates λ .

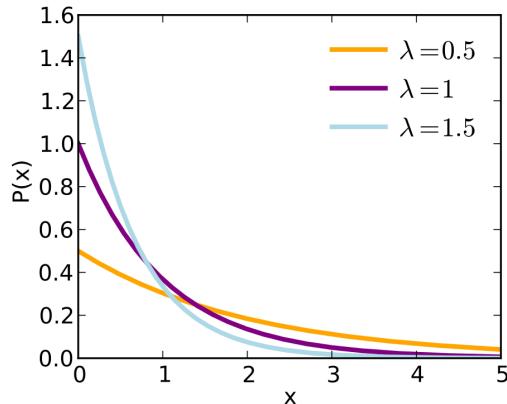


Figure 36: The pdf of the exponential distribution for various values of the parameter λ (image source: https://commons.wikimedia.org/wiki/File:Exponential_pdf.svg).

The exponential distribution plays a big role in spiking neural networks (SNNs). Biological neurons communicate with each other by sending short “point-like” electrical pulses, called *spikes*. Many people believe that Nature invented communication by spikes to let the brain save energy – your head shouldn’t perceptibly warm up (like your PC does) when you do a hefty thinking job! For the same reason, microchip engineers have teamed up with deep learning researchers to design novel kinds of microchips for deep learning applications. These microchips contain neuron-like processing elements which communicate with each other by spikes. IBM and Intel have actually built such chips (check out “IBM TrueNorth” and “Intel Loihi”) if you want learn more). Research about artificial SNNs for machine learning applications goes hand in hand with research in neuroscience.

In both domains, one often uses models based on the assumption that the temporal pattern in a spike sequence sent by a neuron is a stochastic process called a *Poisson process*. In a Poisson process, the waiting times between two consecutive spikes are exponentially distributed. Recordings from real biological neurons often show a temporal randomness of spikes that can be almost perfectly modeled by a Poisson process.

Learning / estimation: The rate λ is the only parameter characterizing an exponential distribution. Training data: a sequence t_1, t_2, \dots, t_N of time points where the event in question was observed. Then the rate can be estimated by $\hat{\lambda} = (1/(N-1)) \sum_{i=1, \dots, N-1} t_{i+1} - t_i$.

The one-dimensional normal distribution. Enter the queen of distributions! Bow in reverence! I am sure you know her from the media and facebook... For royal garments, as everybody knows, she wears the pdf

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad (62)$$

which is fully specified by its mean μ and standard deviation (square root of variance) σ , has the famous bell shape with μ being the location of the maximum and $\mu \pm \sigma$ being the locations of the zeros of the second derivative (Fig. 37).

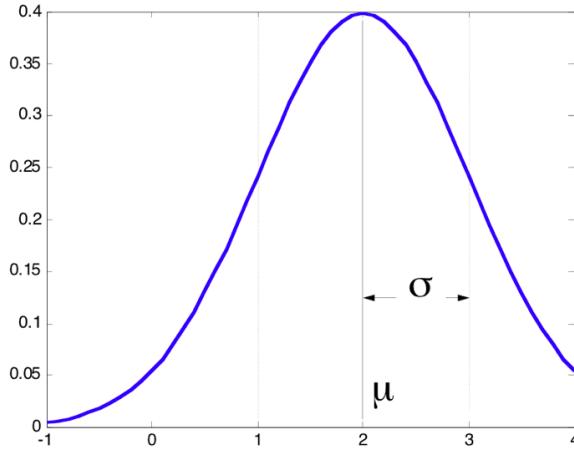


Figure 37: pdf of a normal distribution with mean 2 and standard deviation 1.

The normal distribution with mean μ and variance σ^2 is denoted by $\mathcal{N}(\mu, \sigma^2)$. The normal distribution with zero mean and unit variance, $\mathcal{N}(0, 1)$, is called the *standard normal distribution*. The normal distribution is also called *Gaussian distribution* or simply *Gaussian*.

The normal distribution has a number of nice properties that very much facilitate calculations and theory. Specifically, the sum of normal distributed, independent normal RVs is again normal distributed, and the variance of the sum is the sum of variances:

Proposition 7.1 *Let $X, Y : \Omega \rightarrow \mathbb{R}$ be two independent, normally distributed RVs with means μ and ν and variances σ^2 and τ^2 . Then the sum $X + Y$ is normally distributed with mean $\mu + \nu$ and variance $\sigma^2 + \tau^2$.*

The majestic power of the normal distribution, which makes her reign almost universally over almost all natural phenomena, comes from one of the most central theorems of probability theory, the *central limit theorem*. It is stated in textbooks in a variety of (not always exactly equivalent) versions. It says, in brief, that one gets the normal distribution whenever random effects of many independent small-sized causes sum up to large-scale observable effects. The following definition — which I do not expect you to fully understand (which would need a substantial training in probability theory), but you should at least have seen it — makes this precise:

Definition 7.5 *Let $(X_i)_{i \in \mathbb{N}}$ be a sequence of independent, real-valued, square integrable random variables with nonzero variances $\text{Var}(X_i) = E[(X_i - E[X_i])^2]$. Then we say that the central limit theorem holds for $(X_i)_{i \in \mathbb{N}}$ if the distributions P_{S_n} of the standardized sum variables*

$$S_n = \frac{\sum_{i=1}^n (X_i - E[X_i])}{\sigma(\sum_{i=1}^n X_i)} \quad (63)$$

converges weakly to $\mathcal{N}(0, 1)$.

Explanations:

- A real-valued random variable with pdf p is *square integrable* if its [uncentered] second moment, that is the integral $E[X^2] = \int_{\mathbb{R}} x^2 p(x) dx$ is finite.
- If $(P_n)_{n \in \mathbb{N}}$ is a sequence of distributions over \mathbb{R} , and P a distribution (all over the same measure space $(\mathbb{R}, \mathcal{B})$), then $(P_n)_{n \in \mathbb{N}}$ is said to *converge weakly* to P if

$$\lim_{n \rightarrow \infty} \int f(x) P_n(dx) = \int f(x) P(dx) \quad (64)$$

for all continuous, bounded functions $f : \mathbb{R} \rightarrow \mathbb{R}$. You will find the notation of these integrals unfamiliar, and indeed you see here cases of *Lebesgue integrals* — a far-reaching generalization of the *Riemann integrals* that you know. Lebesgue integrals can deal with a far greater range of functions than the Riemann integral. Mathematical probability theory is formulated exclusively with the Lebesgue integral. We cannot give an introduction to Lebesgue integration theory in this course. Therefore, simply ignore the precise meaning of “weak convergence” and take home that sequences of distributions are required to converge to a target distribution in some subtly defined way.

A sequence $(X_i)_{i \in \mathbb{N}}$ of random variables (or, equivalently, its associated sequence of distribution $(P_{X_i})_{i \in \mathbb{N}}$) obeys the central limit theorem under rather weak conditions – or in other words, for many such sequences the central limit theorem holds.

A simple, important class of (X_i) for which the central limit theorem holds is obtained when the X_i are identically distributed (and, of course, are independent, square integrable and have nonzero variance). Notice that regardless of the shape of the distribution of each X_i , the distribution of the normalized sums converge to $\mathcal{N}(0, 1)$!

The classical demonstration of the central limit theorem is the *Galton board*, named after Sir Francis Galton (1822–1911), an English multi-scientist. The idea is to let little balls (or beans, hence this device is sometimes called “bean machine”) trickle down a grid of obstacles which randomly deflect the ball left or right (Figure 38). It does not matter how, exactly, these deflections act — in the simplest case, the ball is just kicked right or left by one space grid unit with equal probability. The deeper the trickling grid, the closer will the resulting distribution be to a normal distribution. A nice video can be watched at https://www.youtube.com/watch?v=PM7z_03o_kk.

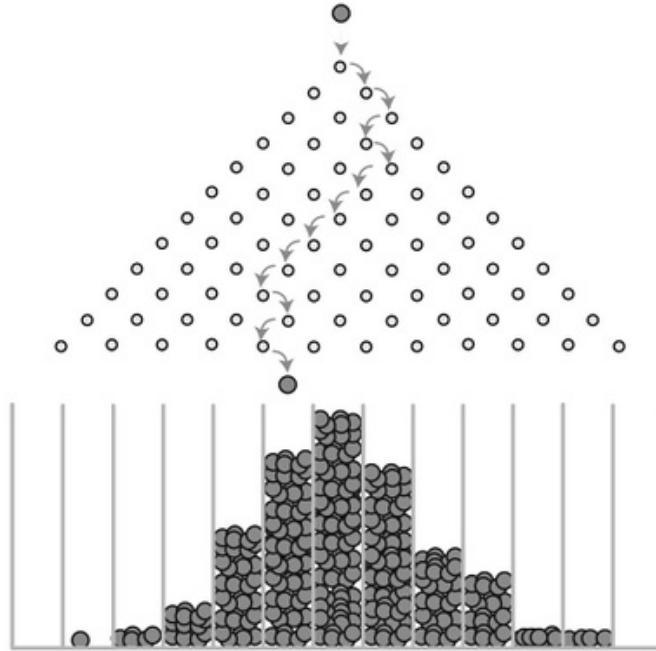


Figure 38: The Dalton board. Compare text for explanation. Figure taken from <https://janav.wordpress.com/2013/09/26/power-law/>.

However, this simple case does not explain the far-reaching, general importance of the central limit theorem (rather, property). In textbooks one often finds statements like, “if the outcomes of some measurement procedure can be conceived to

be the combined effect of many independent causal effects, then the outcomes will be approximately normal distributed". The "many independent causal effects" that are here referred to are the random variables (X_i); they will typically not be identically distributed at all. Still the central limit theorem holds under mild assumptions. Intuitively, all that one has to require is that none of the individual random variables X_i dominates all the others – the effects of any single X_i must asymptotically be "washed out" if an increasing number of other $X_{i'}$ is entered into the sum variable S_n . In mathematical textbooks on probability you may find numerous mathematical conditions which amount to this "washing out". A special case that captures many real-life cases is the condition that the X_i are uniformly bounded, that is, there exists some $b > 0$ such that $|X_i(\omega)| < b$ for all i and ω . However, there exist much more general (nontrivial to state) conditions that likewise imply the central limit theorem. For our purposes, a good enough take-home message is

if (X_i) is a halfway reasonably behaved sequence of numerical RV's, then the normalized sums converge to the standard normal distribution.

As we will see in Part 2 of this lecture, the normal distribution plays an overwhelming role in applied statistics. One often has to actually compute integrals of the pdf (62):

Task: compute the numerical value of $\int_a^b \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$.

There is no closed-form solution formula for this task. Instead, the solution is found in a two-step procedure:

1. Transform the problem from its original version $\mathcal{N}(\mu, \sigma^2)$ to the standard normal distribution $\mathcal{N}(0, 1)$, by using

$$\int_a^b \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \int_{\frac{a-\mu}{\sigma}}^{\frac{b-\mu}{\sigma}} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx. \quad (65)$$

In terms of probability theory, this means to transform the original, $\mathcal{N}(\mu, \sigma^2)$ -distributed RV X to a $\mathcal{N}(0, 1)$ -distributed variable $Z = (X - \mu)/\sigma$. (The symbol Z is often used in statistics for standard normal distributed RVs.).

2. Compute the numerical value of the r.h.s. in (65) by using the cumulative density function of $\mathcal{N}(0, 1)$, which is commonly denoted by Φ :

$$\int_{\frac{a-\mu}{\sigma}}^{\frac{b-\mu}{\sigma}} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx = \Phi\left(\frac{b-\mu}{\sigma}\right) - \Phi\left(\frac{a-\mu}{\sigma}\right).$$

Since there is no closed-form solution for calculating Φ , in former times statisticians found the solution in books where Φ was tabulated. Today, statistics software packages call fast iterative solvers for Φ .

Learning / estimation: The one-dimensional normal distribution is characterized by two parameters, μ and σ^2 . Given a dataset $(x_i)_{i=1,\dots,N}$ of points in \mathbb{R} , the estimation formulas for these two parameters are

$$\begin{aligned}\hat{\mu} &= \frac{1}{N} \sum_i x_i, \\ \hat{\sigma}^2 &= \frac{1}{N-1} \sum_i (x_i - \hat{\mu})^2.\end{aligned}$$

The n -dimensional normal distribution. If data points are not just real numbers but vectors $\mathbf{x} = (x_1, \dots, x_n)' \in \mathbb{R}^n$, whose component RVs X_i which give the vector components x_i fulfill the central limit theorem, the joint distribution of the RVs X_1, \dots, X_n is the multidimensional normal distribution which has the pdf

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{n/2} \det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})' \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right). \quad (66)$$

Here $\boldsymbol{\mu}$ is the expectation $E[(X_1, \dots, X_n)']$ and $\boldsymbol{\Sigma}$ is the covariance matrix of the n component variables, that is $\boldsymbol{\Sigma}(i, j) = E[(X_i - E[X_i])(X_j - E[X_j])]$. Figure 39 shows the pdf of a 2-dimensional normal distribution. In geometrical terms, a multidimensional normal distribution is shaped as an *ellipsoid*, whose main axes coincide with the eigenvectors \mathbf{u}_i of the covariance matrix $\boldsymbol{\Sigma}$. Like in PCA one can obtain them from the SVD: if $UDU' = \boldsymbol{\Sigma}$ is the singular value decomposition of $\boldsymbol{\Sigma}$, the eigenvectors \mathbf{u}_i are the columns of U .

Multidimensional normal distributions (or simply, “Gaussians”) are major players in machine learning. On the elementary end, one can use weighted combinations of Gaussians to approximate complex distributions - we will see this later in today’s lecture. At the advanced end, there is an entire modern branch of ML, *Gaussian processes*, where complex distributions (and hyperdistributions) are modeled by infinitely many, interacting Gaussians. This is quite beyond the scope of our course.

Learning / estimation: Given a set of n -dimensional training data points $(\mathbf{x}_i)_{i=1,\dots,N}$, the expectation $\boldsymbol{\mu}$ and the covariance matrix $\boldsymbol{\Sigma}$ can be estimated from the training data in the obvious way:

$$\hat{\boldsymbol{\mu}} = 1/N \sum_i \mathbf{x}_i \text{ and } \hat{\boldsymbol{\Sigma}} = 1/(N-1) \sum_i (\mathbf{x}_i - \hat{\boldsymbol{\mu}})(\mathbf{x}_i - \hat{\boldsymbol{\mu}})'$$

... and many more! The few common, named distributions that I displayed in this section are only meant to be illustrative picks from a much, much larger reservoir of well-known, completely analyzed, tabulated, pre-computed, and individually named distributions. The online book “Field Guide to Continuous Probability Distributions” by G. E. Crooks (2017) attempts a systematic overview. You should take home the following message:

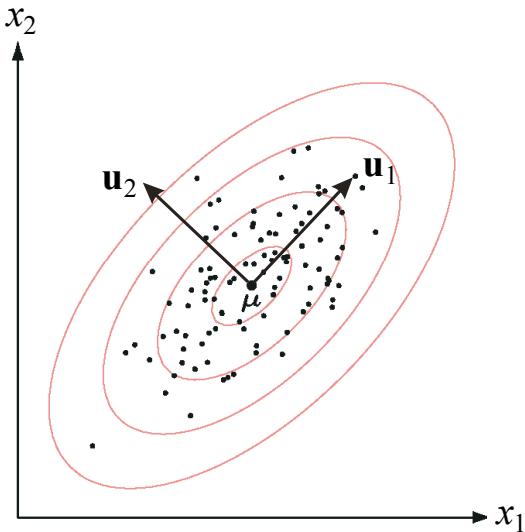


Figure 39: A pdf (indicated by contour lines) of a 2-dimensional normal distribution with expectation μ . The vectors $\mathbf{u}_1, \mathbf{u}_2$ are the principal axes of the ellipses which characterize the geometry of this pdf.

- In 100 years or so of research, statisticians have identified hundreds of basic mechanisms by which nature generates random observations. In this subsection we only looked at only two of them – (i) intermittent rare “impact events” coming from large numbers of independent sources which hit some target system with a mean frequency λ , giving rise to Poisson and exponential distributions; and (ii) stochastic physical measurables that can be understood as the additive effect of a large number of different causes, which leads to the normal distribution.
- One way of approaching a statistical modeling task for a target distribution P_X is to
 1. first analyze and identify the nature of the physical (or psychological or social or economical...) effects that give rise to this distributions,
 2. then do a literature search (e.g. check out what G. E. Crooks says) or ask a statistics expert friend which known and named distribution is available that was tailored to capture exactly these effects, – which will likely give you a distribution formula that is shaped by a small number of parameters θ ,
 3. then estimate θ from available observation data, getting a distribution estimate $\hat{\theta}$, and

4. use the theory that statisticians have developed in order to calculate confidence intervals (or similar accuracy tolerance measures) for $\hat{\theta}$, which
 5. finally allows you to state something like, “*given the observation data, with a probability of 0.95, the true distribution θ^{true} differs from the estimate $\hat{\theta}$ by less than 0.01 percent.*”
- In summary, a typical professional statistical modeling project starts from well-argued assumptions about the type of the true distribution, then estimates the parameters of the distribution, then reports the estimate together with a quantification of the error bound or confidence level (or the like) of the estimate.
 - Professionally documented statistical analyses will *always* state not only the estimated model, but also in addition quantify how accurate the model estimate is. This can take many forms, like error bars drawn around estimated parameters or stating “significance levels”.
 - If you read a report that only reports a model estimate, without any such quantification of accuracy levels, then this report has not been written by a professional statistician. There are two kinds of such reports. Either the author was ignorant about how to carry out a statistical analysis – then trash the report and forget it. Or the report was written by a machine learner in a task context where accuracy levels are not important and/or cannot be technically obtained because it is not possible to identify the distribution kind of the data generating system.

7.3 Mixture of Gaussians; maximum-likelihood estimates by EM algorithms

Approximating a distribution of points $\mathbf{x} \in \mathbb{R}^n$ by a single n -dimensional Gaussian will be a far too coarse approximation which wipes out most of the interesting structure in the data. Consider, for a demonstration, the probability distribution of points in a rectangular subset of \mathbb{R}^2 visualized in Figure 40. The true distribution shown in panel **A** could not be adequately approximated by a single 2-dimensional Gaussian, but if one invests a mixture of twenty of them, the resulting model distribution (panel **D**) starts to look useful.

Approximating a complex probability distribution by such *mixtures of Gaussians* (MoG’s) is a widely used technique. It comes with a transparent and robust optimization algorithm which locates, scales and orients the various Gaussians such that all together they give an optimal approximation. This optimization algorithm is an *expectation-maximization* (EM) algorithm. EM algorithms appear in many variants in machine learning, but all of them are designed according to the same basic optimization principle, which every machine learning professional

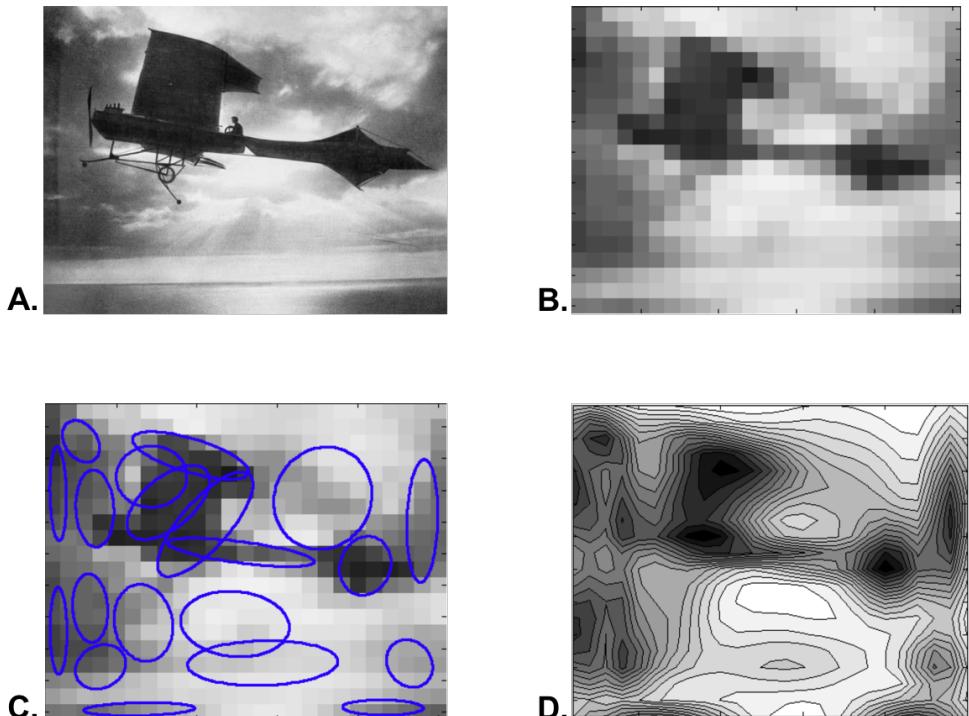


Figure 40: **A.** A richly structured 2-dimensional pdf (darker: larger values of pdf; white: zero value of pdf). **B.** Same, discretized to a pmf. **C.** A MoG coverage of the pmf with 20 Gaussians, showing the ellipsoids corresponding to standard deviations. **D.** Contour plot of the pdf represented by the mixture of Gaussians. (The picture I used here is an iconic photograph from the history of aviation, July 1909. It shows Latham departing from Calais, attempting to cross the Channel in an aeroplane for the first time in history. The motor of his Antoinette IV monoplane gave up shortly before he reached England and he had to water. A few days later his rival Bleriot succeeded. https://en.wikipedia.org/wiki/Hubert_Latham

should know. I will use this occasion of optimizing MoGs to introduce you to the world of EM algorithms as a side-effect.

Let us first fix notation for MoGs. We consider probability distributions in \mathbf{R}^n . A Gaussian (multi-dimensional normal distribution) in \mathbf{R}^n is characterized by its mean μ and its n -dimensional covariance matrix Σ . We lump all the parameters inside μ and Σ together and denote the resulting parameter vector by θ . We denote the pdf of the Gaussian with parameters θ by $p(\cdot|\theta)$. Now we can write down the pdf p of a MoG that is assembled from m different Gaussians as

$$p : \mathbf{R}^n \rightarrow \mathbf{R}^{\geq 0}, \quad \mathbf{x} \mapsto \sum_{j=1}^m p(\mathbf{x}|\theta_j) P(j), \quad (67)$$

where $\theta_j = (\mu_j, \Sigma_j)$ are the parameters of the j th Gaussian, and where the *mixture coefficients* $P(j)$ satisfy

$$0 \leq P(j) \quad \text{and} \quad \sum_{j=1}^m P(j) = 1.$$

For notational simplification we will mostly write $\sum_{j=1}^m p(\mathbf{x}|j) P(j)$ instead of $\sum_{j=1}^m p(\mathbf{x}|\theta_j) P(j)$.

MoG's are used in unsupervised learning situations: given a data set $(\mathbf{x}_i)_{i=1,\dots,N}$ of points in \mathbf{R}^n , one wants to find a “good” MoG approximation of the distribution from which the \mathbf{x}_i have been sampled. But... what does “good” mean?

This is the point to introduce the loss function which is invariably used when the task is to find a “good” distribution model for a set of training data points. Because this kind of loss function and this way of thinking about optimizing distribution models is not limited to MoG models but is the universal way of handling distribution modeling throughout machine learning, I devote a separate subsection to it and discuss it in general terms.

7.3.1 Maximum likelihood estimation of distributions

We consider the following general modeling problem: Given a dataset $D = (\mathbf{x}_i)_{i=1,\dots,N}$ of points in \mathbf{R}^n and a hypothesis set \mathcal{H} of pdf's p over \mathbf{R}^n , where each pdf $p \in \mathcal{H}$ is parametrized by θ_p , we want to select that $p_{\text{opt}} \in \mathcal{H}$ which “best” models the distribution from which the data points \mathbf{x}_i were sampled.

In supervised learning scenarios, we formalized this modeling task around a loss function which compared model outputs with teacher outputs. But now we find ourselves in an unsupervised learning situation. How can a pdf p be a “good” or “bad” model for a given set of training points?

In order to answer this question, one turns the question around: assuming the true distribution is given by a pdf p , how “likely” is it to get the training dataset $D = (\mathbf{x}_i)_{i=1,\dots,N}$? That is, how well can the training datapoints be “explained” by p ?

To make this precise, one considers the probability that the dataset D was drawn, *given* that the true distribution has the pdf p (which is characterized by parameters θ). Since the probability of drawing D is a probability over the set of N datapoints from \mathbb{R}^n , the probability of drawing D is described by a pdf $f(D|\theta)$ over $(\mathbb{R}^n)^N$. At this point one assumes that the datapoints in D have been drawn independently (and each of them from the distribution p), which leads to

$$f(D|\theta) = \prod_{i=1}^N p(\mathbf{x}_i | \theta). \quad (68)$$

This pdf $f(D|\theta)$ value measures the probability of getting the sample D if the underlying distribution is the one modeled by θ . Conversely, one says that this value $f(D|\theta)$ is the *likelihood* of the distribution θ given the data D , and writes $L(\theta | D)$. The words “probability” and “likelihood” are two different words which refer to the same mathematical object, namely $f(D|\theta)$, but this object is read from left to right when speaking of “probability”:

- “the probability to get data D given the distribution is modeled by θ ”, written $f(D|\theta)$,

and it is read from right to left when speaking of “likelihood”:

- “the likelihood of the model θ given a dataset D ”, written $L(\theta | D)$.

A correct usage of the words “probability” vs. “likelihood” is a sign of a good education in machine learning.

The likelihood pdf (68) is not suitable for doing numerical computations, because a product of very many small numbers (the $p(\mathbf{x}_i | \theta)$) will lead to numerical underflow and become squenched to zero on digital computers. To avoid this, one always works with the *log likelihood*

$$\mathcal{L}(\theta | D) = \log \prod_{i=1}^N p(\mathbf{x}_i | \theta) = \sum_{i=1}^N \log p(\mathbf{x}_i | \theta) \quad (69)$$

instead. A model θ is “good” if it has a high (log)likelihood. Machine learning algorithms for finding the best model p in a hypothesis set \mathcal{H} are designed to find the *maximum likelihood* model

$$\theta^{\text{ML}} = \underset{\theta \in \mathcal{H}}{\operatorname{argmax}} \mathcal{L}(\theta | D). \quad (70)$$

Machine learners like to think of their learning procedures as minimizing some cost or loss. Thus one also finds in the literature the sign-inverted version of (70)

$$\theta^{\text{ML}} = \underset{\theta \in \mathcal{H}}{\operatorname{argmin}} -\mathcal{L}(\theta | D), \quad (71)$$

which obviously is equivalent. If one writes out the $\mathcal{L}(\theta | D)$ one sees the structural similarity of this unsupervised learning task with the supervised loss-minimization task which we met earlier in this course (e.g. (41)):

$$\theta^{\text{ML}} = \underset{\theta \in \mathcal{H}}{\operatorname{argmin}} - \sum_{i=1}^N \log p(\mathbf{x}_i | \theta).$$

7.3.2 Maximum-likelihood estimation of MoG models by an EM algorithm

Back to our concrete business of modeling a distribution represented by a dataset $D = (\mathbf{x}_i)_{i=1,\dots,N}$ by a MoG. In order to find a maximum-likelihood solution for this MoG modeling task, one has to answer two questions:

1. What is the appropriate number m of mixture components? Setting m determines the class of candidate solutions in which an optimization algorithm can search for the maximum-likelihood solution. The set \mathcal{H}_m contains all mixtures of m Gaussians. We have here a case of a model class inclusion sequence as discussed in Section 6.4. For determining an appropriate model flexibility m the routine way would be to carry out a cross-validation.
2. If an appropriate m has been determined, how should the parameters θ_j (where $j = 1, \dots, m$) of the participating Gaussians be set in order to solve the optimization problem

$$\theta^{\text{ML}} = \underset{\theta \in \mathcal{H}_m}{\operatorname{argmax}} \mathcal{L}(\theta | D)? \quad (72)$$

The second question is answered by employing an expectation-maximization (EM) algorithm. EM algorithms can be often designed for maximizing the (log) likelihoods of parametrized distributions. There is a general scheme for designing such algorithms, and general theorems concerning their convergence properties. The EM principle had been introduced by Dempster et al. (1977) — one of these classical landmark papers that will forever continue to be continuously cited (as of 2019, 57K Google cites).

EM algorithms is widely used in computational statistics when it comes to estimate model parameters θ from *incomplete* or *partially observable* data. What “incomplete” means depends on the specific case. For instance, it might mean missing data points in time series measurements, or measurements of physical systems where the available sensors give only a partial account of the system state.

The principle of designing EM algorithms is a cornerstone of many important machine learning algorithms that are using “incomplete” training data. What “incomplete” means depends on the specific case. The general idea is that the observed data have been caused by some “hidden” mechanism which

- for filling in “best guesses” for missing values in time series data where observations are unavailable for certain time points (like handwritten texts with unreadable words in them);
- for estimating *hidden Markov models* (HMMs) of stochastic timeseries; HMMs are the most widely used models for stochastic processes with memory;
- for estimating models of complex systems that are described by many interacting random variables, some of which are not observable — an example from human-machine interfacing: modeling the decisions expected from a user assuming that these decisions are influenced by emotional states that are not directly observable; such models are called *Bayesian networks* or more generally, *graphical models*;
- for a host of other practical modeling tasks, like our MoG optimization task.

Because of the widespread usefulness of EM algorithms I will now carry you through a general, abstract explanation of the underlying principle. This is not easy stuff but worth the effort. I follow the modern exposition given in Roweis and Ghahramani (1999), which also covers other versions of EM for other machine learning problems. I use however another notation than these authors and I supply more detail. In blue font I will insert how the abstract picture translates into our specific MoG case. The following treatment will involve a lot of probability formalism — I consider it a super exercise in becoming friends with random variables and products of random variables! try not to give up on the way! I suggest that in a first reading you skip the blue parts and concentrate on the abstract picture alone.

The general situation is the following. Let X be an observable random variable and let Y be a hidden random variable, which take values in value sets (“sample spaces”) S_X and S_Y , respectively. X produces the visible training data D and Y produces the hidden data which have a causal impact on D but cannot be observed. Note: the words *visible* and *hidden* are the common technical terms in the EM literature.

In the MoG case, X would be the random variable which generates the entire training sample $(\mathbf{x}_i)_{i=1,\dots,N}$, thus here $S_X = (\mathbb{R}^n)^N$. Y would be a random variable that decides which of the m Gaussians is used to generate \mathbf{x}_i , thus $S_Y = \{1, \dots, m\}^N$. The RV Y is a product $Y = \bigotimes Y_i$ of i.i.d. RVs Y_i , where Y_i takes values $j \in \{1, \dots, m\}$. Concretely, Y_i selects one of the m Gaussians by a random choice weighted by the probability vector $(P(1), \dots, P(m))$. Thus the value of Y_i is j_i , an element of $\{1, \dots, m\}$.

The RV X is a product $X = \bigotimes X_i$ of i.i.d. RVs X_i , where X_i generates the i -th training data point \mathbf{x}_i by a random draw from the j_i -th Gaussian, that is from the normal distribution $\mathcal{N}(\mu_{j_i}, \Sigma_{j_i})$.

Figure 7.3.2 illustrates what it means not to know the values of the hidden variable Y .

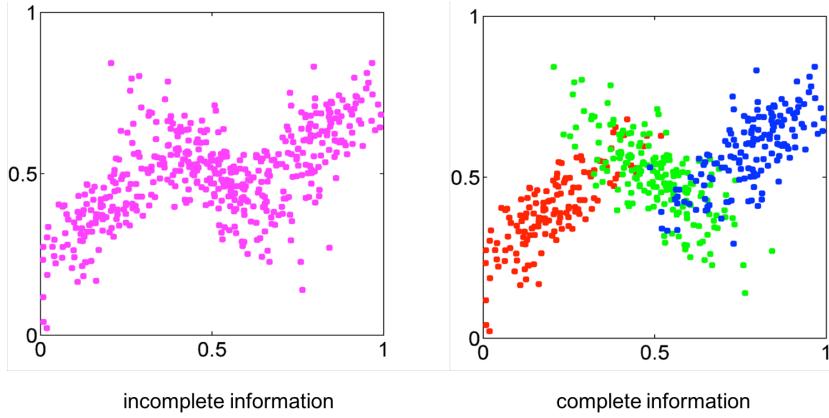


Figure 41: A point set sampled from a mixture of Gaussians. Left: what we (don't) know if we don't know from which Gaussian a point was drawn. Right: the “hidden” variable made visible. (Figure copied from an online presentation of Christopher M. Bishop, no longer accessible)

Let θ be a vector of parameters describing the joint distribution $P_{X,Y}$ of hidden and visible RVs.

In our MoG case, $\theta = (\mu_1, \dots, \mu_m, \Sigma_1, \dots, \Sigma_m, P(1), \dots, P(m))'$.

Let D be the observable sample, that is, the result from a random draw with X .

In the MoG scenario, $D = (\mathbf{x}_i)_{i=1,\dots,N}$.

The objective of an EM algorithm is to determine θ such that the likelihood $L(\theta) = P(X = D | \theta)$ becomes maximal, or equivalently, such that the log likelihood

$$\mathcal{L}(\theta) = \log P(X = D | \theta)$$

becomes maximal.

At this point we would have to start using different notations (pmf's versus pdf's) depending whether X and Y are discrete or continuous RVs. I will carry out the formal description only for the case that both X and Y are continuous, which means that their joint distribution can be described by a pdf $p_{X,Y}$ defined on $S_X \times S_Y$. The marginal distributions of X and Y are then given by parametrized pdf's $p_X(x|\theta) = \int_Y p_{X,Y}(x,y|\theta) dy$ and $p_Y(y|\theta) = \int_X p_{X,Y}(x,y|\theta) dx$. We can then switch to a description of distributions using only pdfs, that is, the distribution of samples D is described by the pdf $p_X(D|\theta)$.

In the case of all-continuous distributions, the log likelihood of θ becomes

$$\mathcal{L}(\theta) = \log p_X(D|\theta). \quad (73)$$

The MoG case has a mix of types: X gives values in the continuous space $(\mathbb{R}^n)^N$ and Y gives values in the discrete space $\{1, \dots, m\}^N$. This makes it necessary to use a mix of pdf's and pmf's when working out the MoG case in detail.

The difficulty we are facing is that the pdf value $p_X(D|\theta)$ depends on the values that the hidden variables take, that is,

$$\mathcal{L}(\theta) = \log p_X(D|\theta) = \log \int_{S_Y} p_{X,Y}(D, y | \theta) dy.$$

In the MoG case, the integral over S_Y becomes a sum over S_Y :

$$\begin{aligned} p_X(D|\theta) &= \sum_{(j_1, \dots, j_N) \in \{1, \dots, m\}^N} P(Y = (j_1, \dots, j_N)) p_{X|Y=(j_1, \dots, j_N)}(D) \\ &= \sum_{(j_1, \dots, j_N) \in \{1, \dots, m\}^N} \prod_{i=1, \dots, N} P(Y_i = j_i) p_{X_i|Y_i=j_i}(\mathbf{x}_i), \end{aligned}$$

where $p_{X|Y=(j_1, \dots, j_N)}$ is the pdf of the conditional distribution of X given that the hidden variables take the value (j_1, \dots, j_N) and $p_{X_i|Y_i=j_i}$ is the pdf of the conditional distribution of X_i given that the hidden variable Y_i takes the value j_i .

Now let q be *any* pdf for the hidden variables Y . Then we can obtain a lower bound on $\log p_X(D|\theta)$ by

$$\begin{aligned} \mathcal{L}(\theta) &= \log \int_{S_Y} p_{X,Y}(D, y | \theta) dy = \\ &= \log \int_{S_Y} q(y) \frac{p_{X,Y}(D, y | \theta)}{q(y)} dy \\ &\geq \int_{S_Y} q(y) \log \frac{p_{X,Y}(D, y | \theta)}{q(y)} dy \\ &= \int_{S_Y} q(y) \log p_{X,Y}(D, y | \theta) dy - \int_{S_Y} q(y) \log q(y) dy \quad (74) \\ &=: \mathcal{F}(q, \theta), \quad (75) \end{aligned}$$

where the inequality follows from a version of Jensen's inequality which states that

$$E[f \circ Y] \leq f(E[Y])$$

for any concave function f (the log is concave).

EM algorithms maximize the lower bound $\mathcal{F}(q, \theta)$ by alternatingly and iteratively maximizing $\mathcal{F}(q, \theta)$ first with respect to q , then with respect to θ , starting from an initial guess $q^{(0)}, \theta^{(0)}$ which is then updated by:

$$\text{Expectation step: } q^{(k+1)} = \underset{q}{\operatorname{argmax}} \mathcal{F}(q, \theta^{(k)}), \quad (76)$$

$$\text{Maximization step: } \theta^{(k+1)} = \underset{\theta}{\operatorname{argmax}} \mathcal{F}(q^{(k+1)}, \theta). \quad (77)$$

The maximum in the E-step is obtained when q is the conditional pdf of Y given the data D ,

$$q^{(k+1)} = p_{Y|X=D,\theta^{(k)}}, \quad (78)$$

because then $\mathcal{F}(q^{(k+1)}, \theta^{(k)}) = \mathcal{L}(\theta^{(k)})$:

$$\begin{aligned} \mathcal{F}(p_{Y|X=D,\theta^{(k)}}, \theta^{(k)}) &= \\ &= \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log p_{X,Y}(D, y | \theta^{(k)}) dy - \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log p_{Y|X=D,\theta^{(k)}}(y) dy \\ &= \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log \frac{p_{X,Y}(D, y | \theta^{(k)})}{p_{Y|X=D,\theta^{(k)}}(y)} dy \\ &= \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log p_X(D | \theta^{(k)}) dy \\ &= \int_{S_Y} \frac{p_{X,Y}(D, y | \theta^{(k)})}{p_X(D | \theta^{(k)})} \log p_X(D | \theta^{(k)}) dy \\ &= \frac{\log p_X(D | \theta^{(k)})}{p_X(D | \theta^{(k)})} \int_{S_Y} p_{X,Y}(D, y | \theta^{(k)}) dy \\ &= \log p_X(D | \theta^{(k)}) \\ &= \mathcal{L}(\theta^{(k)}). \end{aligned}$$

In concrete algorithms, the conditional distribution $p_{Y|X=D,\theta^{(k)}}$ is computed as the expectation of Y given data D and parameters $\theta^{(k)}$, which is why this step has its name, the **Expectation step**.

The maximum in the M-step is obtained if the first term in (74) is maximized, because the second term does not depend on $\theta^{(k)}$:

$$\begin{aligned} \theta^{(k+1)} &= \operatorname{argmax}_{\theta} \int_{S_Y} q^{(k+1)}(y) \log p_{X,Y}(D, y | \theta) dy \\ &= \operatorname{argmax}_{\theta} \int_{S_Y} p_{Y|X=D,\theta^{(k)}}(y) \log p_{X,Y}(D, y | \theta) dy \end{aligned} \quad (79)$$

(80)

How the M-step is concretely computed depends on the particular kind of model. Because we have $\mathcal{F} = \mathcal{L}(\theta^{(k)})$ before each M-step, and the E-step does not change $\theta^{(k)}$, and \mathcal{F} cannot decrease in an EM-double-step, the sequence $\mathcal{L}(\theta^{(0)}), \mathcal{L}(\theta^{(1)}), \dots$ monotonously grows toward a supremum. The iterations are stopped when $\mathcal{L}(\theta^{(k)}) = \mathcal{L}(\theta^{(k+1)})$, or more realistically, when a predefined number of iterations is reached or when the growth rate falls below a predetermined threshold. The last parameter set $\theta^{(k)}$ that was computed is taken as the outcome of the EM algorithm.

It must be emphasized that EM algorithms steer toward a local maximum of the likelihood. If started from another initial guess, another final parameter set may be found. Here is a summary of the EM principle:

E-step: Estimate the distribution $p_{Y|X=D,\theta^{(k)}}(y)$ of the hidden variables, given data D and the parameters $\theta^{(k)}$ of a preliminary model. This can be intuitively understood as inferring knowledge about the hidden variables, thereby *completing* the data.

M-step: Use the (preliminary) “knowledge” of the complete data (given by D for the visibles and by $p_{Y|X=D,\theta^{(k)}}(y)$ for the hidden variables) to compute a maximum likelihood model $\theta^{(k+1)}$.

There are other ways to compute maximum likelihood solutions in tasks that involve visible and hidden variables. Specifically, one can often invoke a gradient descent optimization. A big advantage of EM over gradient descent is that EM does not need a careful tuning of algorithm control parameters (like learning rates, an eternal bummer in gradient descent methods), simply because there are no tuning parameters. Furthermore, EM algorithms are typically numerically robust, which cannot be said about gradient descent algorithms.

Now let us put EM to practice for the MoG estimation. For better didactic transparency, I will restrict my treatment to a special case where we require all Gaussians to be spheric, that is, their covariance matrices are of the form $\Sigma = \sigma^2 I_n$ where σ^2 is the variance of the Gaussian in every direction and I_n is the n -dimensional identity matrix. The general case of mixtures of Gaussians composed of member Gaussians with arbitrary Σ is described in textbooks, for instance Duda et al. (2001). And anyway, you would probably use a ready-made online tool for MoG estimation... Here we go.

A MoG pdf with m spherical components is given by the vector of parameters $\theta = (\mu_1, \dots, \mu_m, \sigma_1^2, \dots, \sigma_m^2, P(1), \dots, P(m))'$. This gives the following concrete optimization problem:

$$\begin{aligned} \theta^{\text{ML}} &= \underset{\theta \in \mathcal{H}_m}{\text{argmax}} \mathcal{L}(\theta | D) \\ &= \underset{\theta \in \mathcal{H}_m}{\text{argmax}} \sum_{i=1}^N \log \left(\sum_{j=1}^m \frac{1}{(2\pi\sigma_j^2)^{n/2}} \exp \left(-\frac{\|\mathbf{x}_i - \mu_j\|^2}{2\sigma_j^2} \right) P(j) \right) \end{aligned} \quad (81)$$

Assume that we are after iteration k and want to estimate $\theta^{(k+1)}$. In the E-step we have to compute the conditional distribution of the hidden variable Y , given data D and the preliminary model

$$\theta^{(k)} = (\mu_1^{(k)}, \dots, \mu_m^{(k)}, \sigma_1^{2(k)}, \dots, \sigma_m^{2(k)}, P^{(k)}(1), \dots, P^{(k)}(m))'.$$

Unlike in the treatment that I gave for the general case, where we assumed Y to be continuous, now Y is discrete. Its conditional distribution is given by the

probabilities $P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)})$. These probabilities are given by

$$P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)}) = \frac{p^{(k)}(\mathbf{x}_i | Y_i = j_i) P^{(k)}(j_i)}{p_X^{(k)}(\mathbf{x}_i)}, \quad (82)$$

where $p^{(k)}(\mathbf{x}_i | Y_i = j_i)$ is the pdf of the j_i -th Gaussian with parameters $\mu_{j_i}^{(k)}, \sigma_{j_i}^{2(k)}$ and

$$p_X^{(k)}(\mathbf{x}_i) = \sum_{j=1}^m P^{(k)}(j) p^{(k)}(\mathbf{x}_i | Y_i = j).$$

In the M-step we have to find maximum likelihood estimates for all parameters in θ . I do not give a derivation here but just report the results, which are intuitive enough:

$$\begin{aligned} \mu_j^{(k+1)} &= \frac{\sum_{i=1}^N P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)}) \mathbf{x}_i}{\sum_{i=1}^N P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)})}, \\ \sigma_j^{2(k+1)} &= \frac{1}{n} \frac{\sum_{i=1}^N P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)}) \|\mu_j^{(k+1)} - \mathbf{x}_i\|^2}{\sum_{i=1}^N P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)})}, \quad \text{and} \\ P^{(k+1)}(j) &= \frac{1}{N} \sum_{i=1}^N P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)}). \end{aligned}$$

I conclude this sudorific (look this up in a dictionary) subsection with a little EM-for-MoG demo that I copied from an online presentation of Christopher Bishop (now apparently no longer online). The sample data $\mathbf{x}_i \in \mathbb{R}^2$ come from observations of the Old Faithful geyser in the Yellowstone National Park (Figure 42).

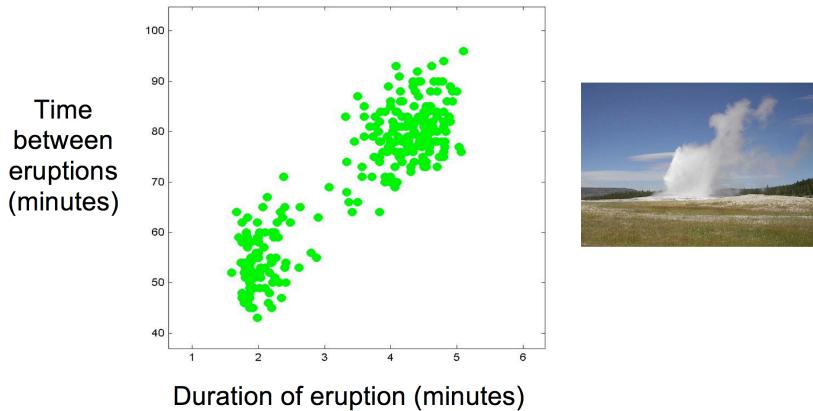


Figure 42: A two-dimensional dataset.

Figure 43 shows 20 EM iterations with $m = 2$ Gaussian components. The first panel shows the initial guess. Color codes are the estimated probabilities

$P(Y_i = j_i | X_i = \mathbf{x}_i, \theta^{(k)})$. The MoG modeling of the aeroplane image in Figure 40 was also done with the EM algorithm, in a version that allowed for arbitrary covariance matrices Σ_j . I did it myself, programming it from scratch, just to prove to myself that all of this really works. It does!

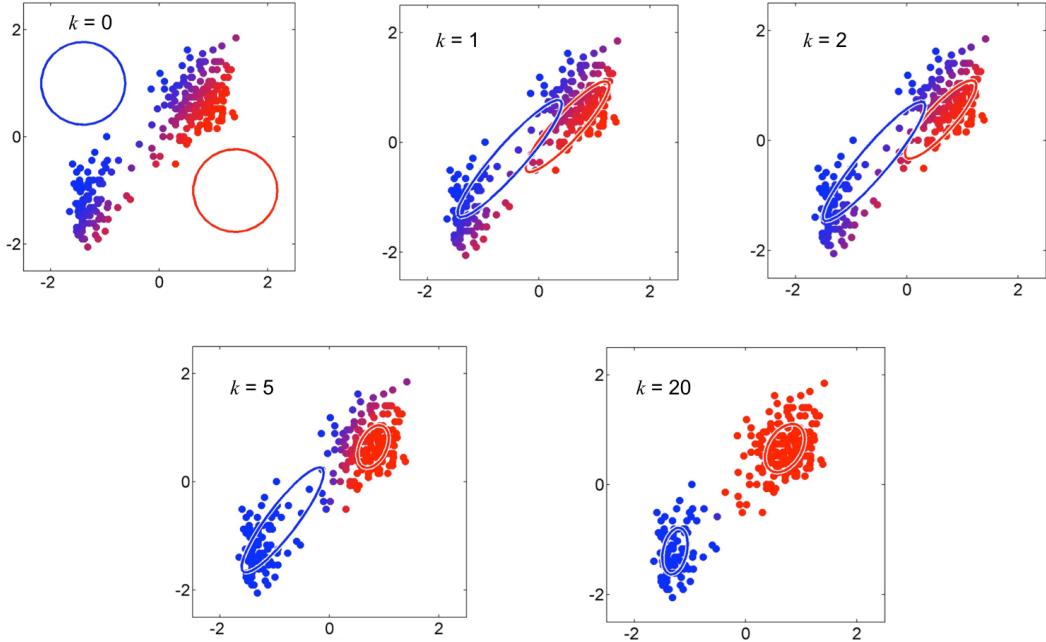


Figure 43: The EM algorithm at work on the Old Faithful dataset.

7.4 Parzen windows

After the heavy-duty work of coming to grips with the EM algorithm, let us finish this section with something easy, for chilling out. Parzen windows!

Parzen windows are a simple representative of the larger class of *kernel-based* representations, provide an alternative to mixture models for representing pdf's. These representations are *non-parametric* — there is no parameter vector θ for specifying a Parzen window approximation to a pdf. To introduce Parzen windows, consider a sample of 5 real-valued points shown in Figure 44. Centered at each sample point we place a unit square area on the x-axis. Weighing them each by $1/5$ and summing them gives an intuitively plausible representation of a pdf.

To make this example more formal and general, consider n -dimensional data points \mathbf{x}_i . Instead of a unit-length square, we wish to make these points the centers of n -dimensional hypercubes of side length d . We need a function H that indicates which points around \mathbf{x}_i fall into the hypercube centered at \mathbf{x}_i . To this end we introduce a *kernel function*, also known as *Parzen window*,

$$H : \mathbb{R}^n \rightarrow \mathbb{R}^{\geq 0}, \quad (x_1, \dots, x_n)' \mapsto \begin{cases} 1, & \text{if } |x_j| < 1/2 \text{ for } j = 1, \dots, n \\ 0, & \text{else} \end{cases} \quad (83)$$

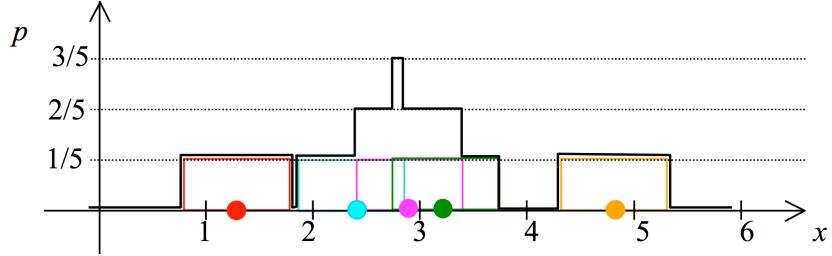


Figure 44: Rectangular Parzen window representation of a distribution given by a sample of 5 real numbers. The sample points are marked by colored circles. Each data point lies in the middle of a square "Parzen window", that is, a rectangular pdf centered on the point. Weighted by 1/5 (colored rectangles) and summed (solid black staircase line) they give a pdf.

which makes H the indicator function of a unit hypercube centered at the origin. Using H , we get the n -dimensional analog of the staircase pdf in Figure 44 for a sample $D = (\mathbf{x}_i)_{i=1,\dots,N}$ by

$$p^{(D)}(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^N \frac{1}{d^n} H\left(\frac{\mathbf{x} - \mathbf{x}_i}{d}\right), \quad (84)$$

observing that the volume of such a cube is d^n . The superscript (D) in $p^{(D)}$ is meant to indicate that the pdf depends on the sample D .

Clearly, given some sample $(\mathbf{x}_i)_{i=1,\dots,N}$, we do not believe that such a rugged staircase reflects the true probability distribution the sample was drawn from. We would rather prefer a smoother version. This can be easily done if we use smoother kernel functions. A standard choice is to use multivariate Gaussians with diagonal covariance matrix and uniform standard deviations $\sigma =: d$ for H . This turns (84) into

$$\begin{aligned} p^{(D)}(\mathbf{x}) &= \frac{1}{N} \sum_{i=1}^N \frac{1}{(2\pi d^2)^{n/2}} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2d^2}\right) \\ &= \frac{1}{N} \sum_{i=1}^N \frac{1}{d^n} \frac{1}{(2\pi)^{n/2}} \exp\left(-\frac{1}{2} \frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{d^2}\right) \end{aligned} \quad (85)$$

where the second line brings the expression to a format that is analog to (84).

It is clear that any nonnegative kernel function H which integrates to unity can be used in an equation of this sort such that the resulting $p^{(D)}$ will be a pdf. The scaling factor d determines the width of the Parzen window and thereby the amount of smoothing. Figure 45 illustrates the effect of varying d .

Comments:

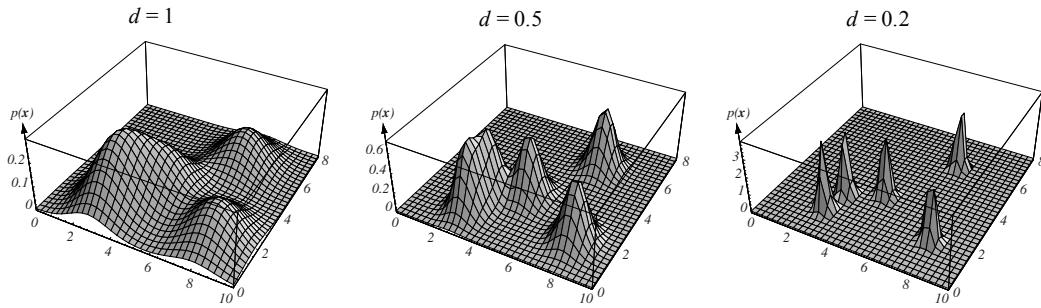


Figure 45: The effect of choosing different widths d in representing a 5-point, 2-dimensional sample by Gaussian windows. (Taken from the online set of figures of the book by Duda, Hart & Stork, ftp://ftp.wiley.com/public/sci_tech_med/pattern/)

- Parzen window representations of pdfs are "non-parametric" in the sense that the shape of such a pdf is determined by a sample (plus, of course, by the shape of the kernel function, which however mainly serves a smoothing purpose). This fact also can render Parzen window representations computationally expensive, because if the sample size is large, a large number of data points have to be stored (and accessed if the pdf is going to be used).
- The basic Parzen windowing scheme, as introduced here, can be refined in many ways. A natural way to improve on it is to use different widths d for different sample points \mathbf{x}_i . One then makes d narrow in regions of the sample set which are densely populated, and wide in regions that are only thinly covered by sample points. One way of doing that (which I invented while I was writing this — there are many ways to go) would be to (i) choose a reasonably small integer K ; (ii) for each sample point \mathbf{x}_i determine its K nearest neighbors $\mathbf{x}_{i1}, \dots, \mathbf{x}_{iK}$; (iii) compute the mean squared distance δ of \mathbf{x}_i from these neighbors, (iv) set d proportional to this δ for this sample point \mathbf{x}_i .
- As Figure 45 demonstrates, the width d has a strong effect on the Parzen pdf. If d is too large, the smoothing becomes too strong, and information contained in the sample is smoothed away — underfitting! In contrast, when d is too small, all that we see in the resulting Parzen pdf is the individual data points — the pdf then models not a distribution, but just the sample. Overfitting! Here one should employ a cross-validation scheme to optimize d , minimizing the validation loss defined by $-\sum_{\mathbf{v} \in V} \log p^{(T)}(\mathbf{v} | d)$, where V is the validation set, T the training set, and $p^{(T)}(\mathbf{v} | d)$ the Parzen pdf obtained from the training set using width d .
- The Parzen-window based distribution (85) will easily lead to numerical

underflow problems for arguments \mathbf{x} which are not positioned close to a sample point, especially if d is set to small values. A partial solution is to use log probabilities instead, i.e. use

$$\log p^{(D)}(\mathbf{x}) = \log \left(\frac{1}{N} \sum_{i=1}^N \frac{1}{(2\pi d^2)^{n/2}} \exp \left(-\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{2d^2} \right) \right). \quad (86)$$

Still this will not usually solve all underflow problems, because the sum-exp terms within the log still may underflow. Here is a trick to circumvent this problem, known as the "log-sum-exp" trick. Exploit the following:

$$\begin{aligned} \log(\exp(-A) + \exp(-B)) &= \\ &= \log (\exp(-A + C) \exp(-C) + \exp(-B + C) \exp(-C)) \\ &= -C + \log (\exp(-A + C) + \exp(-B + C)), \end{aligned}$$

where you use $C = \max(A, B)$.

8 Bayesian model estimation

Machine learning is based on probability, and probability is ... ? — We don't know what probability is! After centuries of thinking, philosophers and mathematicians have not arrived to a general consensus of how to best understand randomness. The proposals can be broadly grouped into *objectivistic* and *subjectivistic* interpretations of "probability".

According to the objectivistic view, probability resides physically in the real world — it is a phenomenon of nature, as fundamentally a property of physical reality as for instance "time" or "energy".

According to the subjectivistic view, probability describes an observer's subjective opinion on something observed — a degree of belief, of uncertainty, of plausibility of judgement, or missing information etc.

Calling the two views "objectivistic" versus "subjectivistic" is the philosophers' terminology. Statisticians and machine learners rather call it *frequentist statistics* versus *Bayesian statistics*.

The duality of understandings of "probability" has left a strong mark on machine learning. The two ways of thinking about probability lead to two different ways of designing learning algorithms. Both are important and both are in daily use. A machine learning professional should be aware of the fundamental intuitions behind the two sorts of algorithms and when to use which sort. In this section I explain the fundamental ideas behind the two understandings of probability, outline the general principles of constructing Bayesian algorithms, and demonstrate them in a case study of great importance in bioinformatics, namely the classification of proteins.

8.1 The ideas behind frequentist statistics

When probability is regarded as a phenomenon of nature, there should be ways to *measure* it. The standard proposition of how one can measure probability is by *relative frequency counting*. For instance, in the textbook example of a loaded (unfair) die, a loaded die may have the physical property that $P(X = 6) = 1/5$ (as opposed to $P(X = 6) = 1/6$ for a fair die), where X is the RV which gives the outcomes of throwing-the-die experiments. This property of the die could be experimentally measured by *repeating* the die-throwing act many times. This would give a sequence of outcomes x_1, x_2, x_3, \dots where $x_i \in \{1, \dots, 6\}$. After N such throws, an estimate of the quantity $P(X = 6)$ is calculated by

$$\hat{P}_N(X = 6) = \frac{\text{number of outcomes } x_i = 6}{N}, \quad (87)$$

where N is the number of throws that the experimentalist measuring this probability has carried out.

We generally use the hat symbol $\hat{\cdot}$ on top of some variable to denote a numerical estimate based on a finite amount of observation data. The true probability $P(X =$

$6) = 1/5$ would then become “measurable in principle” by

$$(*) \quad P(X = 6) = \lim_{N \rightarrow \infty} \hat{P}_N(X = 6). \quad (88)$$

A fundamental mathematical theorem in frequentist probability theory — actually, *the* fundamental law which justifies frequentist probability theory in the first place — is the *law of large numbers*. It states that if one carries out an infinite sequence of repeating the same numerical measurement, obtaining a sequence of measurement values x_1, x_2, \dots , where $x_i \in \mathbb{R}$, then the mean value $\mu_N = (1/N) \sum_{i=1}^N x_i$ of initial sequences up N will “almost always” converge to the same number $\mu = E[X]$, called the expectation of X . Stating and proving this theorem in formal rigor has taken mathematicians a long time and was achieved by Kolmogorov not earlier than in the early 1930’s. A precise formulation needs the full apparatus of measure-theoretic probability theory (which I partly introduce in the tutorial sessions).

More generally, considering some random variable X which takes values in a sample space S , one considers subsets $A \subseteq S$ (called *events*) and then defines

The frequentist view of probability: The probability $P(X \in A)$ that the outcome of sampling a datapoint with procedure X gives an outcome in A is the relative frequency (in the limit of carrying out infinitely many trials of enacting a measurement procedure X) of obtaining a measurement value in A .

If one looks at this “definition” with a critical mind one will find that it is loaden with difficulties.

First, it defines a “measurement” process that is not feasible in reality because one cannot physically carry out infinitely many measurement trials. This is maybe not really disturbing because *any* measurement in the sciences (say, of a voltage) is imprecise and one gets measurements of increasing precision by repeating the measurement, just as when one measures a probability.

Second, it does not inform us about how, exactly, the “repeated trials” of executing X should be done in order to be “unbiased”. What does that mean in terms of experimental procedures? This is a very critical issue. To appreciate its impact, consider the example of a bank who wishes to estimate the probability that a customer will fail to pay back a loan. In order to get an estimate of this probability, the bank can only use customer data collected in the *past*, but wants to base creditworthyness decisions for *future* customers on those data. Picking only past data points to base probability estimates on hardly can qualify as an “absolutely unbiased” sampling of data, and in fact the bank may grossly miscalculate credit risks when

the general customer body or their economical conditions change over time. These difficulties have of course been recognized in practical applications of statistics. Textbooks and statistics courses for psychologists and economists contain entire chapters with instructions on how to create “random samples” or “unbiased samples” as well as possible.

Third, if one repeats the repeated measurement, say by carrying out one measurement sequence giving x_1, x_2, \dots and then (or in parallel) another one giving x'_1, x'_2, \dots , the values \hat{P}_N from Equation (88) are bound to differ between the two series. The limit indicated in Equation (88) must somehow be robust against different versions of the \hat{P}_N . Mathematical probability theory offers several ways to rigorously define limits of series of probability quantities which we do not present here. Equation (88) is suggestive only and I marked it with a (*) to indicate that it is not technically complete.

Among these three difficulties, only the second one is really problematic. The first one is just a warning that in order to measure a probability with increasing precision we need to invest an increasingly large effort, — but that is the same for other measurables in the sciences. The third difficulty can be fully solved by a careful definition of suitable limit concepts in mathematical probability theory. But the second difficulty is fundamental and raises its ugly head whenever statistical assertions about reality are made.

In spite of these difficulties, the objectivist view on probability in general and the frequentist account of how to measure it in particular is widely shared among empirical scientists. It is also the view of probability which is commonly taught in courses of statistics for mathematicians. A student of mathematics may not even have heard of Bayesian statistics at the end of his/her studies. In machine learning, the frequentist view leads to learning algorithms which are based on the principle of maximum likelihood estimation of distributions — as we have seen in Section 7.3.1. In fact, all what I wrote in these lecture notes up to now was implicitly based on a frequentist understanding of probability.

8.2 The ideas behind Bayesian statistics

Subjectivist conceptions of probability also have led to mathematical formalisms that can be used in statistical modeling. A hurdle for the student here is that a number of different formalisms exist which reflect different modeling goals and approaches, and tutorial texts are rare.

The common starting point for subjectivist theories of probability is to cast “probability” as a subjective degree of belief, of certainty of judgement, or plausibility of assertions, or similar — instead of as an objective property of real-world systems. Subjectivist theories of probability do not develop analytical tools to describe randomness in the world. Instead they provide formalisms that code how rational agents (you, I, and AI systems) should *think about* the world, in the face of

various kinds of uncertainty in their knowledge and judgements. The formalisms developed by subjectivists can by and large be seen as generalizations of classical *logic*. Classical logic only knows two truth values: true or false. In subjectivist versions of logic formalisms, a proposition can be assigned graded degrees of “belief”, “plausibility”, etc. For a very first impression, contrast a classical-logic syllogism like

$$\begin{array}{c} \textit{if } A \textit{ is true, then } B \textit{ is true} \\ A \textit{ is true} \\ \hline \\ \textit{therefore, } B \textit{ is true} \end{array}$$

with a “plausibility reasoning rule” like

$$\begin{array}{c} \textit{if } A \textit{ is true, then } B \textit{ becomes more plausible} \\ B \textit{ is true} \\ \hline \\ \textit{therefore, } A \textit{ becomes more plausible} \end{array}$$

This example is taken from Jaynes (2003, first partial online editions in the late 1990ies), where a situation is described in which a policeman sees a masked man running away from a jeweler’s shop whose window was just smashed in. The plausibility rule captures the policeman’s inference that the runner is a thief (A) because if a person is a thief, it is more likely that the person will run away from a smashed-in shop window (B) than when the person isn’t a thief. From starting points like this, a number of logic formalisms have been devised which enrich/modify classical two-valued logic in various ways. If you want to explore these areas a little further, the Wikipedia articles probabilistic logic, Dempster-Shafer theory, fuzzy logic, or Bayesian probability are good entry points. In some of these formalisms the Kolmogorov axioms of frequentist probability reappear as part of the respective mathematical apparatus. Applications of such formalisms arise in artificial intelligence (modeling reasoning under uncertainty), human-machine interfaces (supporting discourse generation), game theory and elsewhere.

The discipline of statistics has almost entirely been developed in an objectivist spirit, firmly rooted in the frequentist interpretation of probability. Machine learning also in large parts roots in this view. However, a certain subset of machine learning models and computational procedures have a subjectivist component. These techniques are referred to as *Bayesian model estimation* methods. Bayesian modeling is particularly effective and important when training datasets are small. I will explain the principle of Bayesian model estimation with a super-simple synthetic example. A more realistic but also more complex example will be given in a separate subsection.

Consider the following statistical modeling task. A measurement procedure yields real-valued outcomes (like measuring the speed of a diving falcon). It is repeated N times, giving a measurement sequence x_1, \dots, x_N . The i th measurement value is obtained from a RV X_i . These RVs X_i which model the individual measurements are i.i.d. We assume that the distribution of each X_i can be represented by a pdf $p_{X_i} : \mathbb{R} \rightarrow \mathbb{R}^{\geq 0}$. The i.i.d. property of the family $(X_i)_{i=1,\dots,N}$ implies that the N -dimensional pdf $p_{\bigotimes X_i} : \mathbb{R}^N \rightarrow \mathbb{R}^{\geq 0}$ for the distribution of the product RV $\bigotimes_i X_i$ can be written as

$$p_{\bigotimes X_i}((x_1, \dots, x_N)) = p_{X_1}(x_1) \cdot \dots \cdot p_{X_N}(x_N). \quad (89)$$

or in another notation (observing that all pdfs p_{X_i} are identical, so we introduce a generic RV X with $p_X = p_{X_i}$ for all i), as

$$p_{\bigotimes_i X}((x_1, \dots, x_N)) = \prod_i p_X(x_i). \quad (90)$$

For concreteness let us consider a case where $N = 2$, that is two observations (*only* two!) have been collected, forming a training sample $D = (x_1, x_2) = (0.9, 1.0)$.

The distribution P_X , represented by the pdf p_X , is unknown. The modeling task is to infer from the observed data D how the distribution P_X looks like, that is, how its pdf p_X is shaped. That looks like a hugely underdetermined problem: one cannot learn much about a potentially complicated pdf from knowing only $N = 2$ data points. So one resorts to a more modest subtask: infer from the sample what is the expectation $E[X]$ of P_X :

$$E[X] =: \mu = \int_{-\infty}^{\infty} x p_X(x) dx. \quad (91)$$

The classical frequentist answer to this question is to estimate the true expectation μ by the sample mean, that is to compute

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i, \quad (92)$$

which in our example gives $\hat{\mu} = (0.9 + 1.0)/2 = 0.95$.

This is the best a classical-frequentist modeler can do. In a certain well-defined sense which we will not investigate, the sample mean is the optimal estimator for the true mean of a real-valued distribution. But “best” is not “good”: with only two data points, this estimate is quite shaky. It has a high variance: if one would repeat the observation experiment, getting a new sample x'_1, x'_2 , very likely one would obtain a quite different sample and thus an estimate $\hat{\mu}'$ that is quite different from $\hat{\mu}$.

Bayesian model estimation shows a way how to do better. It is a systematic method to make use of *prior knowledge* that the modeler may have beforehand.

This prior knowledge takes the form of assumptions concerning the nature of the true distribution P_X . In our super-simple example let us assume that the modeler knows or believes that

1. the true distribution P_X is a normal distribution with unit standard deviation, that is, the pdf has the form $p_X(x) = 1/\sqrt{2\pi} \exp(-(x - \mu)^2/2)$, and
2. the true expectation μ can't be negative and it can't be greater than 1.

This kind of prior knowledge is often available. It includes knowledge or assumptions concerning the *functional type* of the true distribution (here: it is assumed to be normal distributed with unit standard deviation). Abstracting from our example, this kind of knowledge means to fix a *parametric family* of distributions as candidate solutions to the modeling task. In our example, the family is given by the family of pdfs $(1/\sqrt{2\pi} \exp(-(x - \mu)^2/2))_{\mu \in \mathbb{R}}$, which happens to be parametrized by a single parameter $\mu \in [0, 1]$ only – it is a one-dimensional family. In most real-world tasks one faces multi-parameter families, where each candidate distribution from the family is specified by some K -dimensional parameter vector $\theta = (\theta_1, \dots, \theta_K)'$. One then writes $p_X(\theta)$ to highlight the fact that the pdf p_X comes from a parametric family with parameters θ .

Another component of the prior knowledge is given by assumptions concerning the parameters θ of the parametric family $(p_{X_i}(\theta))_{\theta \in \mathbb{R}^K}$. In our mini-example the modeler felt confident to restrict the possible range of the single parameter $\theta_1 = \mu$ to the interval $[0, 1]$.

To start the Bayesian model estimation machinery, available prior knowledge about parameters θ must be cast into the form of a distribution over parameter space. For a K -parametric pdf p_X , the parameter space is \mathbb{R}^K . In our little example, $K = 1$ and we need to rephrase the priorly known constraint $0 \leq \mu \leq 1$ into a distribution over \mathbb{R} . Not knowing anything more detailed than $0 \leq \mu \leq 1$, the natural, most non-committing distribution is the uniform distribution over the interval $[0, 1]$. This distribution has the rectangular pdf $h(\mu)$ shown in Figure 46. Two comments:

- The distribution for model parameters θ is not a distribution in the classical sense (sketched in Appendix B). It is not connected to a random variable and does not model a real-world outcome of observations. Instead it captures subjective beliefs that the modeler has about how the true distribution P_{X_i} of data points *should* look like. It is here that subjectivistic aspects of “probability” intrude into an otherwise classical-frequentist picture.
- Each parameter vector $\theta \in \mathbb{R}^K$ corresponds to one specific pdf $p_X(\theta)$, which in turn represents one possible candidate distribution \hat{P}_X for empirical observation values x_i . A distribution over parameter vectors $\theta \in \mathbb{R}^K$ is thus a distribution over distributions. It is called a *hyperdistribution*.

According to (90), the pdf $p_X(\theta)$ gives rise to a sample distribution with pdf $p_{\otimes_i X}(\theta) : \mathbb{R}^N \rightarrow \mathbb{R}^{\geq 0}$ on the N -dimensional value space of data samples by simply multiplying the N values $p_X(\theta)(x_i)$. We write $p_{\otimes_i X}(D | \theta)$ to denote the pdf value $p_{\otimes_i X}(\theta)((x_1, \dots, x_N)) = p_{\otimes_i X}(\theta)(D)$ of $p_{\otimes_i X}(\theta)$ on a particular training data sample D .

Summarizing:

- The pdf $h(\theta)$ encodes the modeler's prior beliefs about how the parametrized distribution $p_X(\theta)$ should look like. Parameters θ where $h(\theta)$ is large correspond to data distributions that the modeler a priori finds more plausible. The distribution represented by $h(\theta)$ is a hyperdistribution and it is often called the (*Bayesian*) *prior*.
- If θ is fixed, $p_{\otimes_i X}(D | \theta)$ can be seen as a function of data vectors D . This function is a pdf over the training sample data space. For each possible training sample $D = (x_1, \dots, x_N)$ it describes how probable this particular outcome is, assuming the true distribution of X is $p_X(\theta)$.
- If, conversely, D is fixed, then $p_{\otimes_i X}(D | \theta)$ can be seen as a function of θ . Seen as a function of θ , $p_{\otimes_i X}(D | \theta)$ is *not* something like a pdf over θ -space. Its integral over θ will not usually be one. Seen as a function of θ , $p_{\otimes_i X}(D | \theta)$ is called a *likelihood function* — given data D , it reveals certain models θ as being more likely than others. A model (characterized by θ) “explains” given data D better if $p_{\otimes_i X}(D | \theta)$ is higher.

We thus have two sources of information about the sought-after, unknown true distribution $p_X(\theta)$: the likelihood $p_{\otimes_i X}(D | \theta)$ of θ given data, and the prior plausibility encoded in $h(\theta)$. These two sources of information are independent of each other: the prior plausibility is settled by the modeler *before* data have been observed, and should not be informed by data. Because the two sources of information come from “independent” sources of information (belief and data), it makes sense to combine them by multiplication and consider the product

$$p_{\otimes_i X}(D | \theta) h(\theta).$$

This product combines the two available sources of information about the sought-after true distribution $p_X(\theta)$. When data D are given, this product is a function of model candidates θ . High values of this product mean that a candidate model θ is a good estimate, low values mean it's bad — in the light of both observed data and prior assumptions.

With fixed D , the product $p_{\otimes_i X}(D | \theta) h(\theta)$ is a non-negative function on the K -dimensional parameter space $\theta \in \mathbb{R}^K$. It will not in general integrate to unity and thus is not a pdf. Dividing this product by its integral however gives a pdf, which we denote by $h(\theta | D)$:

$$h(\theta | D) = \frac{p_{\otimes_i X}(D | \theta) h(\theta)}{\int_{\mathbb{R}^K} p_{\otimes_i X}(D | \theta) h(\theta) d\theta} \quad (93)$$

The distribution on model parameter space represented by the pdf $h(\theta|D)$ is called the *posterior distribution* or simply the *posterior*. The formula (93) shows how Bayesians combine the subjective prior $h(\theta)$ with empirical information $p_{\otimes_i X}(D|\theta)$ to get a posterior distribution over candidate models. Comments:

- The posterior distribution $h(\theta|D)$ is the final result of a Bayesian model estimation procedure. It is a *probability distribution over candidate models*, which is a richer and often a more useful thing than the single model that is the result of a classical frequentist model estimation (like the sample mean from Equation 92).
- Here I have considered real-valued distributions that can be represented by pdfs throughout. If some of the concerned distributions are discrete or cannot be represented by pdfs for some reason, one gets different versions of (93).
- If one wishes to obtain a single, definite model estimate from a Bayesian modeling exercise, a typical procedure is to compute the mean value of the posterior. The resulting model $\hat{\theta}$ is called the *posterior mean estimate*

$$\hat{\theta} = \int_{\mathbb{R}^K} \theta h(\theta|D) d\theta.$$

- Compared to classical-frequentist model estimation, generally Bayesian model estimation procedures are computationally more expensive and also more difficult to design properly, because one has to invest some thinking into good priors. With diligently chosen priors, Bayesian model estimates may give far better models than classical-frequentist ones, especially when sample sizes are small.
- If one abbreviates the normalization term $\int_{\mathbb{R}^K} p_{\otimes_i X}(D|\theta) h(\theta) d\theta$ in (93) as $p(D)$ (“probability density of seeing data D , averaged over candidate model parameters θ ”), the formula (93) looks like a version of Bayes’ rule:

$$h(\theta|D) = \frac{p_{\otimes_i X}(D|\theta) h(\theta)}{p(D)}, \quad (94)$$

which is why this entire approach is called “Bayesian”. Note that while Bayes’ rule is a *theorem* that can be proven from the axioms of classical probability theory, (94) is a *definition* (of $h(\theta|D)$).

- A note on terminology. There exists a family of models in machine learning named *Bayesian networks*. This name does not imply that Bayesian networks are estimated from data using “subjectivist” Bayesian statistics. Bayesian networks are parametrized models and their parameter vectors can be estimated with classical frequentist or with Bayesian methods.

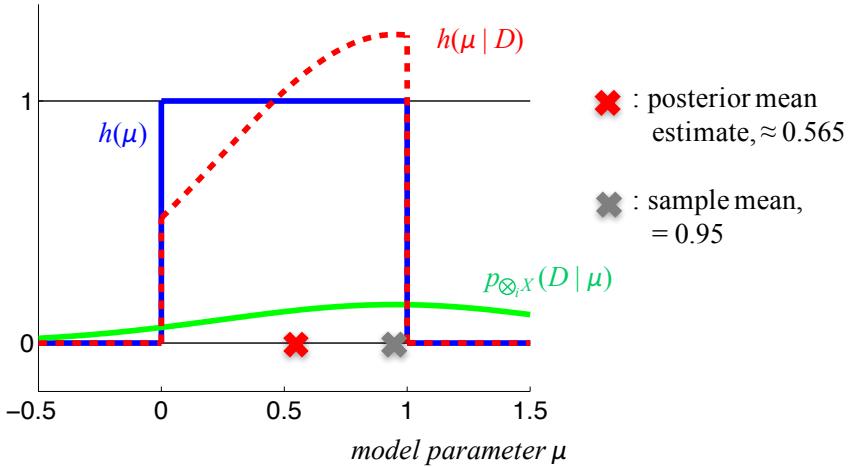


Figure 46: Bayesian model estimation. Compare text.

Let us conclude this section with a workout of our simple demo example. For the given sample $D = (0.9, 1.0)$, the likelihood function becomes

$$\begin{aligned} p_{\otimes_i X}(D | \mu) &= \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-(0.9-\mu)^2}{2}\right) \cdot \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-(1.0-\mu)^2}{2}\right) \\ &= \frac{1}{2\pi} \exp(-0.905 + 1.9\mu - \mu^2) \end{aligned}$$

The green line in Figure 46 gives a plot of $p_{\otimes_i X}(D | \mu)$, and the red broken line a plot of the posterior distribution $h(\mu | D)$. A numerical integration of $\int_{\mathbb{R}} \mu h(\mu | D) d\mu$ yields a posterior mean estimate of $\hat{\theta} \approx 0.565$. This is quite different from the sample mean 0.95, revealing the strong influence of the prior distribution on possible models.

8.3 Case study: modeling proteins

I will now discuss in some detail an elementary modeling task that is easily stated and ubiquitously met in practical applications — yet it needs a surprisingly subtle treatment if training data are scarce.

In an general formulation, the task is to estimate a probability mass function for a finite, discrete distribution, given a histogram from a sample.

For example, one might want to estimate the probabilities that a coin comes up with head or tail after tossing (formally modeled by a RV X). Tossing the coin 100 times gives a histogram over the two symbolic measurement outcomes H and T, say, (55, 45). Based on these training data, one wants to estimate the probabilities $P(X = H)$ and $P(X = T)$. A maximum likelihood estimate is the pmf $\hat{p} = (P(X = H), P(X = T))' = (0.55, 0.45)'$. Easy and obvious.

But things become not-so-easy and non-obvious when the number of symbolic categories is large and the number of available observations is small. I will highlight this with an example taken from a textbook of bioinformatics (Durbin et al., 2000).

Proteins are sequences of amino acids, of which there are 20 different ones which occur in proteins. They are standardly tagged by 20 capital symbols, giving an alphabet $S = \{A, C, D, \dots, Y\}$, intimately familiar to biologists. Proteins come in classes. Some protein in one animal or plant species has typically close relatives in other species. Related proteins differ in detail but generally can be *aligned*, that is, corresponding sites in the amino acid string can be detected, put side by side, and compared. For instance, consider the following short section of an alignment of 7 amino acids, all from the class of “globuline” proteins:

```
...GHGK...
...AHGK...
...KHGV...
...THAN...
...WHAE...
...AHAG...
...ALGA...
```

A basic task in bioinformatics is to estimate the probability distribution of the amino acids in each column in a protein class. These 20-dimensional pmf's (one for each site in the protein) are needed for deciding whether some newly found protein belongs into the class — that is, for carrying out protein classifications, a fundamental and bulk-volume task in bioinformatics.

This task is rendered difficult by the fact that the sample of aligned proteins used to estimate these pmf's is typically rather small — here we have only 7 individuals in the sample. As can be seen in the segment of the alignment shown above, some columns have widely varying entries (e.g. the last column has K K V N E G A). In such a small sample, chances are high that an amino acid, which is generally rare at a given site in the animal/plant kingdom, will not appear in the training data set. The column K K V N E G A only features 6 out of 20 possible amino acids; 14 amino acids are missing at this site in the training data.

But the class of related proteins is huge: in every animal species one would expect at least one class member and typically many more — millions of members in the family of globulines! This implies that at every site, every amino acid will occur in *some* species, though maybe with low probability. A low-probability-in-a-specific-site amino acid, call it X, will often not be represented in that site's training data. Consequences: (1) a maximum-likelihood estimate of the pmf for that site will assign zero probability to X which in turn implies (2), when this model is used for protein classification and a new test protein *does* have X in that site, it will turn out that the test protein cannot possibly be a globuline because it has X at that site and in globulines the probability for this is zero. This wrong

decision will be made for all test proteins which are globulines but happen to host one of the 14 amino acids not in the training set on that site. Since there are many sites in a protein (dozens to hundreds), chances are high for any test sequence to have a zero-probability amino-acid in some site. Most of the test sequences which are, in fact, globulines, would be rejected.

Thus, maximum likelihood estimates of pmf's for sites of proteins are unusable. A Bayesian approach is mandatory.

The following lengthy derivations show how the general formula (94) for Bayesian model estimation is worked out in the case of amino acid distribution estimation.

The 20-dimensional pmf for the amino acid distribution at a given site is characterized by a 20-dimensional parameter vector $\theta = (P(X = \text{A}, \dots, P(X = \text{Y}))'$ (actually 19 parameters would be enough — why?).

We start with the probability of observed data, given that the true distribution is θ . This corresponds to the pdf factor $p_{\bigotimes_i X}(D | \theta)$ in (94), but here we have discrete counting data and hence use a probability instead of a pdf. The observation data D consist in counts n_1, \dots, n_{20} of the different amino acids found at a given site in the training population. These count vectors are distributed according to the *multinomial distribution*

$$P(D | \theta) = \frac{20!}{n_1! \cdots n_{20}!} \prod_{j=1}^{20} \theta_j^{n_j}. \quad (95)$$

Next we turn to the Bayesian prior $h(\theta)$ in (94). The “subjective” background knowledge which is inserted into the picture is the general belief held by biologists that every amino acid *can* appear in any site in a protein of a given class, though maybe with small probability.

The Bayesian prior which reflects the biologists' insight that zero probabilities should not occur in θ is expressed in a hyperdistribution over the space Θ of possible θ vectors. Θ is the subset of \mathbf{R}^{20} which contains all positive vectors whose components sum to one:

$$\Theta = \{(\theta_1, \dots, \theta_{20})' \in \mathbf{R}^{20} \mid \theta_j \in (0, 1) \text{ and } \sum_j \theta_j = 1\}.$$

This is a 19-dimensional hypervolume in \mathbf{R}^{20} (try to imagine its shape — impossible for \mathbf{R}^{20} , but you can do it for embedding spaces $\mathbf{R}^2, \mathbf{R}^3, \mathbf{R}^4$).

Θ is a continuous space, thus a Bayesian prior distribution on Θ can be represented by a pdf. But which distribution makes a reasonable prior? For reasons that will become clear later, one uses the *Dirichlet* distribution. The Dirichlet distribution is parametrized itself, with (in our case) a 20-dimensional parameter vector $\alpha = (\alpha_1, \dots, \alpha_{20})'$ (where all α_i are positive reals). The Bayesian prior pdf $h(\theta|\alpha)$ of the Dirichlet distribution with parameters α is defined as

$$h(\theta|\alpha) = \frac{1}{Z(\alpha)} \prod_{j=1}^{20} \theta_j^{\alpha_j - 1}, \quad (96)$$

where $Z(\alpha) = \int_{\Theta} \prod_{j=1}^{20} \theta_j^{\alpha_j - 1} d\theta$ is the normalization constant which ensures that the integral of h over Θ is one.

Finally, the normalization denominator $p(D)$ in (94) need not be analyzed in more detail because it will later cancel out.

Now we have everything together to calculate the Bayesian posterior distribution on Θ :

$$\begin{aligned} p(\theta | D, \alpha) &= \frac{P(D | \theta) h(\theta | \alpha)}{p(D)} \\ &= \frac{1}{p(D)} \frac{20!}{n_1! \cdots n_{20}!} \prod_{j=1}^{20} \theta_j^{n_j} \frac{1}{Z(\alpha)} \prod_{j=1}^{20} \theta_j^{\alpha_j - 1} \\ &= \frac{1}{p(D)} \frac{20!}{n_1! \cdots n_{20}!} \frac{1}{Z(\alpha)} \prod_{j=1}^{20} \theta_j^{n_j + \alpha_j - 1} \\ &= \frac{1}{p(D)} \frac{20!}{n_1! \cdots n_{20}!} \frac{Z(D + \alpha)}{Z(\alpha)} h(\theta | D + \alpha), \end{aligned} \quad (97)$$

where $D + \alpha = (n_1 + \alpha_1, \dots, n_{20} + \alpha_{20})'$. Because both $p(\theta | D, \alpha)$ and $h(\theta | D + \alpha)$ are pdfs, the product of the three first factors in (97) must be one, hence the posterior distribution of models *theta* is

$$p(\theta | D, \alpha) = h(\theta | D + \alpha). \quad (98)$$

In order to get the posterior mean estimate, we integrate over the model candidates θ with the posterior distribution. I omit the derivation (can be found in Durbin et al. (2000)) and only report the result:

$$\theta_j^{\text{PME}} = \int_{\Theta} \theta_i h(\theta | D + \alpha) d\theta = \frac{n_i + \alpha_i}{N + A}, \quad (99)$$

where $N = n_1 + \cdots + n_{20}$ and $A = \alpha_1 + \cdots + \alpha_{20}$. If one compares this to the maximum-likelihood estimates

$$\theta_j^{\text{ML}} = \frac{n_i}{N}, \quad (100)$$

we can see that the α_j parameters of the Dirichlet distribution can be understood as “pseudo-counts” that are added to the actually observed counts. These pseudo-counts reflect the subjective intuitions of the biologist, and there is no formal rule of how to set them correctly.

Adding the α_j pseudocounts in (99) can also be considered just as a regularization tool in a frequentist setting. One would skip the Bayesian computations that give rise to the Bayesian posterior (98) and directly use (99), optimizing α to navigate on the model flexibility scale in a cross-validation scheme. With α set to all zero, one gets the maximum-likelihood estimate (100) which will usually be overfitting; with large values for α one smoothes out the information contained in the data — underfitting.

The textbook of Durbin et al, from which this example is taken, shares some thoughts on to how a biosequence modeler should select the pseudo-counts. The proper investment of such soft knowledge makes all the difference in real-life machine learning problems.

Appendix

A Elementary mathematical structure-forming operations

A.1 Pairs, tuples and indexed families

If two mathematical objects $\mathcal{O}_1, \mathcal{O}_2$ are given, they can be grouped together in a single new mathematical structure called the *ordered pair* (or just *pair*) of $\mathcal{O}_1, \mathcal{O}_2$. It is written as

$$(\mathcal{O}_1, \mathcal{O}_2).$$

In many cases, $\mathcal{O}_1, \mathcal{O}_2$ will be of the same kind, for instance both are integers. But the two objects need not be of the same kind. For instance, it is perfectly possible to group integer $\mathcal{O}_1 = 3$ together with a random variable (a function!) $\mathcal{O}_2 = X_7$ in a pair, getting $(3, X_7)$.

The crucial property of a pair $(\mathcal{O}_1, \mathcal{O}_2)$ which distinguishes it from the set $\{\mathcal{O}_1, \mathcal{O}_2\}$ is that the two members of a pair are *ordered*, that is, it makes sense to speak of the “first” and the “second” member of a pair. In contrast, it makes no sense to speak of the “first” or “second” element of the set $\{\mathcal{O}_1, \mathcal{O}_2\}$. Related to this is the fact that the two members of a pair can be the same, for instance $(2, 2)$ is a valid pair. In contrast, $\{2, 2\}$ makes no sense.

A generalization of pairs is *N-tuples*. For an integer $N > 0$, an N -tuple of N objects $\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_N$ is written as

$$(\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_N).$$

1-tuples are just individual objects; 2-tuples are pairs, and for $N > 2$, N -tuples are also called *lists* (by computer scientists that is; mathematicians rather don’t use that term). Again, the crucial property of N -tuples is that one can identify its i -th member by its position in the tuple, or in more technical terminology, by its *index*. That is, in an N -tuple, every index $1 \leq i \leq N$ “picks” one member from the tuple.

The infinite generalization of N -tuples is provided by *indexed families*. For any nonempty set I , called an *index set* in this context,

$$(\mathcal{O}_i)_{i \in I}$$

denotes a compound object assembled from as many mathematical objects as there are index elements $i \in I$, and within this compound object, every individual member \mathcal{O}_i can be “addressed” by its index i . One simply writes

$$\mathcal{O}_i$$

to denote the i th “component” of $(\mathcal{O}_i)_{i \in I}$. Writing \mathcal{O}_i is a shorthand for applying the i th projection function on $(\mathcal{O}_i)_{i \in I}$, that is, $\mathcal{O}_i = \pi_i((\mathcal{O}_i)_{i \in I})$.

A.2 Products of sets

We first treat the case of products of a finite number of sets. Let S_1, \dots, S_N be (any) sets. Then the product $S_1 \times \dots \times S_N$ is the set of all N -tuples of elements from the corresponding sets, that is,

$$S_1 \times \dots \times S_N = \{(s_1, \dots, s_N) \mid s_i \in S_i\}.$$

This generalizes to infinite products as follows. Let I be any set — we call it an *index* set in this context. For every $i \in I$, let S_i be some set. Then the *product set indexed by I* is the set of functions

$$\prod_{i \in I} S_i = \{\varphi : I \rightarrow \bigcup_{i \in I} S_i \mid \forall i \in I : \varphi(i) \in S_i\}.$$

Using the notation of indexed families, this could equivalently be written as

$$\prod_{i \in I} S_i = \{(s_i)_{i \in I} \mid \forall i \in I : s_i \in S_i\}.$$

If all the sets S_i are the same, say S , then the product $\prod_{i \in I} S_i = \prod_{i \in I} S$ is also written as S^I .

An important special case of infinite products is obtained when $I = \mathbb{N}$. This situation occurs universally in modeling stochastic processes with discrete time. The elements $n \in \mathbb{N}$ are the points in time when the amplitude of some signal is measured. The amplitude is a real number, so at any time $n \in \mathbb{N}$, one records an amplitude value $a_n \in S_n = \mathbb{R}$. The product set

$$\prod_{n \in \mathbb{N}} S_n = \{\varphi : \mathbb{N} \rightarrow \bigcup_{n \in \mathbb{N}} S_n \mid \forall n \in \mathbb{N} : \varphi(n) \in S_n\} = \{\varphi : \mathbb{N} \rightarrow \mathbb{R}\}$$

is the set of all right-infinite real-valued timeseries (with discrete time points starting at time $n = 0$).

A.3 Products of functions

First, again, the case of finite products: let f_1, \dots, f_N be functions, all sharing the same domain D , with image sets S_i . Then the product $f_1 \otimes \dots \otimes f_N$ of these functions is the function with domain D and image set $S_1 \times \dots \times S_N$ given by

$$\begin{aligned} f_1 \otimes \dots \otimes f_N : D &\rightarrow S_1 \times \dots \times S_N \\ d &\mapsto (f_1(d), \dots, f_N(d)). \end{aligned}$$

Again this generalizes to arbitrary products. Let $(f_i : D \rightarrow S_i)_{i \in I}$ be an indexed family of functions, all of them sharing the same domain D , and where the image set of f_i is S_i . The product $\bigotimes_{i \in I} f_i$ of this set of functions is defined by

$$\begin{aligned} \bigotimes_{i \in I} f_i : D &\rightarrow \prod_{i \in I} S_i \\ d &\mapsto \varphi : I \rightarrow \bigcup_{i \in I} S_i \quad \text{given by } \varphi(i) = f_i(d). \end{aligned}$$

B Joint, conditional and marginal probabilities

Note. This little section is only a quick memory refresher of some of the most basic concepts of probability. It does not replace a textbook chapter!

We first consider the case of two observations of some part of reality that have discrete values. For instance, an online shop creating customer profiles may record from their customers their age and gender (among many other items). The marketing optimizers of that shop are not interested in the exact age but only in age brackets, say $a_1 = \text{at most 10 years old}$, $a_2 = 11 - 20$ years, $a_3 = 21 - 30$ years, $a_4 = \text{older than 30}$. Gender is roughly categorized into the possibilities $g_1 = \text{f}$, $g_2 = \text{m}$, $g_3 = \text{o}$. From their customer data the marketing guys estimate the following probability table:

$P(X = g_i, Y = a_j)$	a_1	a_2	a_3	a_4	(101)
g_1	0.005	0.3	0.2	0.04	
g_2	0.005	0.15	0.15	0.04	
g_3	0.0	0.05	0.05	0.01	

The cell (i, j) in this 3×4 table contains the probability that a customer with gender g_i falls into the age bracket a_j . This is the *joint probability* of the two observation values g_i and a_j . Notice that all the numbers in the table sum to 1.

The mathematical tool to formally describe a category of an observable value is a *random variable* (RV). We typically use symbols X, Y, Z, \dots for RVs in abstract mathematical formulas. When we deal with concrete applications, we may also use “telling names” for RVs. For instance, in Table (101), instead of $P(X = g_i, Y = a_j)$ we could have written $P(\text{Gender} = g_i, \text{Age} = a_j)$. Here we have two such observation categories: gender and age bracket, and hence we use two RVs X and Y for gender and age, respectively. In order to specify, for example, that female customers in the age bracket 11-20 occur with a probability of 0.3 in the shop’s customer reservoir (the second entry in the top line of the table), we write $P(X = g_1, Y = a_2) = 0.3$.

Some more info bits of concepts and terminology connected with RVs. You should consider a RV as the mathematical counterpart of a procedure or apparatus to make observations or measurements. For instance, the real-world counterpart of the **Gender** RV could be an electronic questionnaire posted by the online shop, or more precisely, the “what is your age?” box on that questionnaire, plus the whole internet infrastructure needed to send the information entered by the customer back to the company’s webserver. Or in a very different example (measuring the speed of a car and showing it to the driver on the speedometer) the real-world counterpart of a RV **Speed** would be the total on-board circuitry in a car, comprising the wheel rotation sensor, the processing DSP microchip, and the display at the dashboard.

A RV *always* comes with a set of possible outcomes. This set is called the *sample space* of the RV, and I usually denote it with the symbol S . Mathematically,

a sample space is a set. The sample space for the **Gender** RV would be the set $S = \{\text{m}, \text{f}, \text{o}\}$. The sample space for **Age** that we used in the table above was $S = \{\{0, 1, \dots, 10\}, \{11, \dots, 20\}, \{21, \dots, 30\}, \{31, 32, \dots\}\}$. For car speed measuring we might opt for $S = \mathbb{R}^{\geq 0}$, the set of non-negative reals. A sample space can be larger than the set of measurement values that are realistically possible, but it must contain *at least* all the possible values.

Back to our table and the information it contains. If we are interested only in the age distribution of customers, ignoring the gender aspects, we sum the entries in each age column and get the *marginal probabilities* of the RV Y . Formally, we compute

$$P(Y = a_j) = \sum_{i=1,2,3} P(X = g_i, Y = a_j).$$

Similarly, we get the marginal distribution of the gender variable by summing along the rows. The two resulting marginal distributions are indicated in the table (102).

	a_1	a_2	a_3	a_4	
g_1	0.005	0.3	0.2	0.04	0.545
g_2	0.005	0.15	0.15	0.04	0.345
g_3	0.0	0.05	0.05	0.01	0.110
	0.01	0.5	0.4	0.09	

(102)

Notice that the marginal probabilities of age 0.01, 0.5, 0.4, 0.09 sum to 1, as do the gender marginal probabilities.

Finally, the *conditional probability* $P(X = g_i | Y = a_j)$ that a customer has gender g_i given that the age bracket is a_j is computed through dividing the joint probabilities in column j by the sum of all values in this column:

$$P(X = g_i | Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(Y = a_j)}. \quad (103)$$

There are two equivalent versions of this formula:

$$P(X = g_i, Y = a_j) = P(X = g_i | Y = a_j)P(Y = a_j) \quad (104)$$

where the righthand side is called a *factorization* of the joint distribution on the lefthand side, and

$$P(Y = a_j) = \frac{P(X = g_i, Y = a_j)}{P(X = g_i | Y = a_j)}, \quad (105)$$

demonstrating that each of the three quantities (joint, conditional, marginal probability) can be expressed by the respective two others. If you memorize one of these formulas – I recommend the second one – you have memorized the very key

to master “probability arithmetics” and will never get lost when manipulating probability formulas.

The factorization (104) can be done in two ways: $P(Y = a_j | X = g_i)P(X = g_i) = P(X = g_i | Y = a_j)P(Y = a_j)$, which gives rise to *Bayes’ formula*

$$P(Y = a_j | X = g_i) = \frac{P(X = g_i | Y = a_j)P(Y = a_j)}{P(X = g_i)}, \quad (106)$$

which has many uses in statistical modeling because it shows how one can revert the conditioning direction.

Joint, conditional, and marginal probabilities are also defined when there are more than two categories of observations. For instance, the online shop marketing people also record how much a customer spends on average, and formalize this by a third random variable, say Z . The values that Z can take are spending brackets, say $s_1 = \text{less than 5 Euros}$ to $s_{20} = \text{more than 5000 Euros}$. The joint probability values $P(X = g_i, Y = a_j, Z = s_k)$ would be arranged in a 3-dimensional array sized $3 \times 4 \times 20$, and again all values in this array together sum to 1. Now there are different arrangements for conditional and marginal probabilities, for instance $P(Z = s_k | X = g_i, Y = a_j)$ is the probability that among the group of customers with gender g_i and age a_j , a person spends an amount in the range s_k . Or $P(Z = s_k, Y = a_j | X = g_i)$ is the probability that in the gender group g_i a person is aged a_j and spends s_k . As a last example, the probabilities $P(X = g_i, Z = s_j)$ are the marginal probabilities obtained by *summing away* the Y variable:

$$P(X = g_i, Z = s_j) = \sum_{k=1,2,3,4} P(X = g_i, Y = a_k, Z = s_j) \quad (107)$$

So far I have described cases where all kinds of observations were *discrete*, that is, they (i.e. all RVs) yield values from a finite set – for instance the three gender values or the four age brackets. Equally often one faces *continuous* random values which arise from observations that yield real numbers – for instance, measuring the body height or the weight of a person. Since each such RV can give infinitely many different observation outcomes, their probabilities cannot be represented in a table or array. Instead, one uses *probability density functions* (pdf’s) to write down and compute probability values.

Let’s start with a single RV, say $H = \text{Body Height}$. Since body heights are non-negative and, say, never larger than 3 m, the distribution of body heights within some reference population can be represented by a pdf $f : [0, 3] \rightarrow \mathbb{R}^{\geq 0}$ which maps the interval $[0, 3]$ of possible values to the nonnegative reals (Figure 47). We will be using subscripts to make it clear which RV a pdf refers to, so the pdf describing the distribution of body height will be written f_H .

A pdf for the distribution of a continuous RV X can be used to calculate the probability that this RV takes values within a particular interval, by integrating the pdf over that interval. For instance, the probability that a measurement of

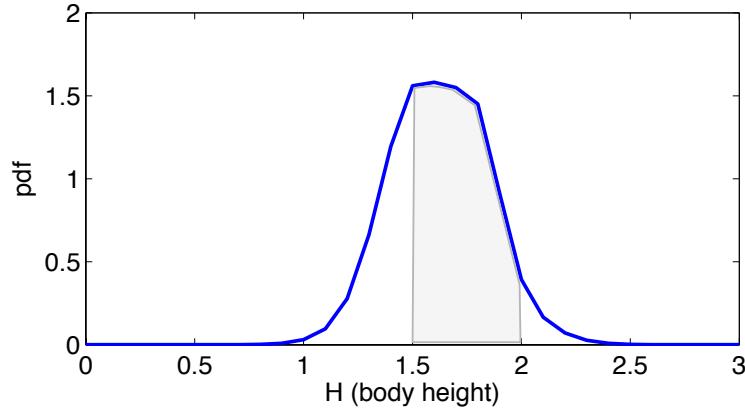


Figure 47: A hypothetical distribution of human body sizes in some reference population, represented by a pdf.

body height comes out between 1.5 and 2.0 meters is obtained by

$$P(H \in [1.5, 2.0]) = \int_{1.5}^{2.0} f_H(x)dx, \quad (108)$$

see the shaded area in Figure 47. Some comments:

- A probability density function is actually *defined* to be a function which allows one to compute probabilities of value intervals as in Equation 108. For a given continuous RV X over the reals there is exactly one function f_X which has this property, *the* pdf for X . (This is not quite true. There exist also continuous-valued RVs whose distribution is so complex that it cannot be captured by a pdf, but we will not meet with such phenomena in this lecture. Furthermore, a given pdf can be altered on isolated points – which come from what is called a *null set* in probability theory – and still be a pdf for the same distribution. But again, we will not be concerned with such subtleties in this lecture.)
- As a consequence, any pdf $f : \mathbb{R} \rightarrow \mathbb{R}^{\geq 0}$ has the property that it integrates to 1, that is, $\int_{-\infty}^{\infty} f(x)dx = 1$.
- Be aware that the values $f(x)$ of a pdf are not probabilities! Pdf's turn into probabilities only through integration over intervals.
- Values $f(x)$ can be greater than 1 (as in Figure 47), again indicating that they cannot be taken as probabilities.

Joint distributions of two continuous RVs X, Y can be captured by a pdf $f_{X,Y} : \mathbb{R}^2 \rightarrow \mathbb{R}^{\geq 0}$. Figure 48 shows an example. Again, the pdf $f_{X,Y}$ of a bivariate

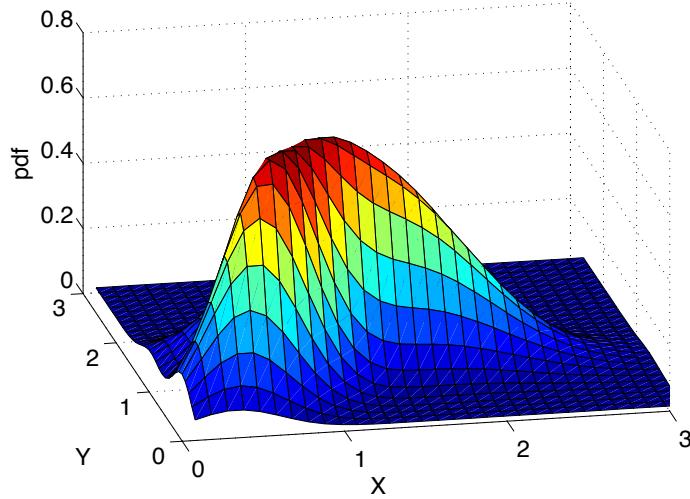


Figure 48: An exemplary joint distribution of two continuous-valued RVs X, Y , represented by its pdf.

continuous distribution must integrate to 1 and be non-negative; and conversely, every such function is the pdf of a continuous distribution of two RV's.

Continuing on this track, the joint distribution of k continuous-valued RVs X_1, \dots, X_k , where the possible values of each X_i are bounded to lie between a_i and b_i can be described by a unique pdf function $f_{X_1, \dots, X_k} : \mathbb{R}^k \rightarrow \mathbb{R}^{\geq 0}$ which integrates to 1, i.e.

$$\int_{a_1}^{b_1} \dots \int_{a_k}^{b_k} f(x_1, \dots, x_k) dx_k \dots dx_1,$$

where also the cases $a_i = -\infty$ and $b_i = \infty$ are possible. A more compact notation for the same integral is

$$\int_D f(\mathbf{x}) d\mathbf{x},$$

where D denotes the k -dimensional box $[a_1, b_1] \times \dots \times [a_k, b_k]$ and \mathbf{x} denotes vectors in \mathbb{R}^k . Mathematicians speak of k -dimensional *intervals* instead of "boxes". The set of points $S = \{\mathbf{x} \in \mathbb{R}^k \mid f_{X_1, \dots, X_k} > 0\}$ is called the *support* of the distribution. Obviously $S \subseteq D$.

In analogy to the 1-dim case from Figure 47, probabilities are obtained from a k -dimensional pdf f_{X_1, \dots, X_k} by integrating over sub-intervals. For such a k -dimensional subinterval $[r_1, s_1] \times \dots \times [r_k, s_k] \subseteq [a_1, b_1] \times \dots \times [a_k, b_k]$, we get its probability by

$$P(X_1 \in [r_1, s_1], \dots, X_k \in [r_k, s_k]) = \int_{r_1}^{s_1} \dots \int_{r_k}^{s_k} f(x_1, \dots, x_k) dx_k \dots dx_1. \quad (109)$$

In essentially the same way as we did for discrete distributions, the pdf's of marginal distributions are obtained by *integrating away* the RV's that one wishes to expel. In analogy to (107), for instance, one would get

$$f_{X_1, X_3}(x_1, x_3) = \int_{a_2}^{b_2} f_{X_1, X_2, X_3}(x_1, x_2, x_3) dx_2. \quad (110)$$

And finally, pdf's of conditional distributions are obtained through dividing joint pdfs by marginal pdfs. Such conditional pdfs are used to calculate that some RVs fall into a certain multidimensional interval given that some other RVs take specific values. We only inspect a simple case analog to (103) where we want to calculate the probability that X falls into a range $[a, b]$ given that Y is known to be c , that is, we want to evaluate the probability $P(X \in [a, b] | Y = c)$, using pdfs. We can obtain this probability from the joint pdf $f_{X,Y}$ and the marginal pdf f_Y by

$$P(X \in [a, b] | Y = c) = \frac{\int_a^b f_{X,Y}(x, c) dx}{f_Y(c)}. \quad (111)$$

The r.h.s. expression $\int_a^b f_{X,Y}(x, c) dx / f_Y(c)$ is a function of x , parametrized by c . This function is a pdf, denoted by $f_{X|Y=c}$, and defined by

$$f_{X|Y=c}(x) = \frac{f_{X,Y}(x, c)}{f_Y(c)}. \quad (112)$$

Let me illustrate this with a concrete example. An electronics engineer is testing a device which transforms voltages V into currents I . In order to empirically measure the behavior of this device (an electronics engineer would say, in order to “characterize” the device), the engineer carries out a sequence of measurement trials where he first sets the input voltage V to a specific value, say $V = 0.0$. Then he (or she) measures the resulting current many times, in order to get an idea of the stochastic spread of the current. In mathematical terms, the engineer wants to get an idea of the pdf $f_{I|V=0.0}$. The engineer then carries on, setting the voltage to other values c_1, c_2, \dots , measuring resulting currents in each case, and getting ideas of the conditional pdfs $f_{I|V=c_i}$. For understanding the characteristics of this device, the engineer needs to know all of these pdfs.

Conditional distributions arise whenever cause-effect relationships are being modeled. The conditioning variables are causes, the conditioned variables describe effects. In experimental and empirical research, the causes are under the control of an experimenter and can (and have to) be set to specific values in order to assess the statistics of the effects – which are not under the control of the experimenter. In ML pattern classification scenarios, the “causes” are the input patterns and the “effects” are the (stochastically distributed) class label assignments. Since research in the natural sciences is very much focussed on determining Nature's cause-effect workings, and 90% of the applications in machine learning concern

pattern classification (my estimate), it is obvious that conditional distributions lie at the very heart of scientific (and engineering) modeling and data analysis.

In this appendix (and in the lecture) I consider only two ways of representing probability distributions: discrete ones by finite probability tables or probability tables; continuous ones by pdfs. These are the most elementary formats of representing probability distributions. There are many others which ML experts readily command on. This large and varied universe of concrete *representations* of probability distributions is tied together by an abstract mathematical theory of the probability distributions themselves, independent of particular representations. This theory is called *probability theory*. It is not an easy theory and we don't attempt an introduction to it. If you are mathematically minded, then you can get an introduction to probability theory in my graduate lecture notes "Principles of Statistical Modeling" (<http://minds.jacobs-university.de/teaching/ln/>). At this point I only highlight two core facts from probability theory:

- A main object of study in probability theory are distributions. They are abstractly and axiomatically defined and analyzed, without reference to particular representations (such as tables or pdfs).
- A probability distribution *always* comes together with random variables. We write P_X for the distribution of a RV X , $P_{X,Y}$ for the joint distribution of two RVs X, Y , and $P_{X|Y}$ for the conditional distribution (a truly involved concept since it is actually a family of distributions) of X given Y .

C The argmax operator

Let $\varphi : D \rightarrow \mathbb{R}$ be some function from some domain D to the reals. Then

$$\operatorname{argmax}_a \varphi(a)$$

is that $d \in D$ for which $\varphi(d)$ is maximal among all values of φ on D . If there are several arguments a for which φ gives the same maximal value, – that is, φ does not have a unique maximum –, or if φ has no maximum at all, then the argmax is undefined.

D Expectation, variance, covariance, and correlation of numerical random variables

Recall that a random variable is the mathematical model of an observation / measurement / recording procedure by which one can "sample" observations from that piece of reality that one wishes to model. We usually denote RVs by capital roman letters like X, Y or the like. For example, a data engineer of an internet

shop who wants to get a statistical model of its (potential) customers might record the gender and age and spending of shop visitors – this would be formally captured by three random variables G, A, S . A random variable always comes together with a *sample space*. This is the set of values that might be delivered by the random variable. For instance, the sample space of the gender RV G could be cast as $\{\text{m, f, o}\}$ – a symbolic (and finite) set. A reasonable sample space for the age random variable A would be the set of integers between 0 and 200 – assuming that no customer will be older than 200 years and that age is measured in integers (years). Finally, a reasonable sample space for the spending RV S could be just the real numbers \mathbb{R} .

Note that in the A and S examples, the sample spaces that I proposed look very generous. We would not really expect that some customer is 200 years old, nor would we think that ever a customer spends 10^{1000} Euros – although both values are included in the respective sample space. The important thing about a sample space is that it must contain all the values that might be returned by the RV; but it may also contain values that will never be observed in practice.

Every mathematical set can serve as a sample space. We just saw symbolic, integer, and real sample spaces. Real sample spaces are used whenever one is dealing with an observation procedure that returns numerical values. Real-valued RVs are of great practical importance, and they allow many insightful statistical analyses that are not defined for non-numerical RVs. The most important analytical characteristics of real RVs are expectation, variance, and covariance, which I will now present in turn.

For the remainder of this appendix section we will be considering random variables X whose sample space is \mathbb{R}^n — that is, observation procedures which return scalars (case $n = 1$) or vectors. We will furthermore assume that the distributions of all RVs X under consideration will be represented by pdf's $f_X : \mathbb{R}^n \rightarrow \mathbb{R}^{\geq 0}$. (In mathematical probability theory, more general numerical sample spaces are considered, as well as distributions that have no pdf — but we will focus on this basic scenario of real-valued RVs with pdfs).

The *expectation* of a RV X with sample space \mathbb{R}^n and pdf f_X is defined as

$$E[X] = \int_{\mathbb{R}^n} x f_X(x) dx, \quad (113)$$

where the integral is written in a common shorthand for

$$\int_{x_1=-\infty}^{\infty} \dots \int_{x_n=-\infty}^{\infty} (x_1, \dots, x_n)' f_X((x_1, \dots, x_n)) dx_n \dots dx_1.$$

The expectation of a RV X can be intuitively understood as the “average” value that is delivered when the observation procedure X would be carried out infinitely often. The crucial thing to understand about the expectation is that it does not depend on a sample, – it does not depend on specific data.

In contrast, whenever in machine learning we base some learning algorithm on a (numerical) training sample $(x_i, y_i)_{i=1,\dots,N}$ drawn from the joint distribution $P_{X,Y}$ of two RVs X, Y , we may compute the average value of the x_i by

$$\text{mean}(\{x_1, \dots, x_N\}) = 1/N \sum_{i=1}^N x_i,$$

but this *sample mean* is NOT the expectation of X . If we would have used another random sample, we would most likely have obtained another sample mean. In contrast, the expectation $E[X]$ of X is defined not on the basis of a finite, random sample of X , but it is defined by averaging over the true underlying distribution.

Since in practice we will not have access to the true pdf f_X , the expectation of a RV X cannot usually be determined in full precision. The best one can do is to *estimate* it from observed sample data. The sample mean is an *estimator* for the expectation of a numerical RV X . Marking estimated quantities by a “hat” accent, we may write

$$\hat{E}[X] = 1/N \sum_{i=1}^N \hat{x}_i.$$

A random variable X is *centered* if its expectation is zero. By subtracting the expectation one gets a centered RV. In these lecture notes I use the bar notation to mark centered RVs:

$$\bar{X} := X - E[X].$$

The *variance* of a scalar RV with sample space \mathbb{R} is the expected squared deviation from the expectation

$$\sigma^2(X) = E[\bar{X}^2], \quad (114)$$

which in terms of the pdf $f_{\bar{X}}$ of \bar{X} can be written as

$$\sigma^2(X) = \int_{\mathbb{R}} x^2 f_{\bar{X}}(x) dx.$$

Like the expectation, the variance is an intrinsic property of an observation procedure X and the part of the real world where the measurements may be taken from — it is independent of a concrete sample. A natural way to estimate the variance of X from a sample $(x_i)_{i=1,\dots,N}$ is

$$\hat{\sigma}^2(\{x_1, \dots, x_N\}) = 1/N \sum_{i=1}^N \left(x_i - 1/N \sum_{j=1}^N x_j \right)^2,$$

but in fact this estimator is not the best possible — on average (across different samples) it underestimates the true variance. If one wishes to have an estimator

that is *unbiased*, that is, which on average across different samples gives the correct variance, one must use

$$\hat{\sigma}^2(\{x_1, \dots, x_N\}) = 1/(N-1) \sum_{i=1}^N \left(x_i - 1/N \sum_{j=1}^N x_j \right)^2$$

instead. The Wikipedia article on “Variance”, section “Population variance and sample variance” points out a number of other pitfalls and corrections that one should consider when one estimates variance from samples.

The square root of the variance of X , $\sigma(X) = \sqrt{\sigma^2(X)}$, is called the *standard deviation* of X .

The *covariance* between two real-valued scalar random variables X, Y is defined as

$$\text{Cov}(X, Y) = E[\bar{X} \bar{Y}], \quad (115)$$

which in terms of a pdf $f_{\bar{X} \bar{Y}}$ for the joint distribution for the centered RVs spells out to

$$\text{Cov}(X, Y) = \int_{\mathbb{R} \times \mathbb{R}} x y f_{\bar{X} \bar{Y}}((x, y)') dx dy.$$

An unbiased estimate of the covariance, based on a sample $(x_i, y_i)_{i=1, \dots, N}$ is given by

$$\widehat{\text{Cov}}((x_i, y_i)_{i=1, \dots, N}) = 1/(N-1) \left(x_i - 1/N \sum_i x_i \right) \left(y_i - 1/N \sum_i y_i \right).$$

Finally, let us inspect the *correlation* of two scalar RVs X, Y . Here we have to be careful because this term is used differently in different fields. In statistics, the correlation is defined as

$$\text{Corr}(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma(X) \sigma(Y)}. \quad (116)$$

It is easy to show that $-1 \leq \text{Corr}(X, Y) \leq 1$. The correlation in the understanding of statistics can be regarded as a normalized covariance. It has a value of 1 if X and Y are identical up to some positive scaling factor, it has a value of -1 if X and Y are identical up to some negative scaling factor. When $\text{Corr}(X, Y) = 0$, X and Y are said to be *uncorrelated*.

The quantity $\text{Corr}(X, Y)$ is also referred to as (*population*) *Pearson's correlation coefficient*, and is often denoted by the greek letter $\varrho(X, Y) = \text{Corr}(X, Y)$.

In the signal processing literature (for instance in my favorite textbook Farhang-Boroujeny (1998)), the term “correlation” is sometimes used in quite a different way, denoting the quantity

$$E[X Y],$$

that is, simply the expectation of the product of the uncentered RVs X and Y . Just be careful when you read terms like “correlation” or “cross-correlation” or

“cross-correlation matrix” and make sure that your understanding of the term is the same as the respective author’s.

There are some basic rules for doing calculations with expectations and covariance which one should know:

1. Expectation is a linear operator:

$$E[\alpha X + \beta Y] = \alpha E[X] + \beta E[Y],$$

where αX is the RV obtained from X by scaling observations with a factor α .

2. Expectation is idempotent:

$$E[E[X]] = E[X].$$

- 3.

$$\text{Cov}(X, Y) = E[XY] - E[X]E[Y].$$

E Derivation of Equation 31

$$\begin{aligned}
& 1/N \sum_i \|x_i - \mathbf{d} \circ \mathbf{f}(x_i)\|^2 = \\
&= 1/N \sum_i \|\bar{x}_i - \sum_{k=1}^m (\bar{x}'_i u_k) u_k\|^2 \\
&= 1/N \sum_i \left\| \sum_{k=1}^n (\bar{x}'_i u_k) u_k - \sum_{k=1}^m (\bar{x}'_i u_k) u_k \right\|^2 \\
&= 1/N \sum_i \left\| \sum_{k=m+1}^n (\bar{x}'_i u_k) u_k \right\|^2 \\
&= 1/N \sum_i \sum_{k=m+1}^n (\bar{x}'_i u_k)^2 = \sum_{k=m+1}^n 1/N \sum_i (\bar{x}'_i u_k)^2 \\
&= \sum_{k=m+1}^n \sigma_k^2.
\end{aligned}$$

References

- A. C. Antoulas and D. C. Sorensen. Approximation of large-scale dynamical systems: an overview. *Int. J. Appl. Math. Comput. Sci.*, 11(5):1093–1121, 2001.
- D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2015. URL <http://arxiv.org/abs/1409.0473v6>.
- J. A. Bednar and S. P. Wilson. Cortical maps. *The Neuroscientist*, 22(6):604–617, 2016.
- L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.
- G. E. Crooks. Field guide to continuous probability distributions, v 0.11 beta. online manuscript, retrieved april 2017, 2017. URL <http://threeplusone.com/fieldguide>.
- A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM-algorithm. *Journal of the Royal Statistical Society*, 39: 1–38, 1977.
- R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (second edition)*. Wiley Interscience, 2001.
- R. Durbin, S. Eddy, A. Krogh, and G. Mitchinson. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 2000.
- D. Durstewitz, J. K. Seamans, and T. J. Sejnowski. Neurocomputational models of working memory. *Nature Neuroscience*, 3:1184–91, 2000.
- S. Edelman. The minority report: some common assumptions to reconsider in the modelling of the brain and behaviour. *J of Experimental and Theoretical Artificial Intelligence*, 2015. URL <http://www.tandfonline.com/action/showCitFormats?doi=10.1080/0952813X.2015.1042534>.
- B. Farhang-Boroujeny. *Adaptive Filters: Theory and Applications*. Wiley, 1998.
- S. Fusi and X.-J. Wang. Short-term, long-term, and working memory. In M. Arbib and J. Bonaiuto, editors, *From Neuron to Cognition via Computational Neuroscience*, pages 319–344. MIT Press, 2016.
- I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *Proc. ICLR 2015*, 2014. arXiv:1412.6572v3.
- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Open access version at <http://www.deeplearningbook.org>.

- A. N. Gorban, B. Kegl, D. C. Wunsch, and A. (eds.) Zinovyev. *Principal Manifolds for Data Visualization and Dimension Reduction*. Springer, 2008.
- A. Graves et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 7626:471–476, 2016.
- L. Hyafil and R. L. Rivest. Computing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.
- G. Indiveri. Rounding methods for neural networks with low resolution synaptic weights. arxiv preprint, Institute of Neuroinformatics, Univ. Zurich, 2015. URL <http://arxiv.org/abs/1504.05767>.
- E. T. Jaynes. *Probability Theory: the Logic of Science*. Cambridge University Press, 2003, first partial online editions in the late 1990ies. First three chapters online at <http://bayes.wustl.edu/etj/prob/book.pdf>.
- R. Kiros, R. Salakhutdinov, and R. S. Zemel. Unifying visual-semantic embeddings with multimodal neural language models. <http://arxiv.org/abs/1411.2539> Presented at NIPS 2014 Deep Learning Workshop, 2014.
- J. Kittler, M. Hatef, R. P. W. Duin, and J. Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998.
- D. Luchinsky and et al. Overheating anomalies during flight test due to the base bleeding. In *Proc. 7th Int. Conf. on Computational Fluid Dynamics, Hawaii July 2012*, 2012.
- T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. 2013. URL <http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf>.
- A. Minnaar. Wor2Vec tutorial part I: the Skip-Gram model. Technical report, 2015. Online tutorial, retrieved from <http://mccormickml.com/2016/04/27/word2vec-resources/#efficient-estimation-of-word-representations-in-vector-space>. Can serve as understandably detailed presentation to Mikolov’s original ideas.
- T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. In *Proc. of BMVC*, 2015. URL <http://www.robots.ox.ac.uk:5000/~vgg/publications/2015/Parkhi15/parkhi15.pdf>.

- R. Pascanu and H. Jaeger. A neurodynamical model for working memory. *Neural Networks*, 24(2):199–207, 2011. DOI: 10.1016/j.neunet.2010.10.003.
- S. E. Peters, C. Zhang, M. Livny, and C. Ré. A machine reading system for assembling synthetic paleontological databases. *PLOS-ONE*, 9(12):e113523, 2014. URL <http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0113523>.
- S. Roweis and Z. Ghahramani. A unifying review of linear Gaussian models. *Neural Computation*, 11(2):305–345, 1999.
- D. Silver et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- F. Suchanek, J. Fan, R. Hoffmann, S. Riedel, and P. P. Talukdar. Advances in automated knowledge base construction. *SIGMOD Records Journal*, (March), 2013. URL <http://suchanek.name/work/publications/sigmodrec2013akbc.pdf>.
- N. V. Swindale and H.-U. Bauer. Application of Kohonen’s self-organizing feature map algorithm to cortical maps of orientation and direction preference. *Proc. R. Soc. Lond. B*, 265:827–838, 1998.
- F. Takens. Detecting strange attractors in turbulence. In D.A. Rand and L.-S. Young, editors, *Dynamical Systems and Turbulence*, number 898 in Lecture Notes in Mathematics, pages 366–381. Springer-Verlag, 1991.
- J. Tenenbaum, T. L. Griffiths, and C. Kemp. Theory-based bayesian models of inductive learning and reasoning. *Trends in Cognitive Science*, 10(7):309–318, 2006.
- M. Weliky, W. H. Bosking, and D. Fitzpatrick. A systematic map of direction preference in primary visual cortex. *Nature*, 379:725 – 728, 1996.
- H. Yin. Learning nonlinear principal manifolds by self-organising maps. In A. N. Gorban, B. Kgl, D. C. Wunsch, and A. Zinovyev, editors, *Principal Manifolds for Data Visualization and Dimension Reduction*, volume 58 of *Lecture Notes in Computer Science and Engineering*, pages 68–95. Springer, 2008.
- P. Young, A. Lai, M. Hodosh, and J. Hockenmaier. From image descriptions to visual denotations: New similarity metrics for semantic inference over event descriptions. *Transactions of the Association for Computational Linguistics*, 2: 67–78, 2014.