Data Science Lecture Notes from Konrad Völkel

Dr. Konrad Völkel

Monday 16th November, 2026

1 Introduction

1.1 What is Data Science?

1.1.1 First off: a spreadsheet that sent Europe into austerity

In 2010 the Harvard economists Carmen Reinhart and Kenneth Rogoff published an analysis of two centuries of public debt and economic growth Reinhart and Rogoff [2010]. Their finding: as soon as a country's debt exceeds 90% of its economic output, the economy shrinks by an average of 0.1% per year. The number went places. It appeared in the US Congress budget proposal, the EU monetary affairs commissioner quoted it, and it served as an argument for austerity in Europe after the financial crisis.

Three years later Thomas Herndon, a doctoral student at Amherst, was given a seminar assignment: recompute the numbers of a well-known study. He asked for the spreadsheet and found a selected cell range that was five rows short. Australia, Austria, Belgium, Canada and Denmark were missing from the average – all of them countries with high debt and respectable growth. On top of that came omitted years and an unusual weighting. With all three corrected, $-0.1\,\%$ growth turned into $+2.2\,\%$ Herndon et al. [2014]. The cliff at 90% did not exist.

Everything this course is about is in that story. The raw numbers were real and public; the analysis was a handful of averages; and the result was still wrong, because nobody had checked which rows actually went into the calculation, and because a spreadsheet does not show you that. Finding the error took no better mathematics, just somebody willing to pick the data up again from the start. That is exactly what we practise: loading data instead of clicking on it; checking every summary figure against the rows it is made of; and writing calculations down so that others can repeat them. It is why we work with Python and not with Excel in this course, and why the chapter Data cleaning insists on logging every change made to the data.

1.1.2 What is "data"?

Gerhard Goos, *Vorlesungen über Informatik*, volume 1:

> The representation of a communication by the variation of a physical quantity over time is called a *signal*. If, in representing and passing on a communication, we abstract away from the medium used and from the details of the signals and their parameters, the communication is called a *message*.
>
> The knowledge needed to assign meaning to a message we call a *context*, or a frame of reference. The meaning assigned is called *information*. The pair <message, assigned information>is called a *datum*. The plural is data.

So we may call a CSV file "data" exactly when we have a context for giving meaning to the bit strings in it. Most of that context never registers consciously – we assume, for instance, that a CSV file opens in a text editor and that its contents are encoded as strings from which we could guess the data types just by looking. How each field of a table is encoded, though, is often something the table itself does not tell you; for that you need metadata.

Take the answers to a single-choice question: what the message usually records is the number of the chosen answer, not the string of the answer itself. That string is written down in a *code book*, a coding table, and to make sense of the answers as data we have to go back to it.

1.1.3 A colourful mix

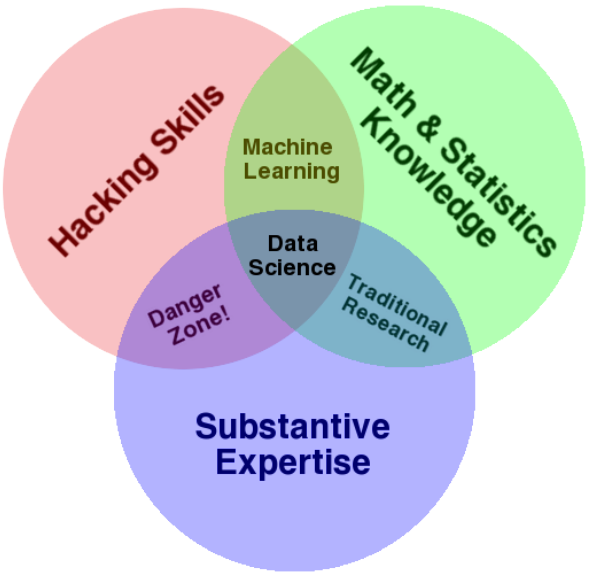


Figure 1: The "Data Science Venn Diagram" by Drew Conway (2013), CC BY-NC 3.0, source: drewconway.com/zia/2013/3/26/the-data-science-venn-diagram

In 2013 Drew Conway gave what is still the most-quoted answer to the question of what data science consists of: the intersection of three circles. *Hacking skills* means being able to handle data in arbitrary formats by programming, *math & statistics knowledge* means knowing what a calculation on that data actually says, and *substantive expertise* is domain knowledge about the field the data comes from. Any two of the circles give you something familiar: hacking and statistics together are *machine learning*, statistics and domain knowledge are *traditional research* as the empirical sciences have long practised it. Hacking and domain knowledge without statistics is what Conway calls the *danger zone*: someone who can read data in, turn a method loose on it and find the results plausible for their field will believe them, without being able to judge whether they are anything more than chance. The spreadsheet at the start of this chapter shows that this zone is dangerous for experts too. This course supplies the statistics circle and practises the hacking circle on small datasets; the domain knowledge you will bring yourself, later, from whatever field you end up working in.

1.1.4 What is a data scientist?

> Nearly all [...] working data scientists make their daily bread and butter through data collection and data cleaning; building dashboards and reports; data visualization; statistical inference; communicating results to key stakeholders; and convincing decision makers of their results. Hugo Bowne-Anderson, Harvard Business Review, 2018

In short: most of the daily work is getting hold of data, cleaning it and communicating the results – not the modelling that takes up most of the room in courses like this one.

> But how do you find data scientists? Whenever someone asks that question, I refer them back to a more fundamental question: what makes a good data scientist? Here is what I look for: · Technical expertise: the best data scientists typically have deep expertise in some scientific discipline. · Curiosity: a desire to go beneath the surface and discover and distill a problem down into a very clear set of hypotheses that can be tested. · Storytelling: the ability to use data to tell a story and to be able to communicate it effectively. · Cleverness: the ability to look at a problem in different, creative ways. All the top data scientists share an innate sense of curiosity. Their curiosity is broad, and extends well beyond their day-to-day activities. They are interested in understanding many different areas of the company, business, industry, and technology. As a result, they are often able to bring disparate areas together in a novel way. For example, I've seen data scientists look at sales processes and realize that by using data in new ways they can make the sales team far more efficient. I've seen data scientists

> apply novel DNA sequencing techniques to find patterns of fraud. What unifies all these people? They all have strong technical backgrounds. Most have advanced degrees (although I've worked with several outstanding data scientists who haven't graduated from college). But the real unifying thread is that all have had to work with a tremendous amount of data before starting to work on the "real" problem DJ Patil: Building Data Science Teams (2011), What Makes a Data Scientist? (p.11ff)

What Patil describes, then, is how to recognise a good data scientist: by depth in some discipline, and above all by the curiosity to keep taking a problem apart until testable hypotheses fall out of it.

A large part of a data scientist's work is preparing data – "data scrubbing". A good deal of classical programming craft goes into it: regular expressions, say, and a knowledge of the industry standards for data types, along with the ways practice departs from them. The list is surprisingly concrete: time and date formats including time zones and daylight saving changes, GIS formats for anything spatial, country codes (of which there are several competing systems), Unicode and the encodings it was meant to replace, and the usual departures from the CSV standard – separators, quotation marks, line breaks in the middle of a field. The craft of it has a chapter of its own in this book, Data cleaning.

Three more things it needs are taught in no lecture: patience, practice, and domain knowledge of the field of application. No data format tells you whether a blood pressure reading of 400 is a measurement error or an age of 0 a missing value – only someone who knows the field can, or someone who asks a person who does. In machine learning too, deep learning included, taking care of the training data buys considerably better results.

1.1.5 Further reading

A career in data science

Robinson and Nolis: Build a Career in Data Science

There is also an hour-long podcast with the authors.

State of Data Science 2021

An hour-long podcast: State of Data Science in 2021 with Stan Seibert

State of Data Science 2024

A blog post from the IDE maker JetBrains: State of Data Science in 2024: 6 Key Data Science Trends

An interesting contrast is a 2010 article by Mike Loukides of O'Reilly, What is data science?. You can see at once that the tools and software packages are in flux, and that some of what looked most exciting in 2010 has already been forgotten. But you can also see that data science is not a passing hype that will soon blow over, but a global trend more than ten years in the making, and that Conway's three circles from the section A colourful mix still describe it well. Labels like "data mining", "big data" and perhaps "data science" too may lose their meaning over time; what is beyond doubt is that statistical methods and machine learning will play a growing role in computer science and in society for a long while yet.

1.1.6 What comes next

Before we work with data, we need a precise language for chance and uncertainty. The next chapter supplies it: probability spaces, events, random variables. It contains two digressions on the measure problem, meant as background and safely skipped on a first reading; the formalism we actually use throughout the course starts at the section *Formalism*.

1.2 Deriving the mathematical formalism of stochastics

You have already met the basics of (discrete) probability theory in Mathematics for Computer Scientists 3. We will go over them again, as far as we need them. In this course we will mostly work with the probability mass function (for discrete random variables) and the probability density function (for continuous random variables), both of which build on the notion of a probability measure. At this point a short digression explains why things are done exactly this way, and how one might arrive at the notion of a probability measure in the first place. The notions of a sigma-algebra (σ-algebra) and of a measurable function are essential for proofs in probability theory, and for introducing more advanced notions (the ones mathematical finance or the theory of machine learning need) – here it is enough to take away when we are allowed to read past them.

Stochastics can be understood as an umbrella term for probability theory and statistics. Statistics builds probabilistic models from observations and draws conclusions from them. Probability theory investigates those models and the theory behind them.

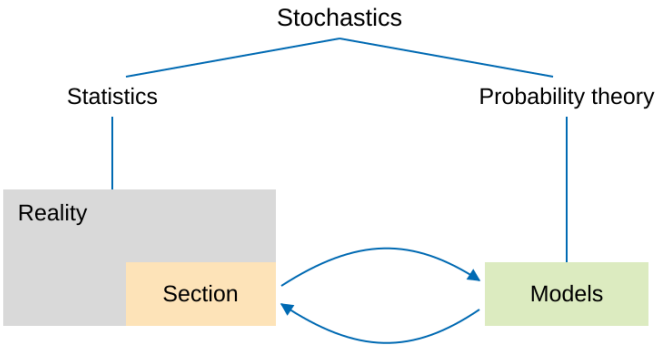


Figure 2: Stochastics as the umbrella term: statistics works on a section of reality, probability theory works on the models.

The two arrows in the middle are what the picture is really saying, and they run in opposite directions. **Modelling** leads from the section of reality to the models: you observe, collect data and set up a model that fits. **Prediction** leads from the models back to reality: you calculate within the model and assert something about the world that can be checked. Both directions are needed, and this book walks each of them several times.

As always in computer science, one essential aim is to model a section of reality: to abstract away from that section, to use the model to predict a piece of reality, and to test the model against reality.

We already know logical models of reality: if it rains, the lawn gets wet. If I am hungry and then eat a lot, I am no longer hungry afterwards.

In the notation of propositional logic, the first model is written as an implication. With N for "the lawn is wet", R for "it has rained" and G for "somebody watered it", it reads

$$N \Rightarrow R \vee G, \tag{1}$$

or, as you would write it in a programming language, `(!N) || R || G`. The model is useful, but it does not answer the question that actually occupies us when we look at the wet lawn: which of the two was it? Does it rain here often? Has anybody ever watered this lawn at all? Propositional logic can only answer "at least one of the two".

Statistical models extend this all-or-nothing logic with degrees of plausibility: if the lawn is wet, it is plausible to assume that it rained recently. It is just as plausible that somebody watered the lawn with a garden hose. Which we find more plausible depends on what else we know about reality. If we watch the lawn for longer, we can establish how often each reason leads to a wet lawn, and sharpen our model accordingly.

If we want to calculate with plausibility the way we calculate with Boolean logic, and eventually implement it on a computer, we have to attach numbers to the plausibility values. The real numbers are a suitable abstraction: they are the only Archimedean ordered complete field containing the rationals (and the subalgebra of computable reals inside them has already served us well).

Example
A first, naive attempt: we simply count. Last month it rained on 20 days, and on 5 days somebody watered. So we set $plaus(R) = 20$ and $plaus(G) = 5$.

How plausible, then, is "it rained or somebody watered"? The obvious answer 25 is wrong as soon as there was even one day on which both happened – that day would be counted twice. The correct answer is

$$plaus(R \vee G) = plaus(R) + plaus(G) - plaus(R \wedge G), \tag{2}$$

and to work it out we need to know whether anybody ever watered into the rain.

The example shows two things. First, calculating with plausibilities is not as naive as it might first appear; it needs rules. Second, the numbers 20 and 5 are unwieldy because they depend on the observation period: two months of observation would give 40 and 10, though nothing about the matter itself has changed.

We also want plausibility logic to be compatible with classical logic: logically equivalent statements should be equally plausible, and different routes to the plausibility of a statement should arrive at the same result (consistency). To put that into mathematics, let $\mathcal{A}$ be a finite set of statements whose plausibility we want to assess; an assignment of plausibility is then a map $p\colon \mathcal{A} \to \mathbb{R}$.

Two rules we want in any case. The first is the formula from the example, known as *inclusion–exclusion*: adding two possibilities counts their overlap twice, so it has to be subtracted once,

$$plaus(A \vee B) = plaus(A) + plaus(B) - plaus(A \wedge B). \tag{3}$$

The second fixes the problem with the observation period. We normalise so that the plausibility of a statement and that of its negation add up to 1,

$$plaus(A) + plaus(\neg A) = 1, \tag{4}$$

since exactly one of the two holds. Every plausibility then lies between 0 and 1, and the absolute length of the observation period drops out: 20 rainy days out of 30 become $\frac{2}{3}$, whether you count for a month or for a year.

Is that already enough to call these numbers "probability"? Not quite – but almost.

Once these requirements are made precise, one can prove Cox's theorem: there is a map cj from the real numbers into the closed interval $[0, 1]$ that takes plausibilities to "probabilities" in such a way that (almost) all the rules of classical probability theory hold. So $P := cj \circ p\colon \mathcal{A} \to [0, 1]$ satisfies the rules

1. $P(TRUE) = 1$ and $P(FALSE) = 0$.
2. If $A_1, \ldots, A_n$ are finitely many pairwise incompatible (mutually exclusive) statements, then finite additivity holds: $P(A_1 \vee \cdots \vee A_n) = \sum_{i=1}^{n} P(A_i)$

We now approach the modern version of the notion of probability. The sigma-algebra will only be mentioned here; afterwards you may (for the purposes of this course) forget it again.

1.2.1 Digression (optional): the measure problem

Note
This digression and the next are pure background reading and **not part of the course proper**. They explain why the definitions in the section *Formalism* look the way they do. If you skip them, carry straight on with the section *Formalism*.

For a die or a coin it is reasonably clear how to model the probability, and how to set up a statistical model that we can also implement. In theory, though, the question soon arises of what happens with, say, an experiment repeated infinitely often. That is not a purely theoretical question: we are interested in the asymptotics, that is, in how an experiment behaves, approximately, once it has been repeated often enough.

An infinitely repeated coin toss can be written down as an infinite sequence of 0s and 1s, and on closer inspection that is also a representation of the real numbers in the unit interval $[0, 1]$ – one that can be implemented on a computer insofar as the computer is a Turing machine (in reality the tape is finite, but we can always extend it; or rather, the memory of our computers is bounded, but not by a fixed value).

We would now like to assign a probability to every outcome (every result of the infinitely repeated coin toss) so that $P(\Omega) = 1$, where $\Omega = \{0, 1\}^{\mathbb{N}}$ is the sample space. Moreover, finite additivity should hold for pairwise disjoint subsets $A_i \subseteq \Omega$:

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

Another natural requirement is flip invariance: it should make no difference to the probability if I flip exactly the n-th coin toss (the n-th bit). This is formalised by an operator $F_n \colon \Omega \to \Omega$ with

$F_n(\omega_1, \dots, \omega_{n-1}, \omega_n, \omega_{n+1}, \dots) = (\omega_1, \dots, \omega_{n-1}, 1 - \omega_n, \omega_{n+1}, \dots)$

We then expect $P(F_n A) = P(A)$ to hold for every subset $A \subseteq \Omega$.

In 1923 Banach proved, using the axiom of choice, that such a probability P can be defined non-constructively – but not uniquely!

If we tighten one of our requirements, from finite additivity to σ-additivity (say "sigma-additivity" or "countable additivity"), then Vitali had already proved in 1905 that no such map P exists at all – so it is impossible to assign probabilities to infinite coin tosses in this way. Why is that, and what does it have to do with σ-additivity?

1.2.2 Digression (optional): measuring open sets

Here is a very practical definition for any set M with a notion of distance (more precisely: a metric space M): a set $U \subseteq M$ is called *open* if around every point $u \in U$ there is a distance $\epsilon > 0$ such that every point $x \in M$ with $d(x, u) < \epsilon$ already lies in U, that is, $x \in U$. For $M = \mathbb{R}$, finite unions of open intervals are open sets of this kind, which is where the name comes from.

One can now show that the classical ϵ-δ definition of continuity is equivalent to this one: $f \colon X \to Y$ is *continuous* if the preimages $f^{-1}(U) \subseteq X$ of open sets $U \subseteq Y$ are open as well.

If we want to define a uniform distribution on the numbers in the interval $[0, 1]$ (as in the infinite coin-toss example), or indeed on all the reals, we already know exactly what we want for intervals: the probability – the measure – of an interval $[a, b]$ should simply be $b - a$.

Since every open set in $\mathbb{R}$ can be written as a countable union of intervals, we know how to assign a measure to the open sets: as the countable sum of the lengths of exactly those intervals. It turns out that a uniform distribution can indeed be defined this way – only that a probability can then no longer be assigned to arbitrary subsets of $\mathbb{R}$ or $[0, 1]$. One says that there are *non-measurable sets*.

From this observation we derive the requirement that our probability P be not merely finitely additive but σ-additive. In the light of Vitali's theorem we accept that not every subset of $\mathbb{R}$ is measurable, and content ourselves with singling out a subalgebra of $\mathcal{P}(\mathbb{R})$: the *measurable sets*.

1.2.3 Formalism

So far we have pinned plausibilities to *statements*: "it rained", "somebody watered". Statements are the right object for logic, but they become unwieldy as soon as you want to calculate – even the question of how many different statements there are leads nowhere. The trick of modern probability theory is to replace every statement by the set of situations in which it holds.

To that end we imagine a set Ω of all **outcomes** ω under consideration – in the lawn example, say, all the states the world could be in this morning. Every statement about the world splits this set in two: the states in which it is true, and the rest. The statement "the lawn is wet" thus becomes the subset $N \subseteq \Omega$ of all world states with a wet lawn, "it rained" becomes $R \subseteq \Omega$ and "somebody watered" becomes $G \subseteq \Omega$. Instead of a plausibility for the statement, we are now after a number for the subset.

The gain: the logical connectives become set operations, which we already know our way around.

Propositional logic	Set logic
A or B, $A \vee B$	$A \cup B$
A and B, $A \wedge B$	$A \cap B$
not A, $\neg A$	$A^c := \Omega \setminus A$

The question of how plausible "it rained or somebody watered" is becomes the question of $P(R \cup G)$, and "it neither rained nor did anybody water" is $R^c \cap G^c$, the complement of $R \cup G$.

A subset of Ω to which we assign a probability is called an **event**. It would be convenient to admit every subset of Ω as an event, and for finite Ω that works. In general – this was the lesson of the measure problem – it does not. So we single out a system $\mathcal{F}$ of "interesting" subsets and require only that it be closed under the operations we have just translated: along with A and B, the sets $A \cup B$, $A \cap B$ and A^c should lie in $\mathcal{F}$, and $\emptyset$ and Ω in any case. The probability is then a map $P\colon \mathcal{F} \to [0,1]$.

In mathematical language such a system of sets is called a σ-algebra – the σ signals that *countably infinite* unions are allowed too, which we need for passing to the limit. With that we have all the ingredients and can write the definitions down.

Definition
Let Ω be a set. Then a system of sets $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is called a *σ-algebra* if

1. $\Omega \in \mathcal{F}$ "TRUE is measurable"
2. $A \in \mathcal{F} \implies A^c := \Omega \setminus A \in \mathcal{F}$ "logical NOT"
3. $A_1, \dots \in \mathcal{F} \implies \bigcup_{i \geq 1} A_i \in \mathcal{F}$ "logical OR"

Terminology: we call the elements of Ω the *outcomes* and the elements of $\mathcal{F}$ the measurable subsets, or *events*. The pair $(\Omega, \mathcal{F})$ is an *event space*.

Example
An ordinary die has state space $\Omega = \{1,2,3,4,5,6\}$, and the power set $\mathcal{P}(\Omega)$ is already a σ-algebra (so there are no non-measurable sets – this always works for finite Ω). A single throw of the die yields an outcome, the number of pips. The statement "the number of pips is even" is a logical statement that corresponds to the event $\{2,4,6\} \subseteq \Omega$. Very often we are interested (of necessity) not in concrete outcomes but in particular events.

Rule of thumb: if it is unclear what the σ-algebra is supposed to be, it is $\mathcal{P}(\Omega)$; in other words, if it is unclear what a "measurable set" is supposed to be, it simply means "set".

Definition
A function $P\colon \mathcal{F} \to [0,1]$ from a σ-algebra on a set Ω into the unit interval is called a *probability measure* if

1. $P(\Omega) = 1$ (normalisation) and
2. for pairwise disjoint events $A_i \in \mathcal{F}$, σ-additivity holds: $P\left(\bigcup_{i \geq 1} A_i\right) = \sum_{i \geq 1} P(A_i)$

The triple $(\Omega, \mathcal{F}, P)$ is called a *probability space*.

Compare this with Definition 12.2.5 and Remark 12.2.8 in the German lecture notes "Mathematik für Informatik", where the special case $\mathcal{F} = \mathcal{P}(\Omega)$ with finite Ω is assumed.

Example
Let us stay with the wet lawn. We set $\Omega = \{R, G, T\}$ (the letters now stand for outcomes, no longer for events as above) for the three explanations "it rained", "somebody watered", "it was dew", and as the σ-algebra we take the power set $\mathcal{F} = \mathcal{P}(\Omega)$. The one-element events $\{R\}, \{G\}, \{T\}$ are called **elementary events**; on a finite set their probabilities already fix the whole measure, since every other event is a disjoint union of them.

A first attempt: $P(\{R\}) = \frac{2}{3}$, $P(\{G\}) = \frac{1}{6}$, $P(\{T\}) = \frac{1}{3}$. This is not a probability measure, because σ-additivity gives

$$P(\Omega) = P(\{R\}) + P(\{G\}) + P(\{T\}) = \frac{2}{3} + \frac{1}{6} + \frac{1}{3} = \frac{7}{6} > 1, \tag{5}$$

contradicting the normalisation $P(\Omega) = 1$. With $P(\{R\}) = \frac{1}{2}$ instead of $\frac{2}{3}$ the sum works out, and we have a probability measure.

A second objection weighs more heavily, and it concerns not the numbers but the model. A probability measure on $\{R, G, T\}$ implicitly claims that the three explanations are mutually exclusive – that it cannot rain *and* somebody water. That is plainly false. Anyone who wants to allow both has to choose the outcomes more finely, for instance

$$\Omega' = \{R\,G,\ R\,\overline{G},\ \overline{R}\,G,\ \overline{R}\,\overline{G}\}, \tag{6}$$

that is, all four combinations of "rained yes/no" and "watered yes/no". The event "it rained" is then no longer an elementary event but the two-element set $\{R\,G,\ R\,\overline{G}\}$.

This shows where the real work of modelling lies: not in distributing the numbers, but in the choice of Ω. The outcomes have to be pairwise exclusive and together cover everything that can happen.

Example

On a finite set Ω of cardinality $N := |\Omega|$ (that is, with N elements), $\mathcal{F} := \mathcal{P}(\Omega)$ together with $P(A) := |A|/N$ defines a probability measure, which we call the *uniform distribution*.

The fair die is the case $\Omega = \{1, \ldots, 6\}$: the event "even number of pips" is $\{2, 4, 6\}$ and has probability $P(\{2, 4, 6\}) = \frac{3}{6} = \frac{1}{2}$. This can also be checked in code, see Programming with stochastics.

Example

For an arbitrary event space $(\Omega, \mathcal{F})$ and a fixed element $\xi \in \Omega$,

$$\delta_\xi := A \mapsto \begin{cases} 1 & \xi \in A, \\ 0 & \text{otherwise} \end{cases}$$

defines a probability measure, the *Dirac measure* with unit mass at the point ξ.

This is the maximally unfair die: a die that always shows a six has distribution δ_6. For it, $P(\{2, 4, 6\}) = 1$ and $P(\{1, 2, 3\}) = 0$ – the event "even number of pips" occurs with certainty, the event "at most three" never. Between these two extremes, the uniform distribution and the Dirac measure, lie all the other dice.

Proposition

Rules of calculation for probability measures: for every probability space $(\Omega, \mathcal{F}, P)$ with $A, B \in \mathcal{F}$:

1. $P(\emptyset) = 0$
2. Finite additivity: $P(A \cup B) + P(A \cap B) = P(A) + P(B)$
3. Monotonicity: $A \subseteq B \implies P(A) \leq P(B)$
4. σ-subadditivity: $P\left(\bigcup_{i\geq 1} A_i\right) \leq \sum_{i\geq 1} P(A_i)$ (where the $A_i \in \mathcal{F}$ need not be disjoint)

Proof

1. $1 = P(\Omega) = P(\emptyset \cup \Omega) = P(\emptyset) + P(\Omega) = P(\emptyset) + 1$
2. $P(A \cup B) = P(A \cup (B \setminus A)) = P(A) + P(B \setminus A)$ and $P(B \setminus A) + P(A \cap B) = P((B \setminus A) \cup (A \cap B)) = P(B)$, hence $P(A \cup B) + P(A \cap B) = P(A) + P(B \setminus A) + P(A \cap B) = P(A) + P(B)$
3. $A \subseteq B \implies B = A \cup (B \setminus A) \implies P(A) \leq P(A) + P(B \setminus A) = P(A \cup (B \setminus A)) = P(B)$
4. left to the reader as an exercise.

Definition
A function $f\colon \Omega \to \Omega'$ between probability spaces Ω and Ω' (we no longer write the σ-algebras and probability measures out separately) is called *measurable*, or a *random variable*, if for every measurable set $U \subseteq \Omega'$ the preimage $f^{-1}(U) \subseteq \Omega$ is measurable as well.

Rule of thumb: if it is unclear what the σ-algebras are supposed to be, we picture every map (= function) as measurable. Strictly speaking that is usually false as soon as continuous random variables are involved – and in practice it will not trouble us all the same.

Example
If we define the uniform distribution on $[0, 1]$ by declaring the interval length to be the probability measure on open intervals, then every continuous function $f\colon [0, 1] \to [0, 1]$ is measurable as well.

The term random variable is highly misleading: there is neither a variable here nor any randomness. The notion of a measurable map makes sense outside probability theory too, but the notion of a random variable is typically used differently and comes with different syntactic habits. Think of this overloading of the term as an API wrapper that gives us extra syntactic sugar – a map that comes dressed as a random variable can be used like a variable that takes random values. We will lean on this perspective a good deal, and practise it.

We will not go further into measure theory here; defining, say, the general Lebesgue integral takes a certain amount of effort. One thing, though, is worth recording because of the applications in computer science, and in machine learning in particular: whereas the Riemann integral approximates with the areas of rectangles, the Lebesgue integral approximates the function itself by step functions.

Definition
A step function is a measurable function taking only finitely many different values.

Theorem
For every σ-algebra on Ω and every measurable function $f\colon \Omega \to \mathbb{R}$ there is a sequence of step functions $t_n\colon \Omega \to \mathbb{R}$ converging to f: $t_n \to f$.

This lets a great many statements be proved by proving them for step functions (the construction of the Lebesgue integral among them). It also yields far-reaching universality statements whenever a model of computation can approximate step functions to arbitrary accuracy – which is how the universality of neural networks can be seen; the course *Introduction to Deep Learning* works this out (universal approximation).

Finally, it should be said that there is a legitimate objection to this formalism: a computer can only implement constructive mathematics, and neither the law of the excluded middle nor the axiom of choice is implementable. In that logic, though (the constructivist, so-called intuitionistic one), every subset of a metric space X may be called measurable, precisely because the non-measurable subsets cannot be constructed. That makes some things easier, but a great many things impossible. So we content ourselves with knowing how to read texts that use the usual formalism, and with having a good excuse at hand for not engaging with it any further.

So you can now pick up a probability textbook, find a theorem saying "let X be a set with a σ-algebra ..." and, for the time being, read past the σ-algebra part, since in practice you will not run into non-measurable sets (those that are not in the system of sets called a σ-algebra). In other words: as a first approximation, imagine that "measurable" is no restriction at all. What we should take away from this chapter at the very least is the notion of a random variable, the rules of calculation for probability measures P, and the important example of the uniform distribution.

If you would like to approach stochastics through interactive playthings as the course goes on, Random Services is just the thing for you. If you prefer things elegant and visual, Seeing Theory will accompany you through the stochastics part of the course.

If you would like the throw of a die (the uniform distribution, also called a Laplace experiment) explained to you again by somebody else in six minutes, you can find that on YouTube on the channel musstewissen Mathe (in German).

1.3 Introduction to Python: data types, control flow and classes

1.3.1 Where Python sits

Python's design philosophy differs from that of C, C++, Perl, Lisp or Pascal, but it borrows freely from earlier languages. An active community process keeps the language evolving.

Python is designed for readability, with a syntax that lets programmers express an idea in fewer lines of code. It is plainly a high-level language, further removed from any concrete machine model than C is. Like Java, it uses garbage collection, so you never have to reserve and release memory by hand just to keep a value in a variable.

Python supports functional programming in the style of R or Haskell as well as object-oriented programming in the style of C# or Java. Unlike those languages, though, Python is *dynamically* typed: the type belongs to the value, not to the name. The same variable name can hold a string, a boolean a moment later, and then an integer. If you prefer stricter types, type annotations and dedicated linters (`mypy`, say) are available these days; the interpreter itself does not check them.

There are two classical ways of using Python. The first is the REPL (read-eval-print loop), which works much like a shell on the command line: you can quickly check whether a short snippet of code is syntactically correct and does what you expect. Every Python user should get comfortable with the interpreter, which is to say with the REPL.

The second is to compile a program (.py) to Python bytecode (.pyc) and have Python execute that, which is how more complex tasks get done. One file can import others as modules.

Originally, packages (which is what Python calls libraries) were added to a Python environment with the program pip (Pip Installs Packages). It fetches them from a package index, typically PyPI, the Python Package Index (the analogue of CPAN for Perl, CRAN for R and CTAN for TeX). Only once a package has been downloaded and installed can you import it into your code and use it.

To keep the packages of different projects out of each other's way, they are installed not system-wide but into a *virtual environment* (venv), one per project. The tool uv does both jobs in one step: it reads the required packages from the project's `pyproject.toml`, creates the environment in the subdirectory `.venv` and downloads the packages from PyPI. For these notes, `uv sync` in the project directory is all it takes; after that, `uv run python` starts the interpreter with every package available, and `uv run jupyter lab` starts the notebook interface. If you would rather do without uv, `python -m venv .venv`, activating the environment and `pip install -r requirements.txt` get you the same result. The Anaconda distribution with its conda package manager, once ubiquitous in data science, works too, but these notes do not use it.

A (Jupyter/IPython) notebook is a document made up of cells. Each cell holds either text, mathematics (LaTeX) or Python code, and the code is executed on the spot, like a REPL in the browser. Under the hood a notebook (.ipynb) is a JSON document.

This book, too, is a collection of notebooks, which Jupyter Book compiles into one coherent set of lecture notes.

Why Python? The choice is neither an accident nor a matter of taste. For years Python has been among the most widely used programming languages there are: in the big developer surveys it regularly comes in just behind JavaScript and HTML/CSS, and in the TIOBE index it has held first place since October 2021. What matters more than the popularity, though, is what has grown out of it.

An ecosystem known as *Scientific Python* has grown up around the language, and this book uses almost nothing else:

- **Numpy** – arrays and vector arithmetic, the foundation for everything that follows (Numpy)
- **SciPy** – distributions, statistical tests, optimisation (Continuous distributions, Statistical tests)
- **Matplotlib** – visualisation (Descriptive statistics)
- **Pandas** – tables with named columns (Pandas)
- **Scikit-learn** – the classical methods of machine learning (Model selection)
- **Jupyter** – the notebooks you are reading right now

Two more packages get a mention and no more: *Seaborn*, which builds on Matplotlib and takes the standard statistical graphics off your hands (in this book Matplotlib stays the only plotting tool), and *PyTorch*, the counterpart for neural networks and the subject of the course *Introduction to Deep Learning*.

These packages coordinate their versions with one another: under NEP 29 and SPEC 0 respectively, each supports a given Python version for about three years. For this book that means the code shown gets by with **Python 3.12** and uses no newer language features, while the notes themselves are built and tested with Python 3.13 or 3.14, because the package versions in `pyproject.toml` demand it. The floor stood at 3.11 until October 2026 and moved up with them, because Numpy from 2.5 and Scipy from 1.18 no longer support Python 3.11. If the right Python is missing, `uv sync` installs that as well.

Four things to remember **1. The REPL is our friend.** Python is an interpreted language, and the interpreter is there to be used interactively: `python` in a console starts the REPL (read-eval-print loop), where you type an expression and see its result at once. Every Jupyter cell is essentially the same thing. If you do not know what a piece of code does, run it rather than puzzling over it – and `help(obj)` and `dir(obj)` will tell you what an object can do.

2. Indentation instead of braces. Python marks its blocks by indentation, not by curly braces. That enforces readable code, and it punishes sloppiness on the spot with an `IndentationError`.

3. Everything is an object. Numbers too, functions too, modules too. An integer has methods, and a function can be assigned to a variable or passed as an argument – every `apply` and every `key=` argument later in the book rests on that.

4. Language models know Python. No programming language has more public code behind it than Python, and language models are correspondingly good at it. That is a real help and a real trap at once: generated code looks plausible even when it is wrong, and running it does not make it right. Check it as you would check somebody else's code: generated code is somebody else's code.

```
print((3).bit_length())      # even a number has methods
print(type(print))           # even a function is an object

square = lambda x: x**2      # functions can be assigned to variables
print(list(map(square, [1, 2, 3])))
```

```
2
<class 'builtin_function_or_method'>
[1, 4, 9]
```

1.3.2 Further reading

There is, of course, no shortage of material on Python by now. Since the language is very much alive, make sure you are working with python3. The latest version at the time of writing is Python 3.14. These notes are built and tested with Python 3.13 or 3.14; the code shown gets by with Python 3.12.

If this chapter leaves you wanting more of the language itself, the notes for the course Einführung in Python (in German, and still under construction) cover what is missing here: packaging and `uv`, testing, profiling, design by contract, typing, debugging and logging, object-oriented programming, asyncio and user interfaces.

A very useful resource is PEP 8, the Python style guide, which records the usual coding conventions. Since whitespace (indentation above all) carries syntactic weight in Python, the style guide is worth skimming once.

PEP stands for Python Enhancement Proposal, the community process through which the language evolves.

The official language documentation is thorough and very useful. For many a task, the Python module index turns up a module that ships with Python and does the job.

Experienced programmers have the freely available Dive Into Python. The freely available Hitchhiker's Guide to Python goes into installation, configuration, IDEs, coding styles and common gotchas at length. A friendly, unhurried introduction to Python for scientists is the Whirlwind Tour of Python (even if the CC0-licensed PDF of the O'Reilly book now takes a little finding).

In German, the Python course by Bernd Klein, among others, is good and up to date.

If you want to explore the more bizarre corners of the language (and of its implementation), WTF Python will serve you well.

1.3.3 The essentials

To read the documentation and the code of Python modules quickly, you need to know Python's syntactic quirks and concepts. Some of the most important are collected here.

Data types **Numbers** are represented as integers (int) or as floating point numbers (float), and there are even complex numbers (complex).

Three things here differ from what you know from C or Java. First, Python integers have **no fixed width and do not overflow**: 2^{100} is a perfectly ordinary number, and Python grows the internal representation as needed. Second, a `float` is an **IEEE 754 binary64** floating point number (what C calls a `double`) – with everything that entails, above all that most decimal fractions cannot be represented exactly. Third, Python is strictly but **dynamically** typed: the type belongs to the value, not to the variable name, and the same name may carry one type after another.

```
print("23 is", type(23), "and 23.0 is", type(23.0))
print("and 23+2j is", type(23+2j))
print("adding an int and a float gives", type(23+23.0))
print("dividing integers gives a float, so 5/2 =", 5/2)
print("if we do not want that, we can compute 5//2 =", 5//2)

print("no overflow:", 2**100)
print("IEEE 754:", 0.1 + 0.2 == 0.3, "since 0.1+0.2 =", 0.1 + 0.2)
print("modulo takes the sign of the divisor:", 7 % 3, -7 % 3, "which goes with", -7 // 3)
```

```
23 is <class 'int'> and 23.0 is <class 'float'>
and 23+2j is <class 'complex'>
adding an int and a float gives <class 'float'>
dividing integers gives a float, so 5/2 = 2.5
if we do not want that, we can compute 5//2 = 2
no overflow: 1267650600228229401496703205376
IEEE 754: False since 0.1+0.2 = 0.30000000000000004
modulo takes the sign of the divisor: 1 2 which goes with -3
```

The middle line is why floating point numbers are **never tested for equality**. Instead of `a == b` you write `abs(a - b) <1e-9` or, in Numpy, `np.isclose(a, b)` – we will meet this throughout the book. The last line shows a quirk that matters when you compute with indices: in Python, `%` always takes the sign of the *divisor*, so `-7 % 3` is 2 rather than the -1 of C or Java, and `//` rounds down to match, rather than towards zero.

Strings are Unicode throughout, with UTF-8 as the default when nothing else is specified. In practice that means source files these days cope with umlauts and accented characters, but that for older files (from the Python 2 era in particular) the encoding may have to be converted or declared. It will not concern us.

Quotation marks may be single or double – `'Data'` and `"Data"` are the same thing – which is handy when the string itself contains quotation marks. Triple quotation marks enclose multi-line strings. What matters more is that strings are **immutable**: no single character can be overwritten, and every method such as `upper()` returns a *new* string rather than changing the old one. In return, slicing works on strings exactly as it does on lists.

```
s = "Data Science"
print(s[0], "|", s[5:], "|", s[ -7:])   # slicing works on strings too

try:
    s[0] = "d"
except TypeError as e:
    print("does not work:", e)

print(s.upper(), "| the original is unchanged:", s)
```

```
D | Science | Science
does not work: 'str' object does not support item assignment
DATA SCIENCE | the original is unchanged: Data Science
```

Truth values and `None`. The type `bool` has the two values `True` and `False`, and it is a special case of `int` – `True == 1` holds. More interesting is that Python will read any value in a condition as a truth value (*truthiness*): the number 0, the empty string, the empty list, the empty dictionary and `None` count as false, everything else as true. That is why you write `if items:` rather than `if len(items) >0:`.

`None` is the only value of type `NoneType` and plays the part that `null` plays in Java: "nothing here". A function without a `return` returns `None`. You test for `None` with `is None`, not `== None` – `is` compares

identity, and there is only ever one `None` in the whole program. In data, we will meet `None` again as a sibling of `NaN` in the chapter Data cleaning.

```
print(bool(0), bool(""), bool([]), bool([0]), bool(None))   # what counts as false?
print("bool is an int:", True == 1, True + True)

x = None
if x is None:                       # not "x == None"
    print("x holds nothing")

def without_return():
    pass

print("return value without return:", without_return())

False False False True False
bool is an int: True 2
x holds nothing
return value without return: None
```

Dictionaries are maps with a finite domain; the mathematical statement $f\colon \{a,b\} \to \mathbb{R},\ a \mapsto 1,\ b \mapsto 2$ with $f(a) = 1$, for instance, is written as

```
f = {"a":1, "b":2}
if f == dict(a=1, b=2):
    print("Syntactic sugar occasionally lets us do without the quotation marks.")
if f["a"] == 1:
    print("All as planned")
print(f, "is a", type(f))
try:
    print(f["c"])
    print("the interpreter never gets to this point")
except KeyError:
    f["c"] = "You have to watch where the square and where the round brackets go"
finally:
    print(f["c"]) # but now it does!

try:
    {[1, 2]: "does not work"}       # a list is mutable
except TypeError as e:
    print("not a valid key:", e)

print({(1, 2): "tuples do work"}[(1, 2)])

Syntactic sugar occasionally lets us do without the quotation marks.
All as planned
{'a': 1, 'b': 2} is a <class 'dict'>
You have to watch where the square and where the round brackets go
not a valid key: cannot use 'list' as a dict key (unhashable type: 'list')
tuples do work
```

Under the hood a dictionary is a **hash map**: the key is turned into a number, and that number points to the slot where the value sits. That is why a lookup is fast no matter how large the dictionary is – and why only **immutable** objects may serve as keys: a key that changed afterwards would be sitting in the wrong slot. So numbers, strings and tuples are fine; lists and dictionaries are not. That is exactly what the last two lines above show.

Tuples are immutable lists of fixed length – pairs or triples, for instance. Careful: addition is not element-wise but list concatenation, and it produces a new tuple (distinct from the old one):

```
pair = (2,3)
print(pair, "is a", type(pair), "of length", len(pair))
triple = (4,5,6)
print(pair, "+", triple, "=", pair+triple)
print((0,1), "* 3 =", (0,1)*3)
```

```
(2, 3) is a <class 'tuple'> of length 2
(2, 3) + (4, 5, 6) = (2, 3, 4, 5, 6)
(0, 1) * 3 = (0, 1, 0, 1, 0, 1)
```

Sets are unordered lists, and they too are written with curly braces:

```
a_set = {3,2,1}
print(a_set)
another_set = set((2,3,1))
a_set == another_set
```

```
{1, 2, 3}
```

```
True
```

Lists are ordered and can grow. Under the hood a list in CPython is an array of pointers, swapped for a larger one when needed. As a user you notice none of this, and for questions of performance it is usually beside the point. A list comprehension iterates over one list in order to build another.

```
items = [3,1,4,1,5,9]
if 5 in items:
    print(items, "of type", type(items), "has length", len(items))
print(items + [ -1, -1, -1])
items += [0]
print(items)
triple = (1,2,3)
print(list(triple))
# here comes the list comprehension:
shifted = [x+2 for x in items]
print(shifted)
```

```
[3, 1, 4, 1, 5, 9] of type <class 'list'> has length 6
[3, 1, 4, 1, 5, 9, -1, -1, -1]
[3, 1, 4, 1, 5, 9, 0]
[1, 2, 3]
[5, 3, 6, 3, 7, 11, 2]
```

Slicing is the name for selecting part of a list with Python's colon syntax.

```
numbers = range(0,10)  # careful, this starts at 0 and 10 is not included.
print(numbers, type(numbers))
numbers = list(numbers)
print(numbers, type(numbers))
print(numbers[1:9]) # here the 9 is not included either.
print(numbers[:5] + numbers[5:])
print("negative indices work too:", numbers[ -1:])
print(numbers[: -1] + numbers[ -1:])
```

```
range(0, 10) <class 'range'>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>
[1, 2, 3, 4, 5, 6, 7, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
negative indices work too: [9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Generators are objects that generalise lists in one respect: you can ask them for the next element. But no complete list has to sit in memory (or be computed in advance), which is why generator expressions are so often the right choice for data-processing pipelines in Python. Every list, every tuple and every `range` object can be iterated over in just the same way; `range` is not a generator but a "lazy" sequence, which likewise produces its elements only on demand.

```
gen = (x + 2 for x in range(0,5))
print(gen, "has type", type(gen))
# and only now is the computation carried out:
for x in gen:
    print(x)
```

```
<generator object <genexpr> at 0x7872fe59c520> has type <class 'generator'>
2
3
4
5
6
```

Equality and identity *Equality of values* is tested with the operator `==`, *equality of references* with the keyword `is`, which amounts to the `id`s being equal.

```
print(123 == 121 + 2)
a = [1,2,3]
b = a[:] # copy
print(id(a) == id(b))
print(a is b)
print(a is [1,2,3])
print(a == b)

True
False
False
False
True
```

Membership of a type is tested with `isinstance`.

```
print(isinstance("a", str))
print(isinstance(123, bool))
print(type(123))
print(isinstance(123, object))

True
False
<class 'int'>
True
```

Weaker still than `==` is comparing the string representations (which is not the same as converting to the type `str`). Here `repr()` is meant to return a string that is as unambiguous as possible, `str()` one that is as readable as possible. Often the `repr` string can be evaluated as Python code with `eval`, which yields a copy of the object.

```
a = dict(key1=1, key2=2)
b = {"key1":'1', "key2":'2'}
print("genuinely different dict objects have different representations:")
print("(", a, "==", b, ") =", a == b)
print("but for dict objects str() is the same as repr()")
print("(", repr(a), "==", str(a), ") = ", repr(a) == str(a))

c = eval(repr(a))
print(c)

print(str(3) == str("3"))
print(repr(3) == repr("3"))
print(repr(3), "!=", repr("3"))

genuinely different dict objects have different representations:
( {'key1': 1, 'key2': 2} == {'key1': '1', 'key2': '2'} ) = False
but for dict objects str() is the same as repr()
( {'key1': 1, 'key2': 2} == {'key1': 1, 'key2': 2} ) =  True
{'key1': 1, 'key2': 2}
True
False
3 != '3'
```

Control flow, f-strings and files In Python, loops and branches look as they do in the cell below; a condition needs no brackets.

Indentation is **four spaces** – not a rule of the interpreter but of the style guide PEP 8, and every tool in the ecosystem assumes it. Mixing tabs and spaces is an error in Python 3.

The `for` loop is always a *for-each*: it runs over the elements of a sequence, never over a counter. If you need a counter, you make a sequence of numbers with `range(n)` and iterate over that – which is why `range` turns up everywhere in Python. A dictionary is iterated over with `.items()`, which hands you key and value together:

```
for i in range(3):          # for loop over a sequence
    if i == 0:
        print(i, "is zero")
    elif i % 2 == 1:
        print(i, "is odd")
    else:
        print(i, "is even")

n = 5
while n > 0:                # while loop
    n -= 2
print("n is now", n)

for colour in ["blue", "red"]:          # for-each over a list
    print(colour)

for k, v in {"a": 1, "b": 2}.items():   # over a dict: key and value
    print(k, " ->", v)
```

```
0 is zero
1 is odd
2 is even
n is now -1
blue
red
a -> 1
b -> 2
```

f-strings insert values into a string. After a colon you can specify the formatting (width, alignment, decimal places). We use them for output from the next chapter on:

```
name, share = "female", 0.7420382
print(f"share {name}: {share}")
print(f"share {name:>8s}: {share:6.2%}")   # 8 characters wide, right-aligned; percent with 2
    decimal places
print(f"{2**10 = }")                       # handy when debugging
```

```
share female: 0.7420382
share   female: 74.20%
2**10 = 1024
```

Files are opened with `open`, ideally inside a `with` block, which makes sure the file is closed again when the block ends, even if an error occurs along the way. An open text file can be iterated over line by line:

```
with open("example.txt", "w", encoding="utf-8") as file:   # "w" = write
    file.write("first line\nsecond line\n")

with open("example.txt", "r", encoding="utf-8") as file:   # "r" = read
    for number, line in enumerate(file, start=1):
        print(f"{number}: {line.strip()}")

import os
os.remove("example.txt")   # tidy up
```

```
1: first line
2: second line
```

```
class MyClass:
    def my_method(self, methodically):
        print(methodically, self)

    def __init__(self):
        self.attribute = 123

    def __repr__(self):
        return str(self.attribute)

x = MyClass()
x.my_method("test")
print(type(x))
isinstance(x, object)
```

```
test 123
<class '__main__.MyClass'>

True
```

Polymorphism goes so far that you often need not know whether a variable refers to a class or to a method – if it is `Callable`, you can put `()` after it and get something back.

Private and public Unlike Java, say, Python has no way of hiding anything (private, protected and so on) except by convention: an object attribute or a method marked with a single leading underscore is to be treated as "private".

Dictionaries and tuples are everywhere Tuples can be unpacked:

```
t = (1, 2, 3)
a, b, c = t
print(t, a, b, c)
```

```
(1, 2, 3) 1 2 3
```

Functions A function is defined with `def`, and `return` supplies its return value. The string directly under the header line is the *docstring*, the built-in documentation that `help` displays. Parameters can have default values:

```
def square(x, plus=0):
    """Square x and optionally add something to it."""
    return x**2 + plus

print(square(3))            # 9
print(square(3, plus=1))    # 10
help(square)
```

```
9
10
Help on function square in module __main__:

square(x, plus=0)
    Square x and optionally add something to it.
```

One detail that surprises people arriving from other languages: in Python, `if`, `for` and `while` do **not** open a scope of their own. A variable created inside such a block lives on after it. Only functions, classes and modules have a scope of their own – what is assigned inside a function is invisible outside it.

Conversely, a function may *read* names from outside without claiming them. The moment it assigns to such a name, that name counts as local throughout the function; if you really mean the outer variable, you have to say so with `global` (or, in nested functions, `nonlocal`). Both are rarely needed and better avoided: a function that modifies variables visible from outside is hard to test.

```
if True:
    inside = 1
print("after the if block there is", inside, " - blocks have no scope of their own")

def f():
    local = 2

f()
try:
    print(local)
except NameError as e:
    print("after the function, on the other hand, there is not:", e)

after the if block there is 1 - blocks have no scope of their own
after the function, on the other hand, there is not: name 'local' is not defined
```

A function definition can allow arbitrarily many positional arguments:

```
def method(*args):
    return args

method()
print(method(1,2,3,"four"))

def new_method(must_be_there, *args):
    return args

print(new_method(1,2,3,"four"))

(1, 2, 3, 'four')
(2, 3, 'four')

try:
    new_method()
except TypeError as e:
    print(e)

new_method() missing 1 required positional argument: 'must_be_there'
```

Keyword arguments can be handled in the same way:

```
def method(**kwargs):
    return kwargs

method()
print(method(key="value"))

try:
    method("123")
except TypeError as e:
    print(e)

{'key': 'value'}
method() takes 0 positional arguments but 1 was given
```

Altogether, then, you can allow a sequence of mandatory positional arguments, a sequence of named keyword arguments, and any number of further ones:

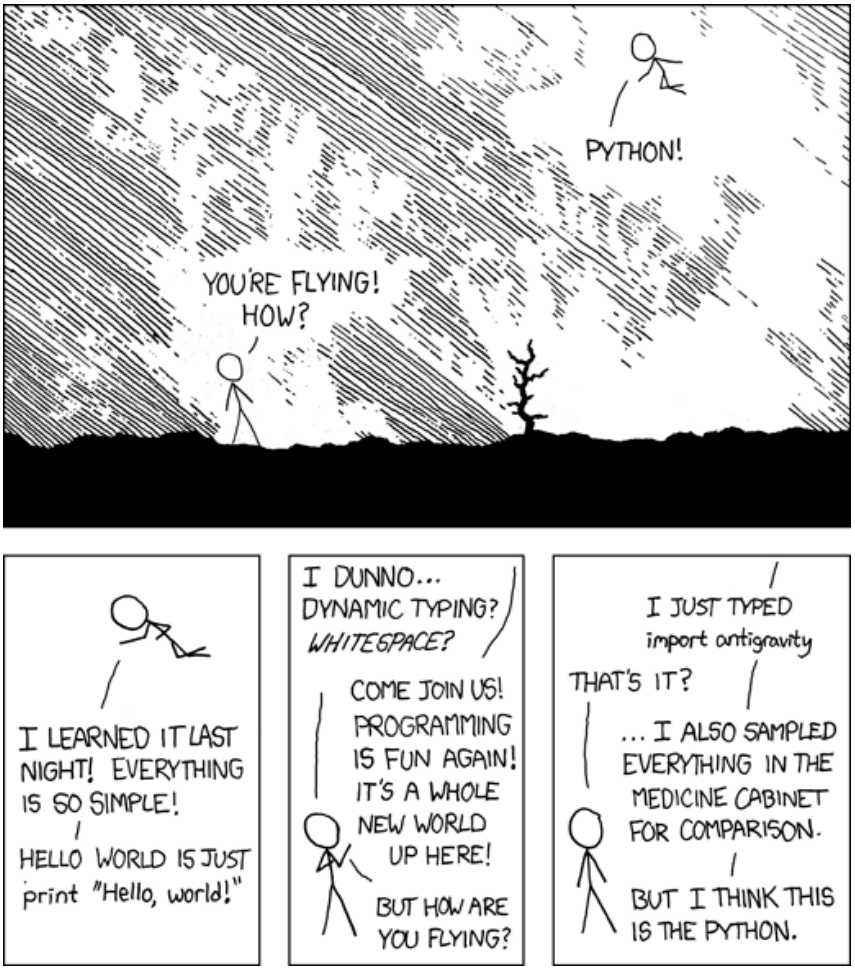
```
def method(posarg, posarg2, *args, kwarg=None, kwarg2=True, **kwargs):
    return args, kwargs

#method() # < - - - does not work

method(1, 2, 3, 4, 5, kwarg2=False, kwarg3="test")

((3, 4, 5), {'kwarg3': 'test'})
```

To pass a whole `dict` directly as keyword arguments, unpack it – and a tuple in just the same way:

```
args = (1, 2, 3, 4, 5)
kwargs = {"kwarg2": False, "kwarg3": "test"}
print(method(*args, **kwargs))

print(args)
print(*args)

((3, 4, 5), {'kwarg3': 'test'})
(1, 2, 3, 4, 5)
1 2 3 4 5
```

Object attributes, too, live in a `dict`:

```
class NewClass:
    def __init__(self):
        self.x = 1
        self.y = 2

new_object = NewClass()
print(new_object.__dict__)
new_object.__dict__["z"] = 3
new_object.z == 3

{'x': 1, 'y': 2}

True
```



Randall Munroe, xkcd 353 "Python" (CC BY-NC 2.5)

2 Discrete stochastics and Numpy

2.1 Programming with stochastics: simulating random experiments

Let us look briefly at how to simulate samples from random experiments in Python.

Python constructs used: Modules/import, Functions/def, Length/len, Sets/set, Generator expression, Range, Assertion/assert, Counting/Counter

2.1.1 Implementing a probability measure

```
from fractions import Fraction

def P(A, Omega):
    """The probability of the event A,
       given equally likely outcomes from a sample space \Omega."""
    return Fraction(len(A & Omega), len(Omega))
```

Now we use it to program a throw of a die:

```
Omega = {1, 2, 3, 4, 5, 6}
even = set((x*2 for x in range(1,4)))
print("Even numbers of pips:",even)

print("Probability of an even number of pips:", P(even, Omega))
```

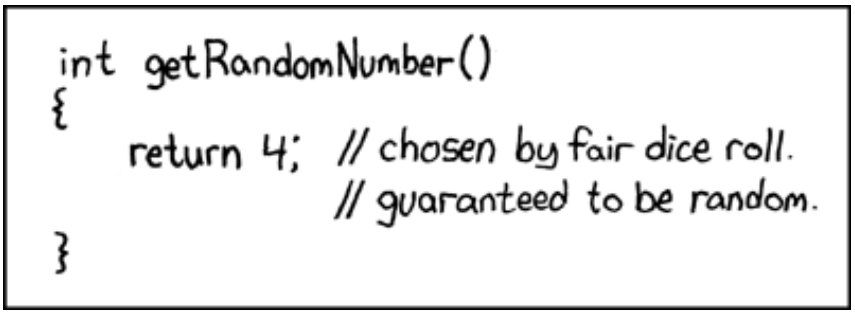
```
Even numbers of pips: {2, 4, 6}
Probability of an even number of pips: 1/2
```

That gives us the probability measure. To draw a sample, though, we still need to get the "randomness" from somewhere.

2.1.2 Importing randomness

Python's `random` module is an interface to pseudo-random numbers. The function `random.random` is a Mersenne Twister implemented directly in C. If you need "real" random numbers, for cryptography say, there is the `secrets` module. The seed of the Mersenne Twister can be set, and should be, so that random simulations are reproducible.

```
from random import random as r
help(r)
print("1", r())
print("2", r())

import random
random.seed(1)
very_random = r()
print("3", very_random)
assert very_random == 0.13436424411240122
print("4", r())
random.seed(1)
print("5", r())

Help on built -in function random:

random() method of random.Random instance
    random() -> x in the interval [0, 1).

1 0.4874556251922124
2 0.14064250428300507
3 0.13436424411240122
4 0.8474337369372327
5 0.13436424411240122
```



Randall Munroe, xkcd 221 "Random Number" (CC BY-NC 2.5)

From a (pseudo-)random number between 0 and 1 (note: possibly 0, but never 1) we can generate random throws of a die:

```
from math import floor
def transform_unit_to_die(x):
    return floor(1 + 6*x)

assert list(range(1,7)) == [transform_unit_to_die((x -1)/6)
                            for x in range(1,7)]

N = 100
throws = [transform_unit_to_die(r()) for n in range(N)]
print(throws)
```

```
[6, 5, 2, 3, 3, 4, 5, 1, 1, 6, 3, 5, 1, 3, 5, 2, 6, 6, 1, 1, 4, 6, 3, 2, 3, 1, 2, 3, 3, 2, 2, 2,
 3, 2, 1, 6, 4, 4, 2, 6, 6, 1, 2, 5, 5, 6, 3, 5, 5, 2, 4, 6, 6, 4, 4, 1, 2, 5, 3, 2, 4, 5,
5, 3, 3, 4, 5, 4, 3, 3, 1, 1, 5, 6, 4, 3, 2, 4, 6, 5, 4, 6, 2, 4, 6, 4, 3, 2, 4, 6, 1, 5, 5,
 6, 5, 5, 4, 4, 3, 1]
```

Let us count how often each number of pips came up. The *relative frequency* is the count divided by N:

```
from collections import Counter

frequencies = Counter(throws)
for pips in range(1, 7):
    print(pips, "came up", frequencies[pips], "times, relative frequency", frequencies[pips] / N
    )
```

```
1 came up 13 times, relative frequency 0.13
2 came up 16 times, relative frequency 0.16
3 came up 18 times, relative frequency 0.18
4 came up 18 times, relative frequency 0.18
5 came up 18 times, relative frequency 0.18
6 came up 17 times, relative frequency 0.17
```

To spare you programming such transformations over and over, there are also `random.randint(1,6)`, `random.choice(range(1,7))` and `random.randrange(1,7)`. Here `randint` is shorthand for the corresponding `randrange`, and `choice` is somewhat more general.

But let us put on record what we have done: starting from a uniformly distributed "random variable" $X =$`random.random` with values in $[0, 1)$, we constructed and implemented a map $t =$`transform_unit_to_die` with values in $\Omega := \{1, 2, 3, 4, 5, 6\}$, and $t(X)$ is uniformly distributed. The mathematical map $t\colon [0, 1) \to \Omega$ is a random variable, and we treat the composition $t \circ X$ as the random variable that models the die.

Now you might object that `random.random()` takes no parameter at all, so it is not a mathematical map from a domain into the set $[0, 1)$. In fact we have to imagine it as a map $X\colon \Omega' \to [0, 1)$, with some probability measure or other defined on Ω' in such a way that X induces the uniform distribution on $[0, 1)$. The set Ω' plays no concrete role for us – it is where "the randomness comes from", and the notation `random.random()` already tells us that we are not plugging in any particular element of Ω', but drawing one pseudo-randomly and plugging that into X.

2.1.3 Larger samples

The function `random.sample(population, k)` draws a sample of size k from a population (an urn) **without** replacement, so the elements drawn are pairwise distinct. For $k = 1$ this amounts to a uniform distribution on the population. The function `random.choices(population, weights=None, *, cum_weights=None, k=1)` draws from the population k times **with** replacement (and lets you assign weights instead of assembling a suitable population).

2.1.4 Looking ahead

The relative frequencies we counted in this notebook for the numbers of pips $1, \ldots, 6$ are estimates of the die's probabilities $p(1), \ldots, p(6)$. In the next chapter we will call the function p the *probability mass function* (pmf) of the random variable 'number of pips', and make the throw of a die our first statistical model.

2.2 Random variables and distributions

The typical 'interface' to probability distributions is the random variable. Unlike a bare distribution, a random variable also keeps track of which concrete values can turn up as outcomes.

2.2.1 Random variables

Let us recall the definition, now in the notation we will use from here on:

> **Definition**
> Given a set Ω with a probability measure P_Ω, a map $X\colon \Omega \to \mathcal{X}$ into a set $\mathcal{X}$ is called a **random variable** if for every event $A \subseteq \mathcal{X}$ the preimage $X^{-1}(A) \subseteq \Omega$ is an event as well.

A reminder: if Ω and $\mathcal{X}$ are finite sets, every subset counts as an event, and then every such map is a random variable. We will see later that we reach for the term random variable precisely when we are interested in expected values or higher moments.

Where does the randomness come from? Before we go on calculating, a question that is rarely asked and even more rarely answered: where in a dataset does the randomness actually come from? The short answer is that it is not in there at all. It is a decision we make.

A measurement that has already been taken is not random – the die has come to rest, the temperature has been read off, the row is sitting in the file. Collecting the data is a deterministic process too: a sensor writes down what it measures. Randomness is not something we *find* in the data; it is the language we use to talk about what we do not know about the data, or do not want to know. We treat the observed frequencies as if they were probabilities and draw conclusions from that model – so randomness is a modelling tool, not a natural substance.

All the same, there are three places where we do meet it quite concretely:

- **In the selection.** When we draw a subset from the data we have – a sample, a mini-batch, a split into training and test data – the randomness is ours. We draw at random deliberately, so that the selection adds no structure of its own that we might later mistake for a result.
- **In sampling from a model.** Drawing from a model distribution produces synthetic data. This randomness, too, is ours.
- **On the computer, not at all.** Both of these run on a deterministic device that produces only *pseudo-randomness*: a sequence of numbers that looks random but is completely determined by its starting value, the *seed*. That is not a shortcoming; it is what makes analyses reproducible (see Programming with stochastics).

The path the data travel can be told in three pictures. First, data collection: a section of the world is chosen, raw data about it accumulate, and cleaning turns the raw data into the data we actually work with.

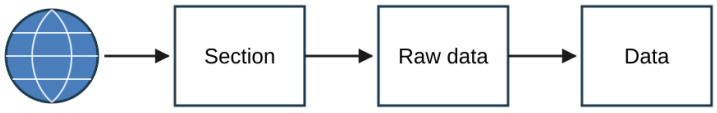


Figure 3: Data collection: from the world via a section and raw data to the data.

Then the modelling. A model is built from the data, and from the model, data can be generated in turn – synthetic data. Whether the model is any good comes down to whether its synthetic data resemble the real thing.

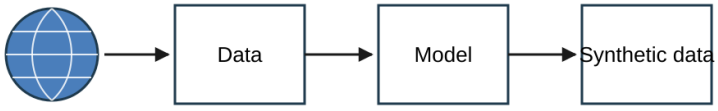


Figure 4: Modelling: from the data to the model and back to synthetic data.

And finally the path to the target variable, which is what we were after all along: the data are transformed, and from the transformed data we compute the one number we wanted to know.

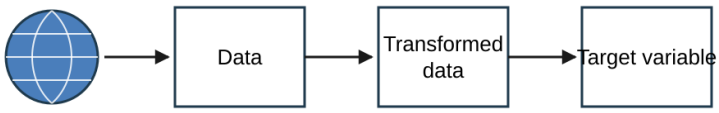


Figure 5: Analysis: from the data via transformed data to the target variable.

Example

A throw of a die, spelled out in full. The *section of the world* is a die on a table. As *raw data*, suppose a camera delivers the 3D coordinates of the eight corners at every instant: $8 \cdot 3$ numbers per frame. Once the unusable frames have been weeded out, these are the *data*. From them we compute *transformed data*: the centre and the rotation angles of the die, $3 + 3$ numbers, since that is all it takes to describe how a cube is lying. Only at the end comes the *target variable* we actually care about: the number of pips on top, a number in $\{1, \ldots, 6\}$.

The whole path from the world to the target variable is exactly what we call a random variable: a map $X \colon \Omega \to \{1, \ldots, 6\}$, where Ω is the set of possible world states. Nothing about this map is random – it is a long but perfectly unambiguous rule for calculating. The randomness lies solely in our not knowing which $\omega \in \Omega$ we are in, which is why we equip Ω with a probability measure. The statement "the die is fair" then reads

$$P(X = 6) := P_\Omega\left(\{\omega \in \Omega : X(\omega) = 6\}\right) = P_\Omega\left(X^{-1}(\{6\})\right) = \frac{1}{6}. \tag{7}$$

So the notation $P(X = 6)$ is shorthand: it means the probability of the set of all world states in which the die ends up showing a six. Exactly this construction – a measure on Ω turning, via X, into a measure on the values of X – is the subject of the next section.

Definition

Given a random variable $X \colon \Omega \to \mathcal{X}$, the probability measure on $\mathcal{X}$ given by

$$P_X \colon A \mapsto P_\Omega\left(X^{-1}(A)\right) \tag{8}$$

is the **distribution of** X.

Proposition

This really is a probability measure, always.

Proof

We check the axioms:

1. We have $X^{-1}(\mathcal{X}) = \Omega$, hence $P_X(\mathcal{X}) = P_\Omega\left(X^{-1}(\mathcal{X})\right) = P_\Omega(\Omega) = 1$.
2. Let $A_1, \ldots$ be a sequence of pairwise disjoint events in $\mathcal{X}$. Then

$$X^{-1}\left(\bigcup_{i\geq 1} A_i\right) = \{\omega \in \Omega \mid X(\omega) \in A_i \text{ for some } i\} = \bigcup_{i\geq 1} X^{-1}(A_i) \tag{9}$$

and hence

$$P_X\left(\bigcup_{i\geq 1} A_i\right) = P_\Omega\left(X^{-1}\left(\bigcup_{i\geq 1} A_i\right)\right) = P_\Omega\left(\bigcup_{i\geq 1} X^{-1}(A_i)\right) \tag{10}$$

$$= \sum_{i\geq 1} P_\Omega\left(X^{-1}A_i\right) = \sum_{i\geq 1} P_X(A_i). \tag{11}$$

Example
30 children are playing in a room: 10 of them are wearing a blue jumper, 10 a red one, 5 a yellow one and another 5 a black one. We model $\Omega =$ the set of children and $X\colon \Omega \to \{r, y, b, k\}$ as the map that sends each child to the letter standing for the colour of their jumper (r red, y yellow, b blue, k black). With that, the distribution of X is defined; it is a probability measure on the set $\{r, y, b, k\}$. For instance, we can compute:

$$P_X\left(\{b, r\}\right) = P_\Omega\left(X^{-1}\left(\{b, r\}\right)\right) = \frac{20}{30} \tag{12}$$

where we have tacitly assumed that all the children are treated alike – that is, whenever we look at an outcome of X, we pick a child uniformly at random and look at the colour of its jumper.

Example
Composing maps yields new random variables. Let X be the throw of a fair die, that is, $\Omega = \{1, \ldots, 6\}$ with the uniform distribution and X the identity.

(a) Let $g\colon \{1, \ldots, 6\} \to \{\text{even}, \text{odd}\}$ be the map that sends a number of pips to its parity. Then

$$P_g(\{\text{even}\}) = P_\Omega\left(g^{-1}(\{\text{even}\})\right) = P_\Omega(\{2, 4, 6\}) = \frac{3}{6} = \frac{1}{2}. \tag{13}$$

Here $g \circ X$ is itself a random variable – the composition of a random variable with a map always is – and its distribution is $P_{g(X)} = P_g$. One writes $g(X)$ for short and speaks of "the random variable *parity of the throw*".

(b) Let f be the map that says whether the number of pips is at least 5. For the answer "no" we decompose the preimage into elementary events and use σ-additivity:

$$P_f(\{\text{no}\}) = P_\Omega\left(\{1, 2, 3, 4\}\right) = \sum_{i=1}^{4} P_\Omega\left(\{i\}\right) = 4 \cdot \frac{1}{6} = \frac{2}{3}. \tag{14}$$

That is the typical calculation with induced measures: find the preimage, cut it into disjoint pieces and add up.

(c) Composing with $x \mapsto 2x$ gives the random variable $2X$ with values in $\{2, 4, 6, 8, 10, 12\}$, each with probability $\frac{1}{6}$. So you can calculate with random variables as you do with numbers – double them, add them, square them – and the distribution changes along with them. How expected value and variance behave under such operations is settled in the chapter Expected value and variance (linearity of the expected value).

Induced probability measure If all we are interested in is the distribution P_X, we can forget about Ω: the same distribution can be obtained from other sets Ω, and from distributions on them that are not uniform.

2.2.2 Probability mass function

In everyday work you usually calculate with probabilities not via the measure P on events but via a function $p(x)$ that assigns each possible value x its probability directly. In the discrete case, which is what this chapter is about, this function is called the *probability mass function*; its continuous counterpart, the density, we will meet in a later chapter:

Definition
A function $p\colon \Omega \to [0, 1]$ is called a *probability mass function* (*pmf*) of a distribution P on Ω if $p(\omega) = P(\{\omega\})$ holds for all $\omega \in \Omega$.

Example
In the previous example we have $P_X(\{r\}) = \frac{10}{30}$, $P_X(\{y\}) = \frac{5}{30}$, $P_X(\{b\}) = \frac{10}{30}$, $P_X(\{k\}) = \frac{5}{30}$, and therefore

$$p(r) := \frac{10}{30},\ p(y) := \frac{5}{30},\ p(b) := \frac{10}{30},\ p(k) := \frac{5}{30} \tag{15}$$

defines a pmf for P_X. One also says that p is the pmf of X. Another common name for the pmf is *discrete density*.

The pmf of the uniform distribution on Ω is $p(\omega) = \frac{1}{|\Omega|}$.

The pmf of the Dirac distribution with mass at a fixed $\xi \in \Omega$ (see The formalism of stochastics) is

$$p(\omega) = \begin{cases} 1 & \omega = \xi, \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

Both presuppose that Ω is at most countable – for uncountable Ω every individual probability $P(\{\omega\})$ is typically 0, and the density takes over from the pmf.

Example
We are expecting an incoming message 3 characters long and model the three characters as random variables X, Y and Z. We know in advance that each character can only come from a finite character set; for the three random variables we write this range as $\mathcal{X}, \mathcal{Y}$ and $\mathcal{Z}$ respectively, and the pmfs as $p(x),\ p(y),\ p(z)$ with $x \in \mathcal{X}, y \in \mathcal{Y}$ and $z \in \mathcal{Z}$.

We can also regard the whole message as a single random variable, XYZ. Then certain sequences of letters may be more probable than their constituents would suggest: if $X = s$ and $Y = o$, then $Z = s$ is surely more probable than it would be for arbitrary values of X and Y. All in all, $XYZ = sos$ is more probable than the product of the probabilities of $X = s$, $Y = o$ and $Z = s$. Writing the pmf of XYZ as $p(x, y, z)$, as usual, this says:

$$p(x, y, z) \neq p(x)p(y)p(z) \tag{17}$$

and we also say: the random variables – or rather their distributions – are not independent.

Proposition
If $X\colon \Omega \to \mathcal{X}$ is a random variable and P_Ω has a pmf $p_\Omega\colon \Omega \to [0, 1]$, then $p_X(x) := \sum_{\omega \in X^{-1}(\{x\})} p_\Omega(\omega)$ is a pmf of X.

Induced pmf

2.2.3 Probability vector

Example
If we put the children's jumper colours into an order, say $r \mapsto 1,\ y \mapsto 2,\ b \mapsto 3,\ k \mapsto 4$, we have thereby defined a map $c\colon \mathcal{X} = \{r, y, b, k\} \to \mathcal{C} = \{1, 2, 3, 4\}$. The pmf of the random variable c – or rather $c \circ X$ – is given by $p(i)$ for $i \in \mathcal{C}$, and this information can also be read as a vector.

Proposition
On an ordered finite set $\Omega = \{\omega_1, \ldots, \omega_n\}$ a probability measure is uniquely determined by the vector $p \in \mathbb{R}^n$ with $p_i := p(\omega_i)$.

Proposition
Let $\Delta_n := \{p \in \mathbb{R}^n \mid 0 \le p_i \le 1 \text{ for all } i = 1, \dots, n, \text{ and } \sum_{i=1}^n p_i = 1\}$ and let $\Omega = \{\omega_1, \dots, \omega_n\}$ be arbitrary. Then every $p \in \Delta_n$ defines a probability distribution on Ω.

Example
The uniform distribution corresponds to $p \in \Delta_n$ with $p_i = \frac{1}{n}$; that is the geometric centre of Δ_n.

The Dirac distribution with mass at $\xi \in \Omega$ corresponds to $p \in \Delta_n$ with $p_i = 1$ if $\omega_i = \xi$ and $p_i = 0$ otherwise. Geometrically that is a vertex of Δ_n, of which there are n.

A distribution given in this way by $p \in \Delta_n$ is also called a *categorical distribution*; we meet it again in the chapter The most important discrete distributions, where we also work out its expected value and its variance. Its simplest case has a name of its own.

Definition
The case $n = 2$ has a name of its own. On $\Omega = \{0, 1\}$ a probability vector is fixed by a single number, since $p_0 + p_1 = 1$ gives $p_0 = 1 - p_1$. Writing $\pi := p_1$, the corresponding distribution is called the **Bernoulli distribution with parameter** π, and one writes $X \sim \mathrm{Ber}(\pi)$ for a random variable with this distribution. (In the chapter The most important discrete distributions and in the literature this parameter is usually called p; here we write π because in this section p is already taken by the probability vector.)

Example
A coin toss is $\mathrm{Ber}(\pi)$ with π the probability of "heads"; every yes/no feature in a dataset ("survived", "clicked", "side effect occurred") can be modelled this way. The two special cases from above turn up again: $\pi = \frac{1}{2}$ is the uniform distribution, $\pi = 0$ and $\pi = 1$ are the two Dirac distributions.

Geometrically, Δ_2 is a line segment – the one joining the two vertices $(1, 0)$ and $(0, 1)$ – and π says where on this segment you are. Choosing a Bernoulli distribution therefore means choosing a point on a line segment.

So Δ_n is the space of all possible probability distributions on an n-element set. The small cases are easy to picture: Δ_2 is a line segment, Δ_3 a filled triangle (two-dimensional, but embedded in $\mathbb{R}^3$), Δ_4 a filled tetrahedron. In general Δ_n is an $(n-1)$-dimensional object with n vertices, called a **simplex**. One dimension is always lost, because the entries have to add up to 1 – exactly what we exploited for the Bernoulli distribution, where two numbers became one.

Remark
This same simplex Δ_n is the image of the *softmax* function, which neural networks use to output class probabilities: it turns n arbitrary real numbers into a vector in Δ_n. The course *Introduction to Deep Learning* takes it for granted that you know Δ_n as the space of distributions on n values.

Example
If a distribution on a finite set Ω is given to us by the vector $p \in \Delta_n$, we can characterise an event $A \subseteq \Omega$ by a vector as well: the bitmask $\chi(A) \in \{0, 1\}^n$ given by $\chi(A)_i = 1$ if $\omega_i \in A$, and $\chi(A)_i = 0$ otherwise. The probability $P(A)$ is then given by

$$P(A) = \sum_{i=1, \omega_i \in A}^{n} p(\omega_i) = \sum_{i=1}^{n} p(\omega_i)\chi(A)_i = p \cdot \chi(A). \tag{18}$$

2.2.4 From data to models

The simplest form data can take is a single number, or a sequence of N numbers. We like to call that an N-dimensional vector, or picture it as the column of a table; in statistics it is a sample of size N. Later we will also deal with tables of more than one column, where the data come as more complicated rows, but for now we stay in the simpler one-dimensional setting.

Even a single such column carries a probability distribution with it, with no modelling assumption at all: we simply count how often each value occurs.

Definition
Let $D = (x_1, \dots, x_N)$ be a dataset with values in a finite set Ω. Then

$$P(\{x\}) := \frac{|\{i : x_i = x\}|}{N}, \tag{19}$$

that is, the relative frequency of the value x in D, defines a probability measure on Ω. We call it the **distribution of the data** (also the *empirical distribution*).

That this really is a probability measure is just the statement that the relative frequencies add up to 1.

Example
A dataset with a single column x holding a colour:

i	1	2	3	4
x_i	blue	red	blue	green

Here $\Omega = \{\text{blue}, \text{red}, \text{green}\}$ and $N = 4$. The distribution of the data is $P(\{\text{blue}\}) = \frac{2}{4}$, $P(\{\text{red}\}) = \frac{1}{4}$ and $P(\{\text{green}\}) = \frac{1}{4}$, or, written as a probability vector, $p = \left(\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right)$.

If the values are coded as numbers $0, 1, 2, \dots$, you get this vector in a single line: `np.bincount(data) / len(data)` (see Numpy, section *Probability vectors*).

The following example is the special case $\Omega = \{0, 1\}$, in which a single number fixes the whole distribution.

Example
Given a sample of N coin tosses $(x_1, \dots, x_N) \in \{0, 1\}^N$ we can ask: 'what success probability π was behind this coin?'

One possible estimate is $\frac{1}{2}$, arrived at without looking at the data.

If we do look at the sample, we can compute $e := \sum_{i=1}^{N} x_i$, the number of successes. Then $N - e$ is the number of failures and $\hat{\pi} := \frac{e}{N}$ an estimate of the success probability π, provided we assume the same coin was tossed throughout.

In the language of the previous section: each toss is $\text{Ber}(\pi)$-distributed, and $\hat{\pi}$ estimates the parameter of that distribution – so we are looking for a point on the line segment Δ_2.

With that, every role in statistics has been cast, even if we will not need the names until later: the coin, which could be tossed as often as you like, is the **population**; the N tosses are a **sample** from it; and $\hat{\pi}$ is a statement about the population, obtained from the sample. In the same way a bag of sweets is the population and a handful from it the sample, from which you estimate the distribution of colours in the bag. What you know or suspect before you reach in (the coin looks normal, the bag is a standard mix) will later be called the **prior**. That the inference from sample to population is never certain, and how to put a number on that uncertainty, is the subject of the chapters Expected value and variance (Chebyshev's inequality) and Statistical tests.

The distribution of the data is not yet a model – it is only another way of writing the data down. A model only arises once an assumption is added. Suppose the colour dataset above has gained a fifth entry:

i	1	2	3	4	5
x_i	blue	red	blue	green	blue

The distribution of the data is now $\left(\frac{3}{5}, \frac{1}{5}, \frac{1}{5}\right)$. If, however, we assume that the colours are equally frequent except for one that occurs twice as often, we choose as our model the distribution $p = \left(\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right)$ – even though the observed relative frequencies differ from it. A distribution chosen in this way is called a **model distribution**. As a rule it has fewer parameters than the dataset has entries, it is smoother, and it says something even about values that do not occur in the data at all.

A model works in both directions. Determining a model distribution from data is the estimation problem the chapter Statistical models and estimators is about. In the other direction, you can draw from a model distribution and obtain **synthetic data**: a dataset that looks like a real one but is not. Synthetic data are the standard tool for testing a method of analysis, because here you know the right answer and can check whether the method finds it.

Exercise
Draw 1000 synthetic colour values for the model distribution $p = \left(\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\right)$ and determine the distribution of the data you have generated. How far does it deviate from p, and how does that change if you draw only 10 values? In the standard library the drawing is done by `random.choices(values, weights=..., k=1000)`, in Numpy by `rng.choice(values, size=1000, p=...)` (`random.choices` in the chapter Programming with stochastics, `rng.choice` in the chapter Numpy, section *Probability vectors*).

We now have two distributions on the same Ω: that of the data and that of the model. The next question is that of the quality of the model – do the data look as though they had been drawn from it? One asks *how likely is that?*, and the number that answers the question is called the **likelihood**: the probability of seeing exactly the observed data if the model is correct. A model that makes this number as large as possible is the best one by the **maximum likelihood principle**; the principle runs through the rest of the course and gets a section of its own in the chapter Statistical models and estimators. It is not the only way to put a number on the difference between two distributions: the chapter Expected value and variance measures the distance between data and synthetic data directly in the data space, and the chapter Relative entropy measures it between the distributions themselves.

That marks out the path the rest of the course takes, and it is the path of Figure 4: choose a section of the world, collect data, clean them (Data cleaning) and translate them into usable **features**, build models from those, and measure the models against the data. Everything else is the working out of these five steps. The next chapter begins with the last of them: comparing data and model in a single number.

2.3 Expected value and variance: moments, entropy and independence

2.3.1 Quantifying deviation

At the end of the previous chapter two distributions stood side by side: the distribution of the data and the model distribution. The question now facing us is how to capture their difference in a single number. The exact answer is the relative entropy, and we are saving that for a later chapter. But there is a much more direct route: instead of comparing the distributions, we compare the data themselves with what the model predicts – the real data against synthetic data, in data space, point by point. That is the route this chapter takes, and because it means taking distances between numbers, from here on we work with real-valued data.

Given two vectors $v, w \in \mathbb{R}^n$ (think of v as our data and w as what we expect), the natural quantity to look at is the mean deviation $\sum_{i=1}^{n} \frac{v_i - w_i}{n}$. The trouble with it is that positive and negative deviations cancel each other out.

Definition

The *mean absolute error* is $\texttt{mae}(v, w) := \sum_{i=1}^{n} \dfrac{|v_i - w_i|}{n}$.

For a number $c \in \mathbb{R}$ we also write $\texttt{mae}(v, c) := \texttt{mae}(v, (c)_{i=1}^{n})$.

Example

If $v = (1, 2, 3, 4)$ and $c = 2$, then $\texttt{mae}(v, c) = \sum_{i=1}^{4} \dfrac{|i - 2|}{4} = \dfrac{4}{4} = 1$.

Absolute deviation The sum without the factor $\frac{1}{n}$ is the 1-norm of the difference,

$$d_1(v, w) := \sum_{i=1}^{n} |v_i - w_i| = \|v - w\|_1 = n \cdot \texttt{mae}(v, w), \tag{20}$$

and the factor $\frac{1}{n}$ in the definition is exactly what makes `mae` independent of the length of the vector: d_1 grows with n even when the model is no worse, `mae` does not. Since comparing is almost always what we are after, we almost always take the mean. (On the role of the various norms, see also the remark on L_0, L_1 and L_2 in the chapter Descriptive statistics, as well as L1 and L2 regularisation in the chapter Regression analysis.)

Example

So far w has been constant. It need not be: let v be four measured heights in centimetres and w what a model predicted for those same four people.

i	1	2	3	4
v_i (measured)	178	160	155	174
w_i (predicted)	178	155	150	180
$v_i - w_i$	0	5	5	-6

Here $d_1(v, w) = 0 + 5 + 5 + 6 = 16$ and hence $\texttt{mae}(v, w) = \frac{16}{4} = 4$: on average the model is four centimetres off.

The mean deviation *without* the absolute value, the one we began this section with, gives $\frac{0+5+5-6}{4} = 1$ instead – deceptively small, because the error on person 4 cancels against those on persons 2 and 3. That is exactly why the absolute value is there.

The obvious next question, for a given vector v, is which vector w has the smallest mean absolute error. Without further conditions on w, though, the answer is immediate: $w = v$. We will come back to this question later; for now we content ourselves with asking which $c \in \mathbb{R}$ minimises $\texttt{mae}(v, c)$. Graphically: put the numbers from 1 to n on the x-axis and plot the values v_i above them. A choice of $c \in \mathbb{R}$ is then a horizontal line, and we are asking which such line has the smallest combined distance to all the points.

Example
Take the constant model $c = 165$ for the heights $v = (178, 160, 155, 174)$, that is, the claim "everybody is 165 cm tall". The deviations are $v_i - c = (13, -5, -10, 9)$, so

$$d_1(v, 165) = 13 + 5 + 10 + 9 = 37, \qquad \texttt{mae}(v, 165) = \frac{37}{4} = 9.25. \tag{21}$$

Is 165 the best choice? That is precisely the question of the minimum, and as we shall see in a moment, it has a surprisingly simple answer.

Definition
$\texttt{median}(v) := \arg\min_{c \in \mathbb{R}} \texttt{mae}(v, c)$ is "the" *median* of v.

So the median is defined as a number $c \in \mathbb{R}$ at which $\texttt{mae}(v, c)$ is minimal. That is, for all c we have $\texttt{mae}(v, c) \geq \texttt{mae}(v, \texttt{median}(v))$. Careful: defined this way, several numbers can be "the" median. That is not a bug, it is a feature.

This definition may be unfamiliar, but it has the following property:

Proposition
If $n \geq 3$ and $v \in \mathbb{R}^n$ is sorted, that is $v_i < v_j$ for $i < j$, then $v' := (v_2, \ldots, v_{n-1}) \in \mathbb{R}^{n-2}$ defines a new (shorter) vector such that for all $c \in \mathbb{R}$ with $v_1 \leq c \leq v_n$:

$$\texttt{mae}(v, c) = \texttt{mae}(v', c) + \frac{v_n - v_1}{n}.$$

Since the second summand does not depend on c, it plays no part in finding the minimum. Applying this inductively, all that remains is to determine $\texttt{mae}(v, c)$ for vectors of length 1 and 2. We have $\texttt{mae}((v_1), c) = |v_1 - c|$, which is minimal for $c = v_1$, and $\texttt{mae}((v_1, v_2), c) = \frac{v_2 - v_1}{2}$, which is minimal for every c between v_1 and v_2. So the median of a sorted list depends only on the middle value, or on the two middle values. The usual definition settles the latter case by choosing $\texttt{median}((v_1, v_2)) = \dfrac{v_1 + v_2}{2}$, and that is what we will do from here on. Convince yourself that for every c with $a \leq c \leq b$ the mean distance to (a, b) is $\frac{b-a}{2}$, so the vector $v = (a, b)$ has many medians.

Proof

$$\begin{aligned}
\texttt{mae}(v, c) &= \sum_{i=1}^{n} \frac{|v_i - c|}{n} \\
&= \frac{|v_1 - c|}{n} + \sum_{i=2}^{n-1} \frac{|v_i - c|}{n} + \frac{|v_n - c|}{n} \\
&= \sum_{i=2}^{n-1} \frac{|v_i - c|}{n} + \frac{-(v_1 - c) + (v_n - c)}{n} \\
&= \texttt{mae}(v', c) + \frac{v_n - c + c - v_1}{n}.
\end{aligned} \tag{22}$$

The median, then, reflects the data rather poorly – it is completely independent of most of them! – so we need a better measure. *Idea*: square the individual error terms in the sum, so that large errors contribute a larger share of it. Conveniently, squaring also spares us the absolute value: it takes care of the sign by itself.

Definition
For vectors $v, w \in \mathbb{R}^n$ the *mean squared error* is defined as

$$\texttt{mse}(v, w) := \sum_{i=1}^{n} \frac{(v_i - w_i)^2}{n} \tag{23}$$

and for $c \in \mathbb{R}$ we write $\texttt{mse}(v, c) := \texttt{mse}(v, (c)_{i=1}^{n})$.

Squared deviation As with the absolute deviation, the sum without $\frac{1}{n}$ is a norm, this time the squared 2-norm:

$$d_2(v, w)^2 := \sum_{i=1}^{n} (v_i - w_i)^2 = \|v - w\|_2^2 = n \cdot \texttt{mse}(v, w). \tag{24}$$

Compared with `mae`, the `mse` comes in the square of the unit – for heights in centimetres that is square centimetres, which is hard to interpret. So the square root is usually taken at the end; for the constant model $w = \texttt{mean}(v)$ that gives the sample standard deviation of the next subsection.

Proposition
For a vector $v \in \mathbb{R}^n$, the (arithmetic) *mean* of v minimises the mean squared error:
$\texttt{mean}(v) := \sum_{i=1}^{n} \frac{v_i}{n} = \arg\min_{c \in \mathbb{R}} \texttt{mse}(v, c)$.

Proof
We write an arbitrary $c \in \mathbb{R}$ as $c = \texttt{mean}(v) + \delta$ and show that $\texttt{mse}(v, c)$ is minimal for $\delta = 0$. To that end we expand $(v_i - \texttt{mean}(v) - \delta)^2 = (v_i - \texttt{mean}(v))^2 - 2\delta(v_i - \texttt{mean}(v)) + \delta^2$ and split the sum accordingly:

$$\begin{aligned}
\texttt{mse}(v, c) = \texttt{mse}(v, \texttt{mean}(v) + \delta) &= \sum_{i=1}^{n} \frac{(v_i - (\texttt{mean}(v) + \delta))^2}{n} \\
&= \frac{1}{n} \sum_{i=1}^{n} \Big((v_i - \texttt{mean}(v))^2 - 2\delta \, (v_i - \texttt{mean}(v)) + \delta^2 \Big) \\
&= \left(\frac{1}{n} \sum_{i=1}^{n} \big((v_i - \texttt{mean}(v))^2 \big) \right) - 2\delta \left(\sum_{i=1}^{n} \frac{v_i}{n} - \texttt{mean}(v) \right) + \delta^2 \\
&= \texttt{mse}(v, \texttt{mean}(v)) - 2\delta(\texttt{mean}(v) - \texttt{mean}(v)) + \delta^2 \\
&= \texttt{mse}(v, \texttt{mean}(v)) + \delta^2
\end{aligned} \tag{25}$$

and this expression is minimal only for $\delta = 0$, because $\delta^2 \geq 0$.

Example
The same four heights, this time with squared errors. For the model predictions $w = (178, 155, 150, 180)$,

$$d_2(v, w)^2 = 0^2 + 5^2 + 5^2 + (-6)^2 = 86, \qquad \texttt{mse}(v, w) = \frac{86}{4} = 21.5. \tag{26}$$

The comparison of the constant models is more interesting. The mean is $\texttt{mean}(v) = \frac{667}{4} = 166.75$. Sorted, the vector reads $(155, 160, 174, 178)$, and by the argument above every number between 160 and 174 is a median; the usual convention picks the midpoint, so $\texttt{median}(v) = 167$. The two error measures really do crown different winners:

c	$\texttt{mae}(v, c)$	$\texttt{mse}(v, c)$
165	9.25	93.75
$\texttt{median}(v) = 167$	9.25	90.75
$\texttt{mean}(v) = 166.75$	9.25	**90.6875**

The middle column says the same thing three times: every number between 160 and 174 minimises the `mae`, 165 included – the median simply is not unique. In the right-hand column, by contrast, exactly one number wins, and that is the mean. Choosing an error measure therefore means choosing a best model.

Definition
Let $v \in \mathbb{R}^n$. We call the square root of the mean squared error about the mean the `rmse` (root mean squared error, also the sample standard deviation):
$\texttt{rmse}(v) = \sqrt{\texttt{mse}\,(v, \texttt{mean}(v))}$.
It is also written $\sigma(v)$, the *standard deviation* of the sample v. The notation matches $\sigma(X)$ for random variables further down.

Sample standard deviation For computation the definition is awkward: you first have to determine the mean and then go over all the entries a second time. The following rearrangement needs only the sum of the entries and the sum of their squares:

Lemma
(Shortcut formula)
For $v \in \mathbb{R}^n$,

$$\texttt{rmse}(v)^2 = \texttt{mse}\,(v, \texttt{mean}(v)) = \texttt{mean}(v^2) - \texttt{mean}(v)^2, \tag{27}$$

where $v^2 := \left(v_i^2\right)_{i=1}^n$ is to be read componentwise.

Proof
With $m := \texttt{mean}(v)$,

$$\texttt{mse}(v, m) = \frac{1}{n}\sum_{i=1}^{n}\left(v_i^2 - 2mv_i + m^2\right) = \texttt{mean}(v^2) - 2m \cdot \texttt{mean}(v) + m^2 = \texttt{mean}(v^2) - m^2. \tag{28}$$

Proposition
For $v \in \mathbb{R}^n$, $v \neq 0$, with $\texttt{mean}(v) = 0$ (a *centred* sample, for instance $v := w - (\texttt{mean}(w))_{i=1}^n$ for an arbitrary $w \in \mathbb{R}^n$) we have $\texttt{rmse}(v)^2 = \texttt{mse}(v, 0) = \frac{1}{n}\sum_{i=1}^n v_i^2$, and the vector $v' := \left(\frac{v_i}{\texttt{rmse}(v)}\right)_{i=1}^n$ satisfies $\texttt{mean}(v') = 0$ and $\texttt{rmse}(v') = 1$ (a *normalised* sample).

Proof
This is a direct calculation. That the centred $v = w - (\texttt{mean}(w))_{i=1}^n$ really has mean 0 is just the linearity of the mean: $\texttt{mean}(v) = \texttt{mean}(w) - \texttt{mean}(w) = 0$. Since $\texttt{mean}(v) = 0$, we have $\texttt{rmse}(v)^2 = \texttt{mse}(v, \texttt{mean}(v)) = \texttt{mse}(v, 0) = \frac{1}{n}\sum_{i=1}^n v_i^2$, which is the shortcut formula with $m = 0$. For v' put $s := \texttt{rmse}(v)$; $s > 0$ because $v \neq 0$. Then $\texttt{mean}(v') = \frac{1}{n}\sum_{i=1}^n \frac{v_i}{s} = \frac{\texttt{mean}(v)}{s} = 0$ and therefore $\texttt{rmse}(v')^2 = \texttt{mse}(v', 0) = \frac{1}{n}\sum_{i=1}^n \frac{v_i^2}{s^2} = \frac{\texttt{rmse}(v)^2}{s^2} = 1$.

Remark
Centring deserves a notation of its own. For $v \in \mathbb{R}^n$ we write

$$\tilde{v} := v - (\texttt{mean}(v))_{i=1}^{n}, \qquad \text{that is, } \tilde{v}_i = v_i - \texttt{mean}(v), \tag{29}$$

and call $\tilde{v}$ the *centred* v. With the shortcut formula from above, the sample standard deviation and the variance can then be written without a summation sign:

$$\sigma(v) = \texttt{rmse}(v) = \frac{1}{\sqrt{n}} \|\tilde{v}\|, \qquad \sigma(v)^2 = \frac{1}{n}\, \tilde{v} \cdot \tilde{v} = \frac{1}{n}\, \tilde{v}^\top \tilde{v}. \tag{30}$$

In the language of linear algebra, then, the sample standard deviation is nothing but the Euclidean length of the centred vector divided by $\sqrt{n}$, and the variance is the inner product of that vector with itself divided by n.

This view carries further than it first appears. In the chapter Covariance and dependence it will turn out that the covariance of two random variables behaves exactly like an inner product – the variance is then the special case "vector with itself", and the correlation coefficient is the cosine of the angle between two centred vectors.

Information content Whether a vector $v \in \mathbb{R}^n$ carries information for us depends on how much we already know about v. If we know, for example, that v is a sample of a random variable uniformly distributed on $\{0, 1\}$, we will not be surprised when v looks like one. Every entry of v corresponds to exactly one bit of information. Suppose instead that the random variable is not uniform but follows, on the numbers $\{1, \ldots, 26\}$, the frequency of the letters of the German language, say (with 1 for "a", 26 for "z" and so on). Then an entry of v potentially carries less information. The "e" is so frequent that we are hardly surprised to see one. An "x", on the other hand, would surprise us a great deal – but then it does not come up very often either. The average information content is therefore lower than for a uniform distribution.

Measures of central tendency Median and mean are traditionally joined by the **mode**, the most frequent value. It fits the same pattern: the median is the best constant model for the absolute error, the mean for the squared one, and the mode for the *counting error*, which counts every deviation the same, however large it is. For a nominal feature, say the `Survived` column of the Titanic list, that is the only error that makes sense; the best constant model is then the most frequent class, and the proportion of the time it is right is its *accuracy* (see Titanic EDA). Here is a short explainer video from Crash Course Statistics on mean, median and mode. The video that follows it in the playlist discusses the related notions for the spread of data (the deviation we started with).

2.3.2 Moments of a random variable

From the mean to the expected value So far we have computed with samples: $\texttt{mean}(v)$ is a number that comes from n concrete measurements. Now we reverse the point of view and ask which number belongs to a *distribution*, that is, to the model the data might have come from. The route there is short, and it explains why the expected value looks exactly the way it does.

Let X be a random variable with finitely many values $y_1, \ldots, y_k$ and distribution $p = (p_1, \ldots, p_k)$, so $p_j = P(X = y_j)$. Suppose for a moment that the p_j are rational numbers with a common denominator N, say $p_j = \frac{a_j}{N}$ with $a_1 + \cdots + a_k = N$. Then there is a sample of size N that hits this distribution *exactly*: the one in which the value y_j occurs precisely a_j times. We call it the **idealised sample** $\tilde{D}$ for p – the sample you would get if chance kept to the probabilities perfectly, just once.

We can write down the mean of this idealised sample directly, since we know how often each value occurs:

$$\texttt{mean}(\tilde{D}) = \frac{1}{N} \sum_{j=1}^{k} a_j y_j = \sum_{j=1}^{k} \frac{a_j}{N} y_j = \sum_{j=1}^{k} p_j y_j. \tag{31}$$

On the right there is no sample any more, only the distribution. With that, the assumption that the p_j are rational has become superfluous too: the right-hand side makes sense for every distribution, and the detour through $\tilde{D}$ served only to find the formula. This quantity is exactly what we call the expected value of X.

Example
A fair die has $p = \left(\frac{1}{6}, \dots, \frac{1}{6}\right)$, so $N = 6$ and $a_j = 1$ for all j. The idealised sample is $\tilde{D} = (1, 2, 3, 4, 5, 6)$ – every face exactly once – with

$$\texttt{mean}(\tilde{D}) = \frac{1+2+3+4+5+6}{6} = \frac{21}{6} = 3.5. \tag{32}$$

So the expected value of a fair die is 3.5, a number the die itself never shows.

A loaded die with $p = \left(\frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{1}{10}, \frac{5}{10}\right)$ has $N = 10$, the idealised sample $(1, 2, 3, 4, 5, 6, 6, 6, 6, 6)$ and hence the expected value $\frac{45}{10} = 4.5$.

The special case $k = 1$ is the acid test: a distribution that takes a single value y_1 with probability 1 – the Dirac measure δ_{y_1} from the chapter The formalism of stochastics – has the idealised sample $(y_1, \dots, y_1)$ and so the expected value y_1. A constant model and the value it predicts are the same thing. In the language of the previous section: the best constant model for the squared error was the mean of the data – and read as a distribution, a constant model is a Dirac measure.

Example
On $\Omega = \{0, 1, \dots, n-1\}$ – exactly what `range(n)` enumerates in Python – let P be the uniform distribution. For $Y(\omega) := \omega$,

$$\mathbb{E}(Y) = \sum_{\omega=0}^{n-1} \frac{\omega}{n} = \frac{1}{n} \cdot \frac{(n-1)n}{2} = \frac{n-1}{2}, \tag{33}$$

whereas for $X(\omega) := \omega + 1$ we get $\mathbb{E}(X) = \mathbb{E}(Y) + 1 = \frac{n+1}{2}$ (that is the linearity we come to further down). For $n = 6$ these are 2.5 and 3.5: whether you start counting the faces of a die at 0 or at 1 makes exactly one pip of difference. In simulations this is the most common source of off-by-one errors.

With that we can set the formula down as a definition. It works word for word when Ω is countably infinite; the finite sum then becomes a series, which need no longer converge.

Definition
Let $X \colon \Omega \to \mathbb{R}$ be a discrete real random variable (so Ω a discrete probability space, at most countable). Then the *expected value* of X is defined as

$$\mathbb{E}X := \mathbb{E}(X) := \sum_{\omega \in \Omega} X(\omega) P(\{\omega\}). \tag{34}$$

If you would rather not work "in Ω", you can rearrange this sum and sum over all the values $\{x_i\}_{i \in I}$ that X takes: $\mathbb{E}(X) = \sum_{i \in I} x_i P(X = x_i)$.

Expected value, discrete So the expected value is the mean of the values of X, weighted by the probabilities.

For finite Ω the sum is finite and always exists. If Ω is countably infinite, the sum is a series, and a series need not converge: there are random variables for which no expected value can be determined. An example: on $\Omega = \{1, 2, 3, \dots\}$, $P(\{k\}) := \frac{6}{\pi^2 k^2}$ defines a probability measure (because $\sum_{k \geq 1} \frac{1}{k^2} = \frac{\pi^2}{6}$), and for $X(k) := k$ the series $\sum_{k \geq 1} k \cdot \frac{6}{\pi^2 k^2} = \frac{6}{\pi^2} \sum_{k \geq 1} \frac{1}{k}$ is the harmonic series, which diverges. This random variable has no expected value.

What we can say is this: if X takes only finitely many values, then not only does the expected value exist but so do all the higher moments $\mathbb{E}(X^k)$ – they are finite sums, and those always converge. With countably infinitely many values, by contrast, you have to check each time. An example that is more than a mathematical curiosity: in the *St Petersburg paradox* a coin is tossed until heads first comes up, and if that takes k tosses, 2^k euros are paid out. The probability of that is 2^{-k}, so the expected value is $\sum_{k \geq 1} 2^k \cdot 2^{-k} = \sum_{k \geq 1} 1 = \infty$. A game with infinite expected value – and yet nobody would pay a high stake to play it.

For continuous random variables, which take uncountably many values, an integral over a density f takes the place of the sum: $\mathbb{E}(X) = \int x f(x)\,dx$. We give this definition in the chapter Continuous distributions, once we know about densities. Everything we establish about the expected value in this chapter holds for continuous random variables in just the same way, with integrals in place of sums.

Note
Digression (not part of the course proper): the sum and the density integral are two special cases of a single definition that works for arbitrary random variables.

Definition
Let $X\colon \Omega \to \mathbb{R}$ be a real random variable; then the *expected value* is defined as
$\mathbb{E}X := \mathbb{E}(X) := \int_\Omega X dP = \int_\Omega X(\omega) dP(\omega) \in \mathbb{R}$
that is, as the integral of the map X with respect to the measure P.

This integral (the Lebesgue integral with respect to P) need not exist, which again means that there are random variables for which no expected value can be determined. For discrete Ω it reduces to the sum $\sum_{\omega\in\Omega} X(\omega)P(\{\omega\})$ from above, and if X has a density f it becomes the integral $\int x f(x)\,dx$. With this general definition the rules of calculation below can be proved for all random variables at once; that is why measure theory sets things up this way.

Digression: the expected value in general The expected value and the samples of a random variable are intimately connected:

Theorem
Law of large numbers
An infinite sequence $(X_i)_{i\in\mathbb{N}}$ of random variables $\Omega \to \mathbb{R}$ is said to satisfy a *strong law of large numbers* if the mean-deviation random variables $\overline{X_n^*} := \frac{1}{n}\sum_{i=1}^n (X_i - \mathbb{E}(X_i))$ converge to 0, and do so P-almost surely (that is, $P(\lim_{n\to\infty} \overline{X_n^*} = 0) = 1$).

If the X_i are independent and identically distributed (*iid* for short: every X_i has the same distribution as a fixed X, and we write $X_i \sim X$) and the expected value $\mathbb{E}(X)$ exists, then the X_i satisfy a strong law of large numbers. That is, $\overline{X_n^*} = \overline{X_n} - \mathbb{E}X$ – the deviation of the arithmetic mean of a sample of X of size n from the expected value – converges P-almost surely to 0. Read the other way round: the probability that the sample mean deviates from the expected value for arbitrarily large n is 0.

P-almost sure convergence also implies convergence in probability: $\lim_{n\to\infty} P\left(|\overline{X_n^*}| \geq \epsilon\right) = \lim_{n\to\infty} P\left(|\overline{X_n} - \mathbb{E}X| \geq \epsilon\right)$ 0 for every $\epsilon > 0$.

In what follows we shall not rack our brains any further over the notions of convergence in stochastics (that can always be caught up on later, and depending on the application it is worth doing). Instead, in the usual tradition, we take with us the reassurance that the expected value is a suitable quantity for analysing a sample expectantly. The larger the sample, the smaller the deviation of the sample mean from the expected value should be – so the law would have it. Put the other way round: for every prescribed bound $\epsilon > 0$ there is a sample size n such that the sample mean of a randomly drawn sample almost always deviates from the expected value by less than ϵ.

That closes the circle back to the idealised sample at the start of the section: there we pretended that a sample kept to the probabilities exactly, just to arrive at the formula at all. The law of large numbers says that real samples do so arbitrarily well for large n – so the detour was no detour.

Lemma
Rules of calculation for the expected value:

1. Linearity: the expected value is linear in X, so for $\alpha, \beta \in \mathbb{R}$ and random variables $X, Y\colon \Omega \to \mathbb{R}$,

$$\mathbb{E}(\alpha X + \beta Y) = \alpha \mathbb{E}X + \beta \mathbb{E}Y. \tag{35}$$

2. Monotonicity: the expected value is monotone in X, so for $X \leq Y$ we have $\mathbb{E}X \leq \mathbb{E}Y$.
3. σ-additivity: for a sequence $X_i \geq 0$ we have $\mathbb{E}(\sum_{i\geq 1} X_i) = \sum_{i\geq 1} \mathbb{E}(X_i)$.
4. Product rule: if X and Y are independent, then $\mathbb{E}(XY) = \mathbb{E}(X)\mathbb{E}(Y)$.

Without proof

We have not yet explained what *independent* means (independently of the pmfs in the discrete case); that comes further down.

Definition
Let $X : \Omega \to \mathbb{R}$ be a random variable. Then
$\mathbb{V}X := \mathbb{V}(X) := Var(X) := \mathbb{E}\left((X - \mathbb{E}X)^2\right)$ is called the *variance* of X.

Proposition
(Shortcut formula)
We have $\mathbb{V}(X) = \mathbb{E}\left(X^2\right) - (\mathbb{E}X)^2$.

Proof
We have $(X - \mathbb{E}X)^2 = X^2 - 2X\mathbb{E}X + (\mathbb{E}X)^2$, and so by the linearity of the expected value

$$\begin{aligned} \mathbb{V}(X) &= \mathbb{E}\left((X - \mathbb{E}X)^2\right) \\ &= \mathbb{E}\left(X^2\right) - 2\mathbb{E}(X)\,\mathbb{E}(X) + (\mathbb{E}X)^2\mathbb{E}(1) \\ &= \mathbb{E}\left(X^2\right) - (\mathbb{E}X)^2. \end{aligned} \tag{36}$$

Variance This is the counterpart of the shortcut formula $\texttt{rmse}(v)^2 = \texttt{mean}(v^2) - \texttt{mean}(v)^2$ for samples above. For a centred random variable, one with $\mathbb{E}(X) = 0$, this simply gives $\mathbb{V}(X) = \mathbb{E}(X^2)$.

Definition
More generally, $\mathbb{E}(X^n)$ is called the n-th *moment* and $\mathbb{E}((X - \mathbb{E}X)^n)$ the n-th *central moment*. So the expected value is the first moment and the variance the second central moment. The third central moment divided by σ^3 is the *skewness*, and the fourth divided by σ^4 the *kurtosis*; the division makes both unit-free and independent of scale. The skewness measures how symmetric the distribution is about the expected value. $\sigma(X) := \sqrt{\mathbb{V}(X)}$ is called the *standard deviation* of X.

The variance helps us judge how probable a given deviation from the expected value is. The key is an inequality that, to begin with, uses only the expected value:

Theorem
(Markov's inequality)
For X a real random variable with $X \geq 0$, and $c > 0$,

$$P(X \geq c) \leq \frac{\mathbb{E}X}{c}. \tag{37}$$

Proof
Let $1_{\{X \geq c\}} \colon \Omega \to \{0, 1\}$ be the indicator function of the event $\{X \geq c\}$, so 1 on $\{X \geq c\}$ and 0 elsewhere. Then $c \cdot 1_{\{X \geq c\}} \leq X$ holds pointwise: where the indicator is 1 we have $X \geq c$, and where it is 0 we have $X \geq 0$ by assumption. The expected value of an indicator function is the probability of the event – in the discrete case, $\mathbb{E}(1_A) = \sum_{\omega \in A} P(\{\omega\}) = P(A)$. By the linearity and monotonicity of the expected value (rules of calculation above),

$$c \cdot P(X \geq c) = \mathbb{E}\left(c \cdot 1_{\{X \geq c\}}\right) \leq \mathbb{E}X, \tag{38}$$

and dividing by $c > 0$ gives the claim.

Markov's inequality is crude, but it immediately yields the inequality we actually want:

Theorem
(Chebyshev's inequality)
For X a real random variable and $c > 0$,

$$P\left(|X - \mathbb{E}X| \geq c\right) \leq \frac{\mathbb{V}X}{c^2}. \tag{39}$$

Proof
The random variable $Y := (X - \mathbb{E}X)^2$ is non-negative, and by the definition of the variance $\mathbb{E}Y = \mathbb{V}X$. Markov's inequality for Y with the bound $c^2 > 0$ gives

$$P\left((X - \mathbb{E}X)^2 \geq c^2\right) \leq \frac{\mathbb{E}Y}{c^2} = \frac{\mathbb{V}X}{c^2}. \tag{40}$$

Because $c > 0$, we have $(X - \mathbb{E}X)^2 \geq c^2$ exactly when $|X - \mathbb{E}X| \geq c$. The two events are therefore the same, and that is the claim.

The special case $c = \sigma$ is trivial: the right-hand side is just 1, and every probability is ≤ 1. The special case $c = k\sigma$ with $k \in \mathbb{N}$, however, simplifies to

$$P\left(|X - \mathbb{E}X| \geq k\sigma\right) \leq \frac{1}{k^2}. \tag{41}$$

So deviations of 2σ have probability $\leq \frac{1}{4}$, and deviations of 3σ only $\leq \frac{1}{9}$. Measured in multiples of σ, then, the probability of a deviation that large is roughly inversely proportional to the square – which is just what the inequality above says. That also gives us a criterion for weeding faulty data points out of a dataset: if deviations larger than $k\sigma$ with $k > 1$ turn up more often than expected, then those data points at least do not belong to the distribution we expect.

The most important application of the inequality is to the sample mean itself. With it we can actually prove the law of large numbers from above, in a weaker form and with exactly the tools we now have:

Theorem
(Weak law of large numbers)
Let $X_1, \ldots, X_n$ be independent and identically distributed like X (iid for short; every X_i has the same distribution as X), and let X have finite variance. Then for the sample mean $\overline{X_n} := \frac{1}{n}\sum_{i=1}^{n} X_i$ and every $c > 0$

$$P\left(|\overline{X_n} - \mathbb{E}X| \geq c\right) \leq \frac{\mathbb{V}X}{nc^2} \xrightarrow{n \to \infty} 0. \tag{42}$$

Proof
By the linearity of the expected value, $\mathbb{E}\overline{X_n} = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}X_i = \mathbb{E}X$, since every X_i has the same expected value as X. By the rules of calculation for the variance (section *Rules of calculation for the variance* further down: $\mathbb{V}(aY) = a^2\,\mathbb{V}Y$, and the variance of a sum of independent random variables is the sum of the variances),

$$\mathbb{V}\overline{X_n} = \frac{1}{n^2}\mathbb{V}\left(\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}\sum_{i=1}^{n}\mathbb{V}X_i = \frac{n\,\mathbb{V}X}{n^2} = \frac{\mathbb{V}X}{n}. \tag{43}$$

Chebyshev's inequality for $\overline{X_n}$ now gives $P(|\overline{X_n} - \mathbb{E}X| \geq c) \leq \mathbb{V}\overline{X_n}/c^2 = \mathbb{V}X/(nc^2)$, and this expression tends to 0 as $n \to \infty$.

This is exactly the convergence in probability that we deduced above from the (unproved) strong law of large numbers: the weak law is the form we can prove with the means at hand; the strong law, with its almost sure convergence, needs more machinery but does without the assumption of finite variance.

The weak law is also the law of large numbers with a number attached: the probability of a deviation by more than c falls like $1/n$ – or, read differently, the typical error of the sample mean shrinks like $1/\sqrt{n}$. To halve the error you need four times the sample. The bound is crude; the chapter The normal distribution supplies a sharper one with the central limit theorem, and the chapter Statistical tests turns that into the recipe for inferring from sample to population.

Remark
In deep learning the true loss of a model is the expected value of the loss function over the distribution of the data, and it cannot be computed, because that distribution is unknown. A mini-batch of a few data points supplies a sample mean of the loss instead: cheap, but with high variance. That is exactly the situation of the weak law of large numbers, and $\mathbb{V}X/n$ says how much noise a batch of size n brings with it. The course *Introduction to Deep Learning* builds on this (stochastic gradient descent).

Applying the law of large numbers to the random variable $(X - \mathbb{E}(X))^2$ shows that the sample standard deviation converges to the standard deviation as well. So the sample standard deviation is a good substitute when the true variance of the random variable cannot be determined.

Interlude: the logarithm The next section computes with logarithms throughout. Since school may be a while ago, here is what you need, in half a page.

Definition
For $b > 0$, $b \neq 1$, the *logarithm to base b* is the inverse of exponentiation: from $b^x = y$ it follows that $x = \log_b(y)$, and conversely. It is defined for $y > 0$.

Three bases have names of their own: $\ln := \log_e$ (the *natural* logarithm, to base $e \approx 2.718$), $\mathrm{lb} := \log_2$ (the *binary* logarithm, also written ld) and $\log_{10}$ (the *common* one).

The logarithm counts orders of magnitude. To base 10 you can read that off directly: $\log_{10}(100) = 2$ and $\log_{10}(1000) = 3$, and because the logarithm is strictly increasing, $\log_{10}(y)$ lies between 2 and 3 for every number y between 100 and 1000. The number of decimal digits of a natural number y is therefore $\lfloor \log_{10}(y) \rfloor + 1$ – and to base 2, likewise, the number of binary digits.

The rules of calculation all follow from $e^{\ln(y)} = y$ and $\ln(e^x) = x$ together with the power law $(e^u)^v = e^{uv}$:

$$\begin{aligned} b^x &= \left(e^{\ln b}\right)^x = e^{\ln(b)\,x}, \\ \ln\left(b^x\right) &= x\ln(b), \\ \ln(y) &= \ln\left(b^{\log_b(y)}\right) = \ln(b)\log_b(y), \\ \text{so}\quad \log_b(y) &= \frac{\ln(y)}{\ln(b)}. \end{aligned} \tag{44}$$

The last line is the *change-of-base formula*: logarithms to different bases differ only by a constant factor. That is why the choice of base for the entropy is a question of units, not of substance – to base 2 you measure in **bits**, to base e in **nats**, and the two differ by the factor $\ln(2) \approx 0.693$.

Example
The number 77 is 1001101_2 in binary, that is $64 + 8 + 4 + 1$, and so needs 7 binary digits. Check: $\log_2(77) = \frac{\ln 77}{\ln 2} \approx 6.27$, and $\lfloor 6.27 \rfloor + 1$ gives the expected 7 digits. Read the other way round: with 7 bits you can distinguish $2^7 = 128$ different things, with 6 bits only 64 – too few for 77.

In numpy the three functions are called `np.log` (natural, *not* common – a frequent source of errors), `np.log2` and `np.log10`.

2.3.3 Entropy

Note
Looking ahead (not part of the course proper): where the word comes from. The notion of entropy comes from thermodynamics, and the physical picture goes a long way towards explaining why the formula below looks the way it does. You do not need this section for the course.

Picture two identical containers with the same number of gas particles in them. In the first the particles are evenly distributed, in the second they are all huddled in one corner.

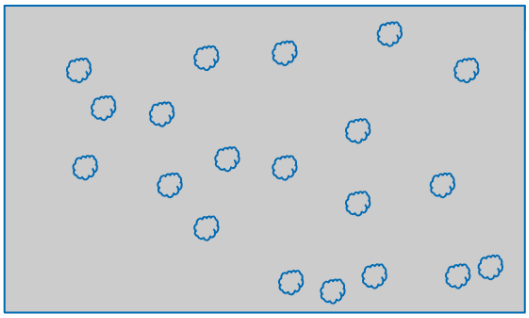


Figure 6: Evenly distributed particles: high entropy.

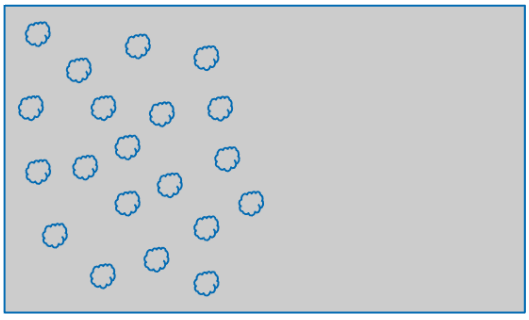


Figure 7: All the particles in one corner: low entropy.

The second state surprises us; the first does not. We also know how the story continues: before long the second container will look like the first, while the first will never look like the second. That is the **second law of thermodynamics**: in a closed system, entropy cannot decrease. A fried egg cannot be turned back into a raw one.

The obvious gloss, "entropy = disorder", is misleading. More precise is the distinction between *macrostate* and *microstate*. The macrostate is what can be measured – temperature, pressure, "all the particles on the left" – and the microstate is the complete specification of where every single particle sits. A macrostate usually comprises an enormous number of microstates, and exactly that number is

the measure: the macrostate "evenly distributed" comprises unimaginably more arrangements than the macrostate "all in the top left corner". Ludwig Boltzmann cast this into the formula

$$S = k_B \ln W \tag{45}$$

in which W is the number of microstates belonging to the given macrostate and k_B a physical constant whose only job is to get the unit right. Put a partition into both containers and the difference becomes tangible:

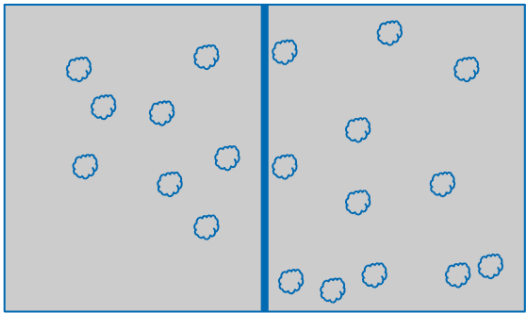


Figure 8: With a partition: there is roughly the same amount of gas on the left and on the right – this macrostate can be produced in a great many ways.

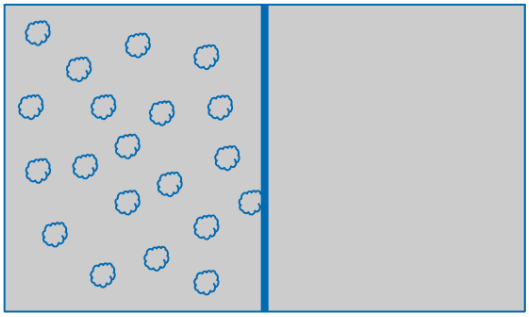


Figure 9: With a partition: all the gas is on the left – this macrostate admits only a few arrangements.

Remember
Entropy is always defined *relative to a macrostate*. We count the possibilities compatible with what we know, and then take the logarithm.

Carry that sentence over unchanged when you read the definition for random variables in a moment: there the macrostate is the distribution we know, and what gets counted (more precisely, weighted) are the outcomes it allows. The logarithm stays.

Claude Shannon (1916–2001) carried the notion from physics into communications engineering in his 1948 paper *A Mathematical Theory of Communication* Shannon [1948], with which information theory began as a field of its own. Shannon asked how many bits are needed, at the very least, to transmit a message, and found the same quantity Boltzmann had written down for gas particles. In his honour it is also called the *Shannon entropy*.

Definition
For a discrete probability space Ω and $x \in \Omega$ we call the *information* (also the *self-information*) of x

$$I(x) := -\log_b(P(\{x\})). \tag{46}$$

The base b of the logarithm is usually 2, but in the end it does not matter much. For a random variable $X\colon \Omega' \to \Omega$ we can now consider the composition $I \circ X\colon \Omega' \to \mathbb{R}$, a real random variable. Its expected value is the average information of X, which we also call the *entropy*

$$\begin{aligned} H(X) := \mathbb{E}\left(I(X)\right) &= \sum_{\omega' \in \Omega'} -\log_b\left(P_X(\{X(\omega')\})\right) P_{\Omega'}(\{\omega'\}) \\ &= -\sum_{\omega \in \Omega} \log_b(P(\{\omega\}))P(\{\omega\}). \end{aligned} \tag{47}$$

Remark
The sum for $H(X)$ has a problem: for $P(\{\omega\}) = 0$, $\log_b(0)$ is undefined. So we extend the function $x \mapsto x \log x$ to 0 by its limit,

$$\lim_{x \to 0^+} x \log x = 0 \tag{48}$$

(l'Hôpital applied to $\frac{\log x}{1/x}$) and set $0 \log 0 := 0$. Summands belonging to impossible outcomes then contribute nothing, and the sum effectively runs only over the ω with $P(\{\omega\}) > 0$. That is the convention behind the sums written "over $p_i \neq 0$" in the chapter Relative entropy.

Example
For a uniform distribution on $\Omega = \{1, \ldots, n\}$ we always have $P(\{x\}) = \frac{1}{n}$ for $x \in \Omega$, so $I(x) = -\log_2(P(\{x\})) = -\log_2 \frac{1}{n} = \log_2(n)$ and hence $H(X) = \sum_{i=1}^{n} \frac{\log_2(n)}{n} = \log_2(n)$. This number tells us that on average we need $\log_2(n)$ bits to communicate a single outcome of X, provided sender and receiver know the distribution; a sample of size N then costs $N \log_2(n)$ bits. There are other distributions for which less information per outcome suffices on average – strictly speaking every other one, in fact, since the uniform distribution has maximal entropy among all discrete distributions. It corresponds to knowing as little as possible and therefore taking all outcomes to be equally probable.

Example
How much information is there in a letter? Count the frequencies of the letters in German texts, put them into the formula, and out comes an entropy of about 4.06 bits per character Fox [2008]. For comparison: if all 26 letters were equally probable it would be $\log_2(26) \approx 4.70$ bits. The difference is exactly what the unequal frequencies of "e" and "x" save.

Take into account which letters have *gone before* as well, and the value drops to about 1.5 bits per letter – after "sch" the next letter is hardly a surprise. Language, then, is highly redundant, which is why text compresses so well and why you can still understand a sentence with half of it missing.

What that means in practice shows up in guessing. A random string of length 10 over the alphabet, say `pashglwnmv`, carries $10 \cdot 4.70 \approx 47$ bits – guessing it takes on the order of 2^{47} attempts. If the guesser knows instead that it is a German word, say `kabelkanal`, that is only $10 \cdot 1.5 = 15$ bits, or $2^{15} = 32{,}768$ attempts. That is the difference between "practically impossible" and "done in a second", and it comes not from the length of the string but from what the guesser knows about it.

This very argument is at the heart of one of the best-known comics on security:

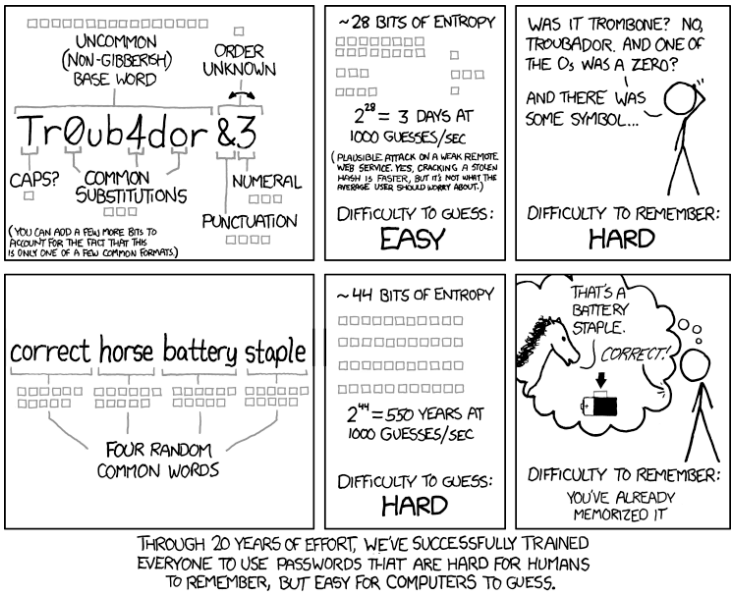


Figure 10: Randall Munroe, xkcd 936 "Password Strength" (CC BY-NC 2.5)

The password `Tr0ub4dor&3` looks complicated, but it was built by a procedure an attacker knows (a word, a few of the usual substitutions, a digit, a special character) – it carries around 28 bits. Four randomly chosen words such as `correct horse battery staple` carry around 44 bits, so they are about $2^{16} \approx 65{,}000$ times harder to guess – and easier to remember into the bargain.

The lesson is the same as in the previous example: entropy depends not on how the outcome looks but on the **procedure that produced it**. A password is secure not because it looks cryptic but because the set of passwords it could have come from is large. That is why the question "how strong is this password?" strictly speaking has no answer for a single password – what can be answered is how strong the procedure is.

At the other end of the scale stands the Dirac distribution: if $P(\{\xi\}) = 1$ for a single $\xi \in \Omega$ and $P(\{\omega\}) = 0$ for all the others, then $I(\xi) = -\log_2(1) = 0$ and hence $H(X) = 0$ (the summands with $P(\{\omega\}) = 0$ fall under the convention $0 \log 0 = 0$ from above). That is the smallest possible value, since $H(X)$ is a sum of non-negative terms. On a finite set Ω, then, we always have $0 \leq H(X) \leq \log_2 |\Omega|$, with the minimum attained exactly by the Dirac distributions (there are $|\Omega|$ of them, one per element) and the maximum only by the uniform distribution.

Entropy is best understood as a relative quantity, and that is how we shall treat it later on. In the rest of the course it turns up in two places: in setting up statistical models – the maximum entropy principle in the chapter The normal distribution and the closely related maximum likelihood principle in the chapter Statistical models and estimators – and in assessing how good a model is, where the tools are cross-entropy and relative entropy in the chapter Relative entropy.

2.3.4 Independence

Definition
Let Ω be a probability space. Events $A, B \subseteq \Omega$ are called *independent of each other* if

$$P(A \cap B) = P(A) \cdot P(B). \tag{49}$$

More generally, a family $(A_i)_{i \in I}$, $A_i \subseteq \Omega$, of events is called *independent* if for every finite subset $J \subseteq I$ of indices:

$$P\left(\bigcap_{j \in J} A_j\right) = \prod_{j \in J} P(A_j). \tag{50}$$

In general it makes a real difference whether a family is independent or only pairwise independent, which is why the definition in terms of families matters.

Example
Independent: two dice. We throw two fair six-sided dice. As the probability space we take $\Omega = \{1, \dots, 6\}^2$ with the uniform distribution, so 36 equally probable outcomes (i, j). Let A be the event "the first die shows an even number" and B the event "the second die shows a six". Then

$$P(A) = \frac{18}{36} = \frac{1}{2}, \qquad P(B) = \frac{6}{36} = \frac{1}{6}. \tag{51}$$

For the intersection we count rather than use a formula: $A \cap B = \{(2,6), (4,6), (6,6)\}$, so $P(A \cap B) = \frac{3}{36} = \frac{1}{12}$. And indeed $\frac{1}{2} \cdot \frac{1}{6} = \frac{1}{12}$, so the two events are independent.

That we counted matters: had we computed $P(A \cap B)$ as a product straight away, we would have assumed independence instead of checking it. Here it is obvious anyway, since A and B refer to different dice.

Example
Dependent: sweets without replacement. There are three sweets in a bag: a red one, a green one and a blue one. We draw two of them one after the other, without putting the first back. The probability space is the set of the 6 ordered pairs of distinct colours,

$$\Omega = \{(r,g), (r,b), (g,r), (g,b), (b,r), (b,g)\}, \tag{52}$$

again with the uniform distribution. Let A be the event "the first sweet is red" and B the event "the second sweet is green". Then $A = \{(r,g), (r,b)\}$ with $P(A) = \frac{2}{6} = \frac{1}{3}$ and $B = \{(r,g), (b,g)\}$ with $P(B) = \frac{2}{6} = \frac{1}{3}$. The intersection contains just one outcome, $A \cap B = \{(r,g)\}$, so

$$P(A \cap B) = \frac{1}{6} \neq \frac{1}{9} = P(A)\,P(B). \tag{53}$$

The two events are therefore *not* independent. That makes sense too: if the red sweet was drawn first, only green and blue are left, and green has suddenly become more probable on the second draw – up from $\frac{1}{3}$ to $\frac{1}{2}$. That number gets a name further down: it is the conditional probability $P(B \mid A)$.

Drawing without replacement is `random.sample` in Python, drawing with replacement `random.choices` (see Programming with stochastics) – the difference between the two functions is exactly the difference between dependent and independent.

Definition
Let Ω be a probability space. A family of random variables $(Y_i)_{i \in I}$, $Y_i \colon \Omega \to \Omega_i$, is called *independent* if for arbitrary events $B_i \subseteq \Omega_i$ the family $(\{Y_i \in B_i\})_{i \in I}$ is independent in Ω.

Remark
For two random variables $X, Y \colon \Omega \to \mathbb{R}$ this reads as follows: X and Y are independent exactly when

$$P(X \in A,\ Y \in B) = P(X \in A)\,P(Y \in B) \tag{54}$$

holds for all events A, B. The left-hand side is shorthand for $P\left((X,Y) \in A \times B\right)$ – so it is not about X and Y separately but about the pair (X,Y), understood as *one* random variable with values in $\mathbb{R}^2$. Its distribution is called the **joint distribution** of X and Y; we define it formally in the chapter Covariance and dependence.

The direction matters: the joint distribution is the real information, and the distributions of X and Y separately follow from it. The other way round does *not* work – from the individual distributions alone the joint one generally cannot be reconstructed. That is exactly what independence says: it is the extra assumption under which the product $p(x)q(y)$ of the two pmfs is the joint pmf.

Example
For $I = \{1, \ldots, n\}$ and each Ω_i discrete, $Y_1, \ldots, Y_n$ are independent exactly when for all outcomes $\omega_i \in \Omega_i$:

$$P(Y_1 = \omega_1, \ldots, Y_n = \omega_n) = \prod_{i=1}^{n} P(Y_i = \omega_i). \tag{55}$$

Another notion comes up very often in this connection:

Definition
Let Ω be a probability space. For events $A, B \subseteq \Omega$,

$$P(A|B) := \frac{P(A \cap B)}{P(B)} \tag{56}$$

is called the *conditional probability* of A given B.

Clearly, for $P(B) > 0$, A and B are independent of each other exactly when $P(A|B) = P(A)$. In the sweets example above, $P(B \mid A) = \frac{P(A \cap B)}{P(A)} = \frac{1/6}{1/3} = \frac{1}{2}$, which is not $P(B) = \frac{1}{3}$ – that is what dependence looks like in numbers. Conditional probabilities therefore come up (or rather, become interesting) precisely when two events, or two random variables, are not independent.

Looking ahead
Independence is an all-or-nothing property; in real data the more interesting question is *how strongly* two quantities are related. We will meet two measures for that, and it is worth keeping them apart from the start.

The first is the **covariance** and the correlation coefficient formed from it (chapter Covariance and dependence). It measures *linear* relationships. Independence implies that the covariance is 0 – but not conversely: two quantities can depend on each other perfectly and still be uncorrelated, for instance X and X^2 for symmetrically distributed X. So "correlated" is a stronger statement than "dependent", and "uncorrelated" a weaker one than "independent".

The second measure captures dependence of any shape: the **relative entropy** between the joint distribution and the product of the individual distributions, also called *transinformation* or *mutual information* $I(X;Y)$ (chapter Relative entropy). It is 0 exactly when the two random variables are independent – so it really does measure dependence and not just its linear part. The price: you need the joint distribution, and estimating that is considerably more work than computing a covariance.

Rules of calculation for the variance The expected value is linear; the variance is not. Instead, the following two rules hold, and we will need them constantly in the next chapter.

Lemma
Let $X : \Omega \to \mathbb{R}$ be a random variable with finite variance.

1. For $a, b \in \mathbb{R}$, $\mathbb{V}(aX + b) = a^2\,\mathbb{V}(X)$.
2. If X and Y are independent, then $\mathbb{V}(X + Y) = \mathbb{V}(X) + \mathbb{V}(Y)$. More generally, for pairwise independent $X_1, \ldots, X_n$,

$$\mathbb{V}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathbb{V}(X_i). \tag{57}$$

Proof

(1) By the linearity of the expected value, $\mathbb{E}(aX+b) = a\,\mathbb{E}X + b$, so

$$aX + b - \mathbb{E}(aX+b) = a\,(X - \mathbb{E}X) \tag{58}$$

and hence $\mathbb{V}(aX+b) = \mathbb{E}\left(a^2(X-\mathbb{E}X)^2\right) = a^2\,\mathbb{V}(X)$. In particular, a shift by b does not change the variance.

(2) We expand and use linearity:

$$\begin{aligned}\mathbb{V}(X+Y) &= \mathbb{E}\left(((X-\mathbb{E}X)+(Y-\mathbb{E}Y))^2\right)\\ &= \mathbb{E}\left((X-\mathbb{E}X)^2\right) + \mathbb{E}\left((Y-\mathbb{E}Y)^2\right) + 2\,\mathbb{E}\left((X-\mathbb{E}X)(Y-\mathbb{E}Y)\right)\\ &= \mathbb{V}(X) + \mathbb{V}(Y) + 2\left(\mathbb{E}(XY) - \mathbb{E}(X)\mathbb{E}(Y)\right).\end{aligned} \tag{59}$$

If X and Y are independent, the bracketed expression equals 0 by the product rule. The general case follows the same way: expand, and all the mixed terms $\mathbb{E}(X_iX_j) - \mathbb{E}X_i\,\mathbb{E}X_j$ with $i \neq j$ vanish – pairwise independence is enough for that (chapter Covariance).

The term $\mathbb{E}(XY) - \mathbb{E}(X)\mathbb{E}(Y)$ that has just vanished is an interesting quantity in its own right: it is called the *covariance* of X and Y, and it measures how strongly the two variables are linearly related. In the chapter Covariance we put it under the microscope and prove the rule once more there, in its general form, without the assumption of independence.

Rule (2) is why the variance of a sum of independent repetitions of the same experiment is so easy to compute – we use that in the chapter The most important discrete distributions for the binomial distribution.

2.4 Numpy: vectorisation and arrays

This short chapter introduces the essential ideas behind Numpy. A longer and better version is the Numpy fundamentals in the Numpy User Guide, which you would do well to skim afterwards. A proper (and very good) textbook for anyone already at home in Python – which is what we take ourselves to be at this point in the course – is Nicolas Rougier's From Python to Numpy; chapter 3 is the part that concerns us first.

In short: Numpy brings array-oriented programming (also called *vectorisation*) to Python.

Why is that worth having? Three reasons, and they hang together.

First, Numpy is above all an **interface to very fast libraries** for linear algebra and statistics – written in C, in Fortran and here and there in assembler, and optimised over decades. An `np.dot` calls code that nobody would want to rebuild in Python.

Second, this way of working allows **SIMD** (*single instruction, multiple data*): instead of looping over a list and handling each element on its own, you apply one operation to a whole vector. Modern processors can compute with several values in a single step – but only if the same thing is to be done to all of them. Branches (`if` tests inside the loop) ruin that. Hence this chapter's working rule: **write branch-free code, which is to say, vectorise.** Where a case distinction really is needed, you express it as a mask over the whole array rather than an `if` in the body of a loop; the section *Masks and fancy indexing* shows how.

Third, this way of thinking carries over. PyTorch, the standard library for neural networks, works in almost the same way – there the object is called a `tensor` rather than an `ndarray`, and it lives either in main memory or on the graphics card. Whoever has learned to vectorise here can carry on doing it there; the course *Introduction to Deep Learning* assumes precisely that.

We use Numpy version 2.5 or higher, for which Python 3.12 or above is required. Most of the code in these notes should run under Numpy 1.26 as well, though. NEP 29 has more on version compatibility.

2.4.1 Arrays

Numeric Python (Numpy) is usually imported under the abbreviation `np`:

```
import numpy as np
np.__version__
```

```
'2.4.6'
```

```
sample = np.random.random(10)
print(sample)
print(type(sample))
sample
```

```
[0.74080928 0.51759931 0.12066137 0.42954233 0.74301052 0.61968243
 0.84294153 0.67199131 0.84591432 0.97695916]
<class 'numpy.ndarray'>
```

```
array([0.74080928, 0.51759931, 0.12066137, 0.42954233, 0.74301052,
       0.61968243, 0.84294153, 0.67199131, 0.84591432, 0.97695916])
```

Numpy contains a great many useful helper functions, which in turn operate on Numpy's own data structure, the array.

```
sample = np.random.randint(low=0, high=10, size=5000)
print("mean    =", np.mean(sample),   "\nexpected=", 9/2)
# 99/12 = (n² -1)/12 with n=10, the variance of the discrete uniform distribution
# (derived in the chapter "The most important discrete distributions"):
print("std²    =", np.std(sample)**2, "\nvariance=", 99/12)
```

```
mean    = 4.611
expected= 4.5
std²    = 8.365679
variance= 8.25
```

There is an `arange` function too, but in Numpy it does not produce range objects:

```
myRange = np.arange(1, 7)
print(myRange, type(myRange), myRange.dtype)
myRange
```

```
[1 2 3 4 5 6] <class 'numpy.ndarray'> int64
```

```
array([1, 2, 3, 4, 5, 6])
```

Unlike a Python list, the Numpy array `ndarray` carries one fixed data type that all its elements share. That can be any of the Numpy data types (dtypes): `double`, for instance (compatible with the Python `float`), `float32` (on most platforms compatible with the C `float`) or `long` (compatible with the Python `int`).

```
import timeit
ordinary_list = [1,2,3,4,5,6,5,4,3,2,1]*10
def sort_array(dtype):
    a = np.array(ordinary_list, dtype)
    a.sort()
print(timeit.timeit(lambda : sort_array(np.byte)))
print(timeit.timeit(lambda : sort_array(np.float64)))
```

```
2.85545536223799
```

```
3.4881999790668488
```

Because the data type is known exactly, Numpy can use algorithms optimised for it and implemented directly in C.

The documentation lists the other `dtype`s and how to use them:

- "Structured Arrays"
- "Data types" in the "Numpy fundamentals"
- "Scalars"
- "Data type objects (dtype)"
- "numpy.dtype"

Arrays can be created in more ways than `arange` or converting a Python sequence: `ones`, for instance, gives you an array filled with 1, and `zeros` one filled with 0.

The array is called `ndarray` because that stands for "n-dimensional array". If you wanted to store a matrix in plain Python, you would do it as a list of the row vectors (or of the column vectors), something like

```
matrix = [[1,0,0], [0,1,0], [0,0,1]]  # identity matrix
print(matrix, type(matrix))
shamtrix = matrix[:]   # a "copy" by slicing
print(shamtrix[0][0])
shamtrix[0][0] = 0     # we change the top left entry
print(matrix[0][0])
```

```
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] <class 'list'>
1
0
```

This exposes a problem: Python treats our matrix as a list – which is, after all, how we wrote it down – and so copying the list does not copy its contents (the row vectors), only the pointers to them.

```
npmatrix = np.identity(3, int)  # identity matrix
print(npmatrix, type(npmatrix))
npshamtrix = npmatrix[:]
print(npshamtrix[0][0])
npshamtrix[0][0] = 2     # we change the top left entry
print(npmatrix[0][0])
# With a real copy this would not have happened:
real_copy = npmatrix.copy()
real_copy[0][0] = 1
assert npmatrix[0][0] != 1
```

```
[[1 0 0]
 [0 1 0]
 [0 0 1]] <class 'numpy.ndarray'>
1
2
```

Note that the Numpy array behaves exactly like our Python list of lists: we index and slice it the same way, and copying it via the slicing syntax has the same problem.

The `shape` parameter tells us what shape our Numpy array has. It is a d-tuple, where d is the dimension. A matrix is 2-dimensional, a vector 1-dimensional and a scalar 0-dimensional.

```
print(npmatrix.shape)
print(npmatrix[0], npmatrix[0].shape)
print(npmatrix[0][0], npmatrix[0][0].shape)
```

```
(3, 3)
[2 0 0] (3,)
2 ()
```

Numpy lets you slice rather more finely. The general syntax is `[start:stop:step, ..]`, where the comma moves you from one axis to the next. A two-dimensional array has two axes: axis 0 runs from top to bottom and axis 1 from left to right. The "step" works with Python lists too; indexing along several axes is a Numpy speciality.

```
numbers = list(range(1,10))
npmatrix = np.array(numbers)
print("not in shape:", npmatrix)
npmatrix = npmatrix.reshape(3, 3)
print("in much better shape:\n"+ str(npmatrix))
print("row vector, row 0:", npmatrix[0])
print("column vector, column 0:", npmatrix[:,0])
print("columns 1 -2:\n"+ str(npmatrix[:,1:]))
print("all rows, step 2\n"+ str(npmatrix[0::2]))
```

```
not in shape: [1 2 3 4 5 6 7 8 9]
in much better shape:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
row vector, row 0: [1 2 3]
column vector, column 0: [1 4 7]
columns 1 -2:
[[2 3]
 [5 6]
 [8 9]]
all rows, step 2
[[1 2 3]
 [7 8 9]]
```

In classic Python we would reach an entry of a matrix with `matrix[x][y]`, and that works in Numpy too. But it first creates a further array object `matrix[x]` (a view, see below) and only then calls `[y]` on it. In principle, then, Numpy's own `[x,y]` is the more efficient choice.

When slicing, Numpy deliberately makes no copies, only a different view of the same region of memory (which is also why `[:]` behaved as it did above). `np.may_share_memory` checks whether two arrays point into the same region of memory. A positive answer by no means says that the arrays depend on one another – the first and the second column of a matrix also point into the same region of memory, namely the whole matrix. Change one of the two vectors and the other stays as it was – but the whole matrix changes along with it.

```
npmatrix = np.array(list(range(1,10))).reshape(3,3)
candidates = (npmatrix[0::2], npmatrix[1])
print("May share memory (but actually don't):",
      candidates, np.may_share_memory(*candidates))
print(type([0,0,0]))     # before the assignment
npmatrix[1] = [0,0,0]
```

```
print(type(npmatrix[1])) # after the assignment
print(npmatrix)          # the whole matrix has changed with it

May share memory (but actually don't): (array([[1, 2, 3],
       [7, 8, 9]]), array([4, 5, 6])) True
<class 'list'>
<class 'numpy.ndarray'>
[[1 2 3]
 [0 0 0]
 [7 8 9]]
```

For the details of slicing and indexing, see the documentation.

2.4.2 Broadcasting

For Python lists, the addition operator means concatenation, which also gives multiplying a list by a scalar a well-defined meaning, while multiplying two lists is undefined. Numpy arrays support a good many more arithmetic operations:

```
E = np.identity(3, int)
print(E, "= E")
A = np.ones((3,3), int)
A[0] = [0,0,0]
print(A, "= A")
print(E + A, "= E + A")
print(E * A, "= EA")
print((E+A)**2, "= (E+A)(E+A)")

[[1 0 0]
 [0 1 0]
 [0 0 1]] = E
[[0 0 0]
 [1 1 1]
 [1 1 1]] = A
[[1 0 0]
 [1 2 1]
 [1 1 2]] = E + A
[[0 0 0]
 [0 1 0]
 [0 0 1]] = EA
[[1 0 0]
 [1 4 1]
 [1 1 4]] = (E+A)(E+A)
```

Look closely and you will notice that these are not matrix multiplications: the calculations have simply been carried out element by element, because that is how the operations on arrays are defined. This one is particularly treacherous:

```
v = np.ones(3, int)
print(v, "= v")
print(A*v, "= A*v (but not the matrix product)")

[1 1 1] = v
[[0 0 0]
 [1 1 1]
 [1 1 1]] = A*v (but not the matrix product)
```

For explicit matrix algebra, people used to reach for the Numpy data type `matrix`, but it is marked as `deprecated` and will disappear in future Numpy versions. These days you use `np.matmul` or the infix operator `@`.

```
print(np.matmul(A,v))
print(A@v)

[0 3 3]
[0 3 3]
```

For many problems it is a great help to leave matrix algebra aside and carry out the arithmetic element by element. *Broadcasting* is the mechanism that makes such element-wise arithmetic rather more convenient. The operation ($n \times n$ array) * (n-vector), for instance, is interpreted automatically by copying the n-vector n times, so that each row of the left-hand array is multiplied (element by element) by the vector.

It really is worth skimming the documentation on this once.

A *tensor* in PyTorch, the library of the course *Introduction to Deep Learning*, is essentially a Numpy array with two additions: a device assignment (CPU or GPU) and automatic differentiation. `shape`, axes (`axis`) and broadcasting work there exactly as they do in Numpy, which is why that course assumes this chapter.

2.4.3 Masks and fancy indexing

A comparison such as `a >2` is evaluated element by element and yields an array of `True`/`False`: a **mask**. A mask can serve as an index: you get exactly the entries where it says `True`, and you can assign to those entries as well. This is the tool we filter data with in the next chapter ("all passengers under 13", "all rows with no missing value"). `np.where(condition, a, b)` chooses element by element between two arrays: where the condition holds, the entry from `a`, otherwise the one from `b`. That is how a per-element `if` test becomes a single array operation:

```
a = np.array([3, 1, 4, 1, 5, 9])
mask = a > 2
print(mask)                          # mask: array of True/False
print(a[mask])                       # only the entries where it says True
print(a[(a > 2) & (a < 9)])          # combine conditions with & (and), | (or), ~ (not),
                                     # not with and/or; every condition in brackets!
a[a == 1] = 0                        # assigning to a mask
print(a)
print(a[[0, 2, 4]])                  # "fancy indexing": a list of positions as the index
print(np.isnan(np.array([1.0, np.nan, 3.0])))   # masks for missing values
print(np.where(a > 2, a, 0))         # np.where: where a > 2 the value from a, otherwise 0 (no if
    needed)
```

```
[ True False  True False  True  True]
[3 4 5 9]
[3 4 5]
[3 0 4 0 5 9]
[3 4 5]
[False  True False]
[3 0 4 0 5 9]
```

2.4.4 Probability vectors

In the chapter on random variables, a distribution on n values was a vector $p \in \Delta_n$: non-negative entries that sum to 1. In Numpy that is simply an array divided by its own sum, and a sample from this distribution is a call to `choice` with the argument `p`:

```
p = np.array([1, 1, 1, 1, 1, 3], dtype=float)   # loaded die: the 6 is three times as likely
p = p / p.sum()                                   # normalise: p now lies in the simplex \
    Delta_6
print(p, p.sum())

rng = np.random.default_rng(1)
rolls = rng.choice(np.arange(1, 7), size=10000, p=p)   # sample from the pmf p
print(np.bincount(rolls)[1:] / len(rolls))             # relative frequencies, roughly p
```

```
[0.125 0.125 0.125 0.125 0.125 0.375] 1.0
[0.1269 0.1222 0.1218 0.1244 0.1285 0.3762]
```

2.4.5 ufuncs

`ufunc` stands for "universal function" and denotes a function that can run vectorised on Numpy arrays.

Every `ufunc` supports a `reduce`, for instance, which applies the `ufunc` along one axis of the array to the smaller arrays that result.

To write a `ufunc` of your own you have to program in C or else wrap a Python function.

It is worth taking a quick look at all the `ufunc`s that are already defined: Available ufuncs

2.4.6 What do you need to know?

Numpy and the other libraries of scientific computing in Python are large; nobody can know every function. For these notes, a pass over the code cells of all the notebooks (`grep -ohE "np\.[a-z_]+(\.[a-z_]+)?" *.ipynb | sort -u`, tidied up by hand) yields the complete list below, as of September 2026. You do not have to learn it by heart, but it does give a clear upper bound: more than this you definitely do not need to know! The functions in **bold** are the ones you really should have in your head, because you type them constantly; the rest is a reference work. Bear in mind, too, that the list covers the whole of these notes, and that much of it we have not met yet.

For day-to-day work a cheat sheet helps more than a list: the Numpy reference has an overview sorted by topic, and for getting started there is the compact beginners' chapter of the Numpy documentation. Print it out, keep it beside you, use it – learning it by heart gets you nowhere here.

- **abs**
- **all**
- **allclose**
- append
- **arange**
- **argmin**
- **array**
- atleast_1d
- bincount
- byte
- c_
- column_stack
- **concatenate**
- **copy**
- **corrcoef**
- cos
- **cov**
- **cumsum**
- diag
- **diff**
- **dot**
- dstack
- equal
- **exp**

- **eye**
- float64
- **full**
- genfromtxt
- heaviside
- **hstack**
- identity
- **isclose**
- **isnan**
- linalg.det
- linalg.eig
- linalg.eigh
- **linalg.inv**
- **linalg.norm**
- linalg.svd
- **linspace**
- **log**
- log2
- logspace
- **matmul**
- **max**
- **maximum**
- may_share_memory
- **mean**
- **median**
- **meshgrid**
- mgrid
- **min**
- **minimum**
- **nan**
- nansum
- **ones**
- ones_like
- outer
- **pi**
- power

- ptp
- **random.choice**
- **random.default_rng**
- random.multivariate_normal
- random.permutation
- random.rand
- random.randint
- random.randn
- random.random
- **random.seed**
- random.uniform
- **round**
- searchsorted
- set_printoptions
- sin
- size
- **sort**
- **sqrt**
- **square**
- **stack**
- **std**
- **sum**
- **unique**
- **var**
- **vstack**
- **where**
- **zeros**
- zeros_like

2.5 The most important discrete distributions in SciPy

We now have everything we need for an overview of the most important discrete distributions: the pmf from the chapter Random variables and distributions, expected value, variance and entropy from the chapter Expected value and variance, and from the chapter Numpy the machinery for actually computing with them.

We present every distribution in the same format – parameters, pmf, expected value, variance, entropy, example applications and the name in `scipy.stats` – just as we will later do for the continuous distributions.

2.5.1 Convention: discrete probability spaces

In this chapter Ω is always *at most countable*, and usually even finite. Then the power set $\mathcal{P}(\Omega)$ is a σ-algebra, so every subset of Ω is an event, and the probability measure P is completely determined by the values

$$p(\omega) := P(\{\omega\}) \tag{60}$$

on the individual outcomes – because every event $A \subseteq \Omega$ is the (at most countable) disjoint union of its elements, so that $P(A) = \sum_{\omega \in A} p(\omega)$. This function p is the *probability mass function* (pmf); *discrete density* is another name in common use. Everything we want to know about a discrete distribution can be read off it:

$$\mathbb{E}(X) = \sum_x x\, p_X(x), \qquad \mathbb{V}(X) = \mathbb{E}(X^2) - (\mathbb{E}X)^2, \qquad H(X) = -\sum_x p_X(x) \log_2 p_X(x). \tag{61}$$

> **Remark**
> Some sources call the pmf the *distribution function* in the discrete case. We avoid that name: in the chapter Continuous distributions it belongs to the *cumulative* distribution function $F(x) = P(X \leq x)$.

2.5.2 Discrete distributions in SciPy

`scipy.stats` provides objects for discrete distributions just as it does for continuous ones. The main difference: where a continuous distribution has a density `pdf`, a discrete distribution has a pmf, and the method is accordingly called `pmf`.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as st


def plotXdiscrete(k, X, name, params):
    description = ", ".join(f"{value}" for value in params)
    fig, axs = plt.subplots(1, 2, figsize=(12, 4))
    # First the probability mass function:
    axs[0].stem(k, X.pmf(k), basefmt=" ")
    axs[0].set_xlabel("$k$")
    axs[0].set_ylabel("$p(k) = P(X = k)$")
    axs[0].set_title(f"pmf of the {name} with {description}")
    # Then the cumulative distribution function:
    axs[1].step(k, X.cdf(k), where="post")
    axs[1].set_xlabel("$k$")
    axs[1].set_ylabel("$F(k) = P(X \\leq k)$")
    axs[1].set_title(f"cdf of the {name}")
    # tight_layout keeps the two sets of axis labels from overlapping:
    plt.tight_layout()
    plt.show()
```

The methods we need over and over again are the same for every distribution. If `X` is a discrete random variable from `scipy.stats`, then:

wanted	call
$P(X = k)$	`X.pmf(k)`
$P(X \le k)$	`X.cdf(k)`
$P(X > k)$	`X.sf(k)`, equivalent to `1 - X.cdf(k)`
$P(X \ge k)$	`X.sf(k-1)`
$P(X < k)$	`X.cdf(k-1)`
a sample of size n	`X.rvs(size=n)`
$\mathbb{E}(X)$, $\mathbb{V}(X)$, $\sigma(X)$	`X.mean()`, `X.var()`, `X.std()`

The `sf` stands for *survival function*.

Caution
For discrete distributions the difference between $\le$ and $<$ is not merely cosmetic, because individual values carry strictly positive probability. "At most k" is `X.cdf(k)`, "fewer than k" is `X.cdf(k-1)`, "more than k" is `X.sf(k)` and "at least k" is `X.sf(k-1)`. This is by far the most common source of error when computing with these distributions.

```
X = st.binom(10, 0.3)
print("P(X = 3)   =", X.pmf(3))
print("P(X <= 3)  =", X.cdf(3))
print("P(X > 3)   =", X.sf(3), "= 1 - P(X <= 3) =", 1 - X.cdf(3))
print("P(X >= 3)  =", X.sf(2))
assert np.isclose(X.cdf(3) + X.sf(3), 1.0)
```

```
P(X = 3)   = 0.2668279319999998
P(X <= 3)  = 0.6496107184000002
P(X > 3)   = 0.3503892815999998 = 1 - P(X <= 3) = 0.3503892815999998
P(X >= 3)  = 0.6172172136000003
```

2.5.3 Discrete uniform distribution

- Parameter n, notation $\mathrm{Uni}\{1, \dots, n\}$
- pmf $p(k) = \dfrac{1}{n}$ for $k \in \{1, \dots, n\}$
- Expected value $\dfrac{n+1}{2}$
- Variance $\dfrac{n^2 - 1}{12}$
- Entropy $\log_2 n$ – the maximum among all distributions on a set with n elements (see Expected value and variance, section Entropy)
- Example applications:
  - The fair die: $n = 6$, so $\mathbb{E}(X) = 3.5$ and $\mathbb{V}(X) = \frac{35}{12} \approx 2.92$.
  - An idealised random number generator that returns integers in a fixed range.
- In `scipy.stats`: `st.randint(1, n+1)` – careful: as with `range`, the upper bound is **exclusive**.

We already know the expected value. For the variance we use $\mathbb{V}(X) = \mathbb{E}(X^2) - (\mathbb{E}X)^2$. With the summation formula $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ we get

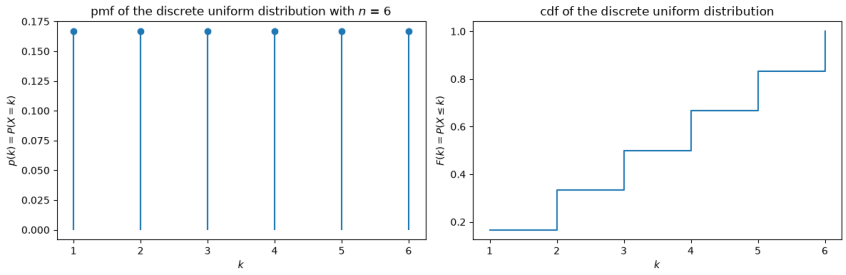
$$\mathbb{E}(X^2) = \frac{1}{n} \sum_{i=1}^{n} i^2 = \frac{(n+1)(2n+1)}{6}, \tag{62}$$

and therefore

$$\mathbb{V}(X) = \frac{(n+1)(2n+1)}{6} - \frac{(n+1)^2}{4} = \frac{(n+1)\left(2(2n+1) - 3(n+1)\right)}{12} = \frac{(n+1)(n-1)}{12} = \frac{n^2-1}{12}. \tag{63}$$

```
n = 6
X = st.randint(1, n + 1)   # upper bound exclusive, so 1 to 6
print("E(X) =", X.mean(), " expected:", (n + 1) / 2)
print("V(X) =", X.var(),  " expected:", (n**2 - 1) / 12)

plotXdiscrete(np.arange(1, n + 1), X, "discrete uniform distribution", (f"$n$ = {n}",))
```

```
E(X) = 3.5  expected: 3.5
V(X) = 2.9166666666666665  expected: 2.9166666666666665
```



2.5.4 Categorical distribution

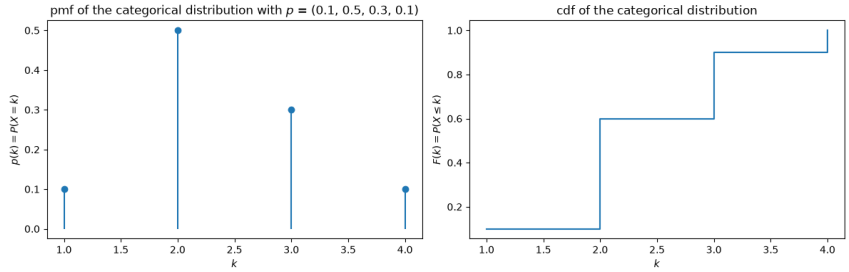
- Parameter a probability vector $p = (p_1, \ldots, p_n) \in \Delta_n$, that is $p_i \geq 0$ and $\sum_i p_i = 1$
- pmf $p(\omega_i) = p_i$ on a set $\Omega = \{\omega_1, \ldots, \omega_n\}$ with n elements
- Expected value (if we identify ω_i with the number i) $\sum_{i=1}^{n} i\, p_i$ – which in general does not simplify any further
- Variance $\sum_{i=1}^{n} i^2 p_i - \left(\sum_{i=1}^{n} i\, p_i\right)^2$
- Entropy $-\sum_i p_i \log_2 p_i$
- Example applications:
  - An urn with m balls in n colours, n_i of them of colour i; drawing **with** replacement gives $p_i = \frac{n_i}{m}$.
  - Any survey with a fixed list of mutually exclusive answers.
- In `scipy.stats`: no object of its own, but `st.rv_discrete(values=(values, p))` builds one; if all you want is to draw samples, `rng.choice(values, p=p)` from the chapter Numpy is enough.

The parameter space is exactly the simplex Δ_n we met in the chapter Random variables and distributions: the uniform distribution sits at its centre, the Dirac distributions at its vertices. The categorical distribution is therefore the most general distribution on a finite set – every distribution that follows in this chapter is either a special case of it or built from it. The four jumper colours from that chapter are a categorical distribution too, with $n = 4$.

> **Remark**
> In machine learning Δ_n shows up as an *output space*: when a classification algorithm is meant not merely to name a class but to express how uncertain it is, it returns a point in Δ_n – a categorical distribution over the classes.

```
values = np.arange(1, 5)
p = np.array([0.1, 0.5, 0.3, 0.1])
X = st.rv_discrete(values=(values, p))

print("E(X) =", X.mean(), " recomputed:", np.sum(values * p))
print("V(X) =", X.var())
print("H(X) =", X.entropy() / np.log(2), "bits")   # scipy computes in nats
print("for comparison, uniform distribution on 4 values:", np.log2(4), "bits")

plotXdiscrete(values, X, "categorical distribution", (f"$p$ = {tuple(p.tolist())}",))
```

```
E(X) = 2.4  recomputed: 2.4
V(X) = 0.6400000000000006
H(X) = 1.6854752972273344 bits
for comparison, uniform distribution on 4 values: 2.0 bits
```



2.5.5 Bernoulli distribution

- Parameter $p \in [0, 1]$, notation $\mathrm{Ber}(p)$
- pmf $p(1) = p$ and $p(0) = 1 - p$ on $\Omega = \{0, 1\}$
- Expected value p
- Variance $p(1 - p)$
- Entropy $h(p) := -p \log_2 p - (1 - p) \log_2(1 - p)$, the *binary entropy function*
- Example applications:
  - A single coin toss, unfair coins included.
  - Any yes/no feature: a component is defective or it is not, an email is spam or it is not.
- In `scipy.stats`: `st.bernoulli(p)`

The Bernoulli distribution is the special case $n = 2$ of the categorical distribution, and the simplest nontrivial case there is – it has **one** parameter. Expected value and variance are quick to compute: $\mathbb{E}(X) = 0 \cdot (1 - p) + 1 \cdot p = p$, and since $X^2 = X$ (because $0^2 = 0$ and $1^2 = 1$), we also have $\mathbb{E}(X^2) = p$ and therefore

$$\mathbb{V}(X) = \mathbb{E}(X^2) - (\mathbb{E}X)^2 = p - p^2 = p(1 - p). \tag{64}$$

The variance is maximal at $p = \frac{1}{2}$ and vanishes for $p \in \{0, 1\}$ – there the outcome is no longer random at all.

In the chapter Random variables and distributions we already estimated a value $\hat{\pi}$ from data without giving it a name: it was the parameter of a Bernoulli distribution.

The binary entropy function The entropy of a Bernoulli distribution, as a function of p, deserves a look of its own. Let us differentiate it. With $\log_2 x = \frac{\ln x}{\ln 2}$ we have

$$\frac{d}{dp}\left(-p \log_2 p\right) = -\log_2 p - \frac{1}{\ln 2}, \qquad \frac{d}{dp}\left(-(1 - p) \log_2(1 - p)\right) = \log_2(1 - p) + \frac{1}{\ln 2}, \tag{65}$$

so the two constants cancel and what remains is

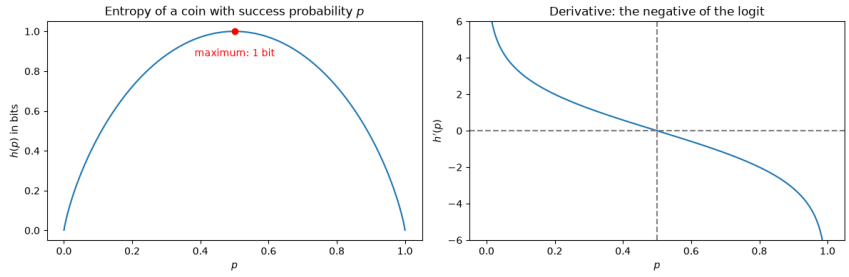
$$h'(p) = \log_2(1 - p) - \log_2 p = -\log_2 \frac{p}{1 - p}. \tag{66}$$

The only zero is at $p = 1 - p$, that is, at $p = \frac{1}{2}$: the fair coin toss has the maximal entropy, namely $h(\frac{1}{2}) = 1$ bit. That fits: the uniform distribution is the maximum entropy distribution.

Looking ahead

The expression $\log_2 \frac{p}{1-p}$ we have run into here has a name of its own: the fraction $\frac{p}{1-p}$ is called the *odds*, and its logarithm the *logit*. So the derivative of the binary entropy is exactly the negative of the logit. The inverse of the logit is the logistic sigmoid function, the building block from which *logistic regression* – and with it the single neuron of a neural network – is built. Anyone taking *Introduction to Deep Learning* after this course will meet this curve again straight away.

```
p = np.linspace(1e -4, 1 - 1e -4, 500)
h = -p * np.log2(p) - (1 - p) * np.log2(1 - p)
h_prime = -np.log2(p / (1 - p))

fig, axs = plt.subplots(1, 2, figsize=(12, 4))

axs[0].plot(p, h)
axs[0].plot(0.5, 1.0, "o", color="red")
axs[0].annotate("maximum: 1 bit", (0.5, 1.0), textcoords="offset points",
                xytext=(0, -25), ha="center", color="red")
axs[0].set_xlabel("$p$")
axs[0].set_ylabel("$h(p)$ in bits")
axs[0].set_title("Entropy of a coin with success probability $p$")

axs[1].plot(p, h_prime)
axs[1].axhline(0, linestyle=" - -", color="gray")
axs[1].axvline(0.5, linestyle=" - -", color="gray")
axs[1].set_ylim( -6, 6)
axs[1].set_xlabel("$p$")
axs[1].set_ylabel("$h'(p)$")
axs[1].set_title("Derivative: the negative of the logit")

plt.tight_layout()
plt.show()

X = st.bernoulli(0.3)
print("E(X) =", X.mean(), " V(X) =", X.var(), " H(X) =", X.entropy() / np.log(2), "bits")
```




```
E(X) = 0.3  V(X) = 0.21  H(X) = 0.8812908992306927 bits
```

2.5.6 Binomial distribution

- Parameters $n \in \mathbb{N}$ and $p \in [0, 1]$, notation $\mathrm{Bin}(n, p)$
- pmf $p(k) = \binom{n}{k} p^k (1-p)^{n-k}$ for $k \in \{0, 1, \ldots, n\}$
- Expected value np
- Variance $np(1-p)$
- Entropy: no convenient closed expression; it is smaller than $\log_2(n+1)$, since the maximum would be the uniform distribution on $\{0, \ldots, n\}$
- Example applications:

- Tossing the same coin n times and counting the heads.
- Drawing a sample of n components and counting the defective ones.
- An urn with two colours, N balls of which n_1 are red, $p = \frac{n_1}{N}$, drawing n times **with** replacement.

- In `scipy.stats`: `st.binom(n, p)`

The binomial distribution arises from the Bernoulli distribution by repetition. If $X_1, \ldots, X_n$ are **independent** random variables with $X_i \sim \mathrm{Ber}(p)$, then

$$X := \sum_{i=1}^{n} X_i \tag{67}$$

counts the number of successes, and $X \sim \mathrm{Bin}(n, p)$. So it has **two** parameters.

Why the pmf looks the way it does: by independence, any one fixed pattern of k successes and $n-k$ failures – say $1, 1, 0, 1, 0, \ldots$ – has probability $p^k(1-p)^{n-k}$, whichever positions the successes occupy. And there are exactly $\binom{n}{k}$ such patterns, one for each subset of $\{1, \ldots, n\}$ with k elements. Summing over them gives the formula.

Expected value and variance now come for free. The expected value follows from linearity:

$$\mathbb{E}(X) = \sum_{i=1}^{n} \mathbb{E}(X_i) = n\,p. \tag{68}$$

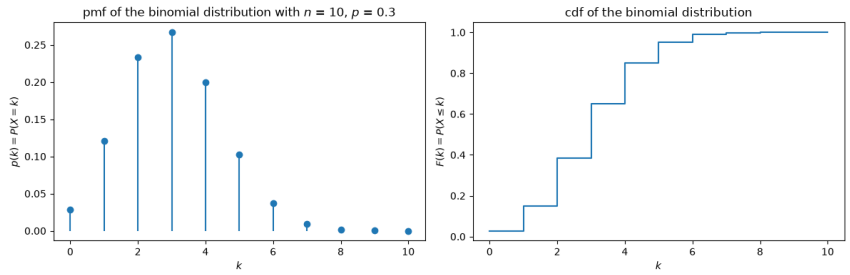
For the variance we need the independence – this is exactly the rule of calculation from the section Rules of calculation for the variance:

$$\mathbb{V}(X) = \sum_{i=1}^{n} \mathbb{V}(X_i) = n\,p(1-p). \tag{69}$$

Without independence this would be false. Here you can see clearly what that rule is good for.

```
n, p = 10, 0.3
X = st.binom(n, p)
print("E(X) =", X.mean(), " expected:", n * p)
print("V(X) =", X.var(),  " expected:", n * p * (1 - p))

plotXdiscrete(np.arange(0, n + 1), X, "binomial distribution", (f"$n$ = {n}", f"$p$ = {p}"))
```

```
E(X) = 3.0  expected: 3.0
V(X) = 2.1  expected: 2.0999999999999996
```



Which p fits the data best? So far we have treated p as given and asked for the probability of k successes. Now we turn the question around: we have found $k = 3$ defective components in a sample of $n = 10$ and want to know which p fits this observation.

For **every** p the pmf gives us a probability for exactly this observation:

$$P(X = 3) = \binom{10}{3} p^3 (1-p)^7. \tag{70}$$

Let us compare two candidates, $p = 0.2$ and $p = 0.5$. The binomial coefficient is the same in both cases and cancels, so it is enough to compare $p^3(1-p)^7$:

$$0.2^3 \cdot 0.8^7 = 0.008 \cdot 0.2097\ldots \approx 0.00168, \qquad 0.5^3 \cdot 0.5^7 = 0.5^{10} \approx 0.00098. \tag{71}$$

So our observation is about 1.7 times as likely under $p = 0.2$ as under $p = 0.5$. In this sense $p = 0.2$ fits better.

The natural next question is which p fits best of all. We are looking for the maximum of $p \mapsto p^k(1-p)^{n-k}$, so we differentiate:

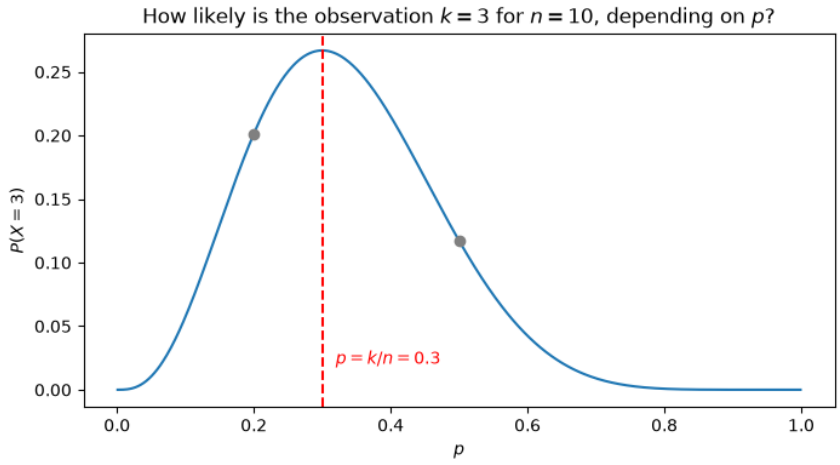
$$\frac{d}{dp}\left(p^k(1-p)^{n-k}\right) = k\,p^{k-1}(1-p)^{n-k} - (n-k)\,p^k(1-p)^{n-k-1} = p^{k-1}(1-p)^{n-k-1}\Big(k(1-p) - (n-k)p\Big). \tag{72}$$

For $0 < p < 1$ the leading factor is nonzero, so what remains is the condition $k(1-p) = (n-k)p$. Solving it: $k - kp = np - kp$, so $k = np$ and therefore

$$p = \frac{k}{n}. \tag{73}$$

For our observation that is $p = \frac{3}{10} = 0.3$ – simply the proportion of successes in the sample. That is exactly what you would have guessed without any calculation; what is remarkable is that here it drops out of a calculation, from a clearly stated criterion: choose the parameter that makes the observation you actually made as likely as possible.

```
k, n = 3, 10
p = np.linspace(0.001, 0.999, 500)
probability = st.binom(n, p).pmf(k)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(p, probability)
ax.axvline(k / n, linestyle=" - -", color="red")
ax.annotate(f"$p = k/n = {k/n}$", (k / n, 0.02), color="red",
            textcoords="offset points", xytext=(8, 0))
for candidate in (0.2, 0.5):
    ax.plot(candidate, st.binom(n, candidate).pmf(k), "o", color="gray")
ax.set_xlabel("$p$")
ax.set_ylabel(f"$P(X = {k})$")
ax.set_title(f"How likely is the observation $k={k}$ for $n={n}$, depending on $p$?")
plt.show()

print("P(X=3) at p=0.2:", st.binom(n, 0.2).pmf(k))
print("P(X=3) at p=0.5:", st.binom(n, 0.5).pmf(k))
print("P(X=3) at p=0.3:", st.binom(n, 0.3).pmf(k), "  < - the maximum")
```




```
P(X=3) at p=0.2: 0.20132659199999992
P(X=3) at p=0.5: 0.1171875
P(X=3) at p=0.3: 0.2668279319999998   < - the maximum
```

This procedure – choosing the parameter that makes the observation most likely – is called *maximum likelihood estimation*. The chapter Statistical models treats it in general and for other distributions.

2.5.7 Digression: the Poisson binomial distribution

Note
This digression is here mainly to motivate the Poisson distribution in the next section; it can be skipped on a first reading.

- Parameters $p_1, \dots, p_n \in [0, 1]$, one probability per trial
- pmf $p(k) = \sum_{A \in F_k} \prod_{i \in A} p_i \prod_{i \notin A} (1 - p_i)$, where F_k is the set of subsets of $\{1, \dots, n\}$ with k elements
- Expected value $\sum_{i=1}^{n} p_i$
- Variance $\sum_{i=1}^{n} p_i(1 - p_i)$
- Example application: n components from different batches, each with its own failure probability – how many of them fail?
- In `scipy.stats`: no object of its own; the pmf is easy enough to compute yourself.

We are no longer tossing the same coin n times but n different coins, each with its own success probability p_i. The $X_i \sim \mathrm{Ber}(p_i)$ are still independent, just no longer identically distributed. Expected value and variance therefore follow exactly as for the binomial distribution, from linearity and from the variance rule for independent summands; when all the p_i are equal, as for the binomial distribution, we get np and $np(1 - p)$ back.

The pmf itself is unwieldy – the sum runs over $\binom{n}{k}$ subsets. In practice it is computed iteratively: start with "no coins, no successes, with probability 1" and work the coins in one at a time.

The interesting case is many small p_i. If we set every $p_i = \frac{\lambda}{n}$ and let n grow, the expected value stays fixed at λ while each individual trial becomes ever less likely. This is the passage to the Poisson distribution:

Theorem
Let $\lambda > 0$ and let $(p_n)_n$ be a sequence in $[0, 1]$ with $n\,p_n \to \lambda$ as $n \to \infty$. Then for every fixed $k \in \mathbb{N}_0$

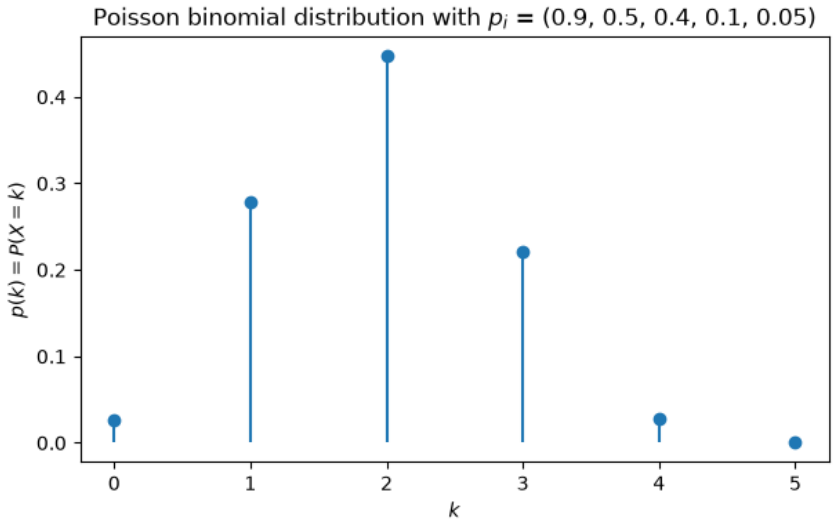
$$\lim_{n \to \infty} \binom{n}{k} p_n^k (1 - p_n)^{n-k} = e^{-\lambda} \frac{\lambda^k}{k!}. \tag{74}$$

Without proof

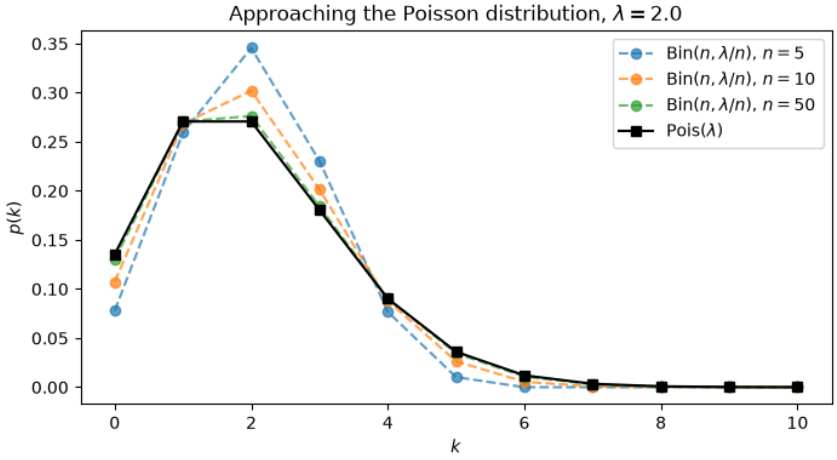
```
def poisson_binomial_pmf(ps):
    """pmf of a Poisson binomial distribution, built up iteratively.

    We start with "no coins, no successes" and work the coins in one at a time:
    distribution[k] is the probability of k successes among the coins considered so far.
    """
    distribution = np.array([1.0])
    for p in ps:
        without_success = np.append(distribution * (1 - p), 0.0)
        with_success = np.append(0.0, distribution * p)
        distribution = without_success + with_success
    return distribution


ps = np.array([0.9, 0.5, 0.4, 0.1, 0.05])
pmf = poisson_binomial_pmf(ps)
k = np.arange(len(pmf))

print("sum of the pmf =", pmf.sum())
print("E(X) =", np.sum(k * pmf), " expected:", ps.sum())
print("V(X) =", np.sum(k**2 * pmf) - np.sum(k * pmf) ** 2, " expected:", np.sum(ps * (1 - ps)))
```

```
fig, ax = plt.subplots(figsize=(7, 4))
ax.stem(k, pmf, basefmt=" ")
ax.set_xlabel("$k$")
ax.set_ylabel("$p(k) = P(X = k)$")
ax.set_title(f"Poisson binomial distribution with $p_i$ = {tuple(ps.tolist())}")
plt.show()
```

```
sum of the pmf = 1.0
E(X) = 1.9500000000000004  expected: 1.95
V(X) = 0.7174999999999998  expected: 0.7174999999999999
```




```
# The passage to the limit: all p_i equal to lambda/n, n grows.
lam = 2.0
k = np.arange(0, 11)

fig, ax = plt.subplots(figsize=(8, 4))
for n in (5, 10, 50):
    ax.plot(k, st.binom(n, lam / n).pmf(k), "o - -", alpha=0.7, label=f"$\\operatorname{{Bin}}(n
    , \\lambda/n)$, $n={n}$")
ax.plot(k, st.poisson(lam).pmf(k), "s -", color="black", label="$\\operatorname{Pois}(\\lambda)$
    ")
ax.set_xlabel("$k$")
ax.set_ylabel("$p(k)$")
ax.set_title(f"Approaching the Poisson distribution, $\\lambda = {lam}$")
ax.legend()
plt.show()
```



2.5.8 Poisson distribution

- Parameter $\lambda > 0$, notation $\text{Pois}(\lambda)$
- pmf $p(k) = e^{-\lambda}\dfrac{\lambda^k}{k!}$ for $k \in \mathbb{N}_0$
- Expected value λ
- Variance λ
- Example applications:
  - How many requests reach the server in one second?
  - How many typos are there on one page?
  - How many atomic nuclei in a specimen decay in one minute?
- In `scipy.stats`: `st.poisson(mu=`λ`)`

The guiding question: how many events occur in a fixed period of time, if they occur independently of one another and at a constant mean rate? In contrast to every distribution in this chapter so far, this one does not have finite support – in principle there could be arbitrarily many events, even if large numbers become very unlikely.

That the pmf sums to 1 follows from the exponential series $\sum_{k\geq 0}\frac{\lambda^k}{k!} = e^{\lambda}$:

$$\sum_{k=0}^{\infty} e^{-\lambda}\frac{\lambda^k}{k!} = e^{-\lambda}e^{\lambda} = 1. \tag{75}$$

The expected value follows with an index shift – the summand for $k = 0$ is 0 and drops out:

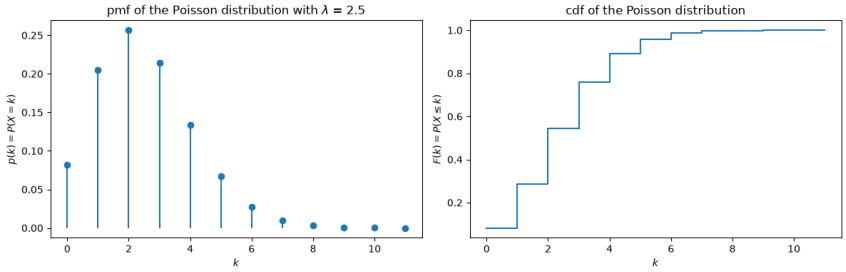
$$\mathbb{E}(X) = \sum_{k=1}^{\infty} k\, e^{-\lambda}\frac{\lambda^k}{k!} = \lambda e^{-\lambda}\sum_{k=1}^{\infty}\frac{\lambda^{k-1}}{(k-1)!} = \lambda e^{-\lambda}e^{\lambda} = \lambda. \tag{76}$$

For the variance, compute $\mathbb{E}(X(X-1)) = \lambda^2$ in just the same way, which gives $\mathbb{V}(X) = \mathbb{E}(X^2) - \lambda^2 = \lambda^2 + \lambda - \lambda^2 = \lambda$. So here expected value and variance coincide – a practical test of whether a Poisson distribution can fit a dataset at all.

> **Remark**
> Count events with $\text{Pois}(\lambda)$ per unit of time, and the *waiting time* until the next event is $\text{Exp}(\lambda)$-distributed. With "How long does it take until the next packet arrives at the server?", the exponential distribution in the chapter The most important continuous distributions answers exactly the continuous flip side of the same question.

```
lam = 2.5
X = st.poisson(lam)
print("E(X) =", X.mean(), " V(X) =", X.var(), " (both equal to lambda)")
print("P(X > 2) =", X.sf(2), " by hand:", 1 - np.exp( -lam) * (1 + lam + lam**2 / 2))

plotXdiscrete(np.arange(0, 12), X, "Poisson distribution", (f"$\\lambda$ = {lam}",))
```

```
E(X) = 2.5  V(X) = 2.5  (both equal to lambda)
P(X > 2) = 0.45618688411667035  by hand: 0.4561868841166704
```



2.5.9 For completeness: the geometric distribution

- Parameter $p \in (0, 1]$, notation $\mathrm{Geom}(p)$
- pmf $p(k) = (1-p)^{k-1}p$ for $k \in \{1, 2, 3, \dots\}$
- Expected value $\dfrac{1}{p}$
- Variance $\dfrac{1-p}{p^2}$
- Example applications:
  - How many times do I have to throw a die until the first six comes up? ($p = \frac{1}{6}$, on average 6 throws)
  - How many connection attempts does it take until the first success?
- In `scipy.stats`: `st.geom(p)`

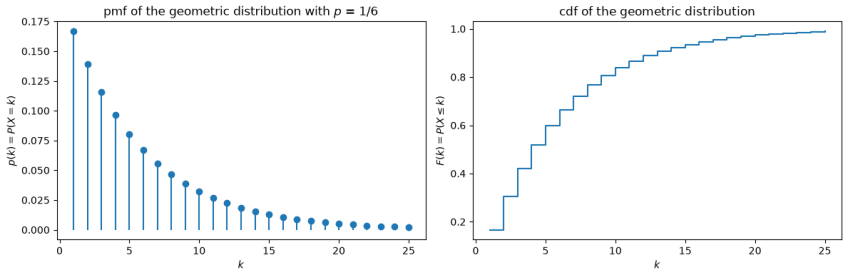
We repeat a Bernoulli experiment until the first success and count the trials. For the first $k-1$ trials to fail and the k-th to succeed has probability $(1-p)^{k-1}p$ – hence the pmf.

The geometric distribution is *memoryless*: if you have already thrown the die ten times in vain, the probability of needing another k throws is the same as it was at the start. The die does not remember. It is thus the discrete counterpart of the exponential distribution, the only memoryless distribution in the continuous case.

> **Caution**
> There are two conventions in common use: the number of **trials up to the first success** (then $k \geq 1$, expected value $\frac{1}{p}$) and the number of **failures before it** (then $k \geq 0$, expected value $\frac{1-p}{p}$). We use the first, and so does `scipy.stats.geom`.

```
p = 1 / 6
X = st.geom(p)
print("E(X) =", X.mean(), " expected:", 1 / p)
print("V(X) =", X.var(),  " expected:", (1 - p) / p**2)

plotXdiscrete(np.arange(1, 26), X, "geometric distribution", (f"$p$ = 1/6",))
```

```
E(X) = 6.0  expected: 6.0
V(X) = 30.000000000000007  expected: 30.000000000000004
```



2.5.10 Looking back

All the distributions in this chapter hang together:

- The **categorical distribution** is the most general distribution on a finite set; its parameter space is the simplex Δ_n.
- The **discrete uniform distribution** is the centre of that simplex and has maximal entropy.
- The **Bernoulli distribution** is the case $n = 2$.

- The **binomial distribution** counts the successes in n independent Bernoulli trials with the same p, the **Poisson binomial distribution** allows different p_i.
- The **Poisson distribution** is their limiting case for many trials, each of them very unlikely.
- The **geometric distribution** turns the question around: not how many successes in a fixed number of trials, but how many trials up to the first success.

Next we leave countable sets of outcomes behind. Look at the binomial distribution not for small p but for large n, and the passage to the limit leads not to the Poisson distribution but to the **normal distribution** – that is the de Moivre–Laplace theorem, and the chapter The normal distribution opens with it.

2.6 Titanic EDA with Numpy, a first pass

In this notebook we use nothing but Numpy (no Pandas, no scikit-learn or the like) to carry out a discrete analysis of the Kaggle Titanic dataset. The obvious tasks are exploratory data analysis (EDA) and modelling the variable 'probability of survival' for the instances (an extract from the Titanic's passenger list). Along the way, and by way of motivation, we also take a first look at a multivariate analysis, even though we have not yet covered the theory behind it.

2.6.1 Loading the data

To begin with, we naively load the comma-separated file from Kaggle (the one that comes with a header row) and see what happens.

```
import numpy as np

filename = 'titanic.csv'

try:
    data = np.genfromtxt(
        filename,
        delimiter=',',
        dtype=None,
        encoding='utf -8',
        names=True
    )
except ValueError as e:
    print("\n".join(e.args[0].split("\n")[:3]), "\n\t...")
```

```
Some errors were detected !
    Line #2 (got 13 columns instead of 12)
    Line #3 (got 13 columns instead of 12)
        ...
```

Evidently line 2 contains more variables than the header row!

Let us print the first two lines and inspect them by hand:

```
with open(filename, "r") as file:
    for i in range(2):
        print(file.readline())
```

```
PassengerId,Survived,Pclass,Name,Sex,Age,SibSp,Parch,Ticket,Fare,Cabin,Embarked

1,0,3,"Braund, Mr. Owen Harris",male,22,1,0,A/5 21171,7.25,,S
```

The fix is a CSV parser that understands quotation marks: in `"Braund, Mr. Owen Harris"` the comma is part of the name, not a separator. The `csv` module in the Python standard library can do that (and so can `pandas.read_csv`, which we will meet later). `np.genfromtxt` knows nothing about quotation marks, so we write the file out once as a TSV (tab-separated values). None of the values contains a tab, so the TSV no longer needs quotation marks:

```
import csv

with open("titanic.csv", newline="", encoding="utf -8") as source, \
     open("titanic.tsv", "w", encoding="utf -8") as target:
    for row in csv.reader(source):          # reads "Braund, Mr. Owen Harris" as one field
        assert not any("\t" in field for field in row)   # otherwise we would need quotation
    marks again
        target.write("\t".join(row) + "\n")

filename = 'titanic.tsv'

data = np.genfromtxt(
    filename,
    delimiter='\t',
    dtype=None,
```

```
    encoding='utf -8',
    names=True
)

print("Columns:", data.dtype.names)
print("First 5 rows:")
print(data[:5])
```

```
Columns: ('PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch', 'Ticket',
     'Fare', 'Cabin', 'Embarked')
First 5 rows:
[(1, 0, 3, 'Braund, Mr. Owen Harris', 'male', 22., 1, 0, 'A/5 21171',  7.25  , '', 'S')
 (2, 1, 1, 'Cumings, Mrs. John Bradley (Florence Briggs Thayer)', 'female', 38., 1, 0, 'PC
    17599', 71.2833, 'C85', 'C')
 (3, 1, 3, 'Heikkinen, Miss. Laina', 'female', 26., 0, 0, 'STON/O2. 3101282',  7.925 , '', 'S')
 (4, 1, 1, 'Futrelle, Mrs. Jacques Heath (Lily May Peel)', 'female', 35., 1, 0, '113803', 53.1
     , 'C123', 'S')
 (5, 0, 3, 'Allen, Mr. William Henry', 'male', 35., 0, 0, '373450',  8.05  , '', 'S')]
```

2.6.2 Selecting the relevant features

```
survived = data['Survived'].astype(int)
pclass = data['Pclass'].astype(int)
sex = data['Sex']
age = data['Age']  # may contain nan
sibsp = data['SibSp'].astype(int)
parch = data['Parch'].astype(int)
embarked = data['Embarked']

used_columns = {'Survived', 'Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Embarked'}
all_columns = set(data.dtype.names)
unused_columns = all_columns - used_columns
print("Unused columns:", unused_columns)
```

```
Unused columns: {'Fare', 'Cabin', 'PassengerId', 'Ticket', 'Name'}
```

2.6.3 NaN = Not a Number

Replacing missing values with the median or dropping those rows: either is a decision that distorts the data. How to take such decisions systematically and put them on record is covered in the chapter Data cleaning.

```
print("\nNumber of missing age values:", np.sum(np.isnan(age)))

# replace missing age values by the median (just as an example)
non_nan_age = age[~np.isnan(age)]
age_filled = age.copy()
age_filled[np.isnan(age)] = np.median(non_nan_age)

print("Missing age values replaced by the median:", np.median(non_nan_age))
```

```
Number of missing age values: 177
Missing age values replaced by the median: 28.0
```

2.6.4 Analysing survival

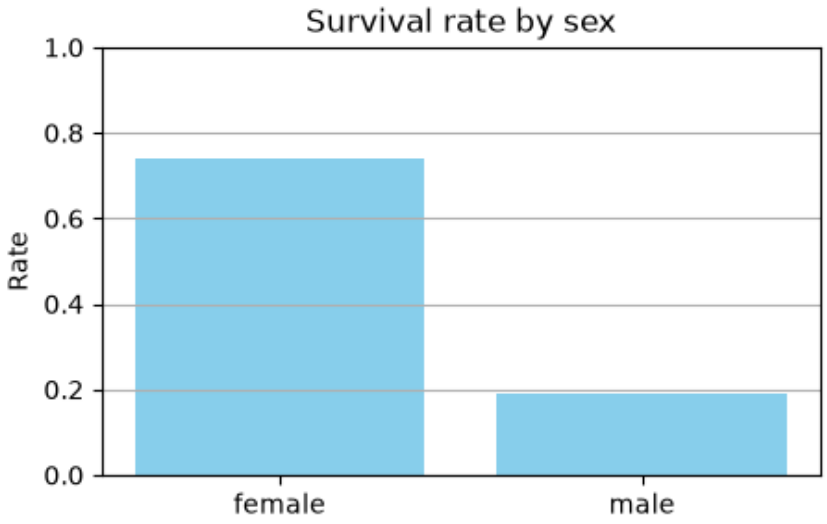
```
# new feature
def survival_rate(mask):
    total = np.sum(mask)
    survived_total = np.sum(survived[mask])
    rate = survived_total / total if total > 0 else 0
```

```
    return total, survived_total, rate

# group by Sex
for s in np.unique(sex):
    mask = (sex == s)
    total, survived_s, rate = survival_rate(mask)
    print(f"Sex: {s:6s} | Total: {total:3d} | Survived: {survived_s:3d} | Rate: {rate:.2f}")

# group by passenger class
for c in np.unique(pclass):
    mask = (pclass == c)
    total, survived_c, rate = survival_rate(mask)
    print(f"Pclass: {c} | Total: {total:3d} | Survived: {survived_c:3d} | Rate: {rate:.2f}")

# Binning
missing = np.isnan(age)  # keep track of the missing entries separately, no sentinel value!

age_group = np.full(age.shape, 'Unknown', dtype='<U10')
age_group[~missing] = 'Adult'
age_group[~missing & (age < 13)] = 'Child'
age_group[~missing & (age >= 60)] = 'Senior'

for grp in np.unique(age_group):
    mask = (age_group == grp)
    total, survived_grp, rate = survival_rate(mask)
    print(f"Age Group: {grp:7s} | Total: {total:3d} | Survived: {survived_grp:3d} | Rate: {rate
    :.2f}")
```

```
Sex: female | Total: 314 | Survived: 233 | Rate: 0.74
Sex: male   | Total: 577 | Survived: 109 | Rate: 0.19
Pclass: 1 | Total: 216 | Survived: 136 | Rate: 0.63
Pclass: 2 | Total: 184 | Survived:  87 | Rate: 0.47
Pclass: 3 | Total: 491 | Survived: 119 | Rate: 0.24
Age Group: Adult   | Total: 619 | Survived: 243 | Rate: 0.39
Age Group: Child   | Total:  69 | Survived:  40 | Rate: 0.58
Age Group: Senior  | Total:  26 | Survived:   7 | Rate: 0.27
Age Group: Unknown | Total: 177 | Survived:  52 | Rate: 0.29
```

2.6.5 Visualisation: survival by sex

```
import matplotlib.pyplot as plt

labels = []
rates = []
for s in np.unique(sex):
    mask = (sex == s)
    _, _, rate = survival_rate(mask)
    labels.append(s)
    rates.append(rate)

plt.figure(figsize=(5, 3))
plt.bar(labels, rates, color='skyblue')
plt.title('Survival rate by sex')
plt.ylabel('Rate')
plt.ylim(0, 1)
plt.grid(axis='y')
plt.show()
```



2.6.6 A first classifier

The survival rates above suggest a rule: "women survive, men do not". A rule that turns the features of a row into a **label** (here the column `Survived`) is called a **classifier**. The label is the column we want to predict; for the rows we already have it is known, and that is the *ground truth*. Before we learn any procedure that finds such a rule from data on its own, let us see how a rule is assessed. All that takes is masks:

```
prediction = (sex == 'female')    # prediction: survived
truth = (survived == 1)           # ground truth

tp = np.sum( prediction &  truth)   # true positive:  predicted survived, did survive
fp = np.sum( prediction & ~truth)   # false positive: predicted survived, died
fn = np.sum(~prediction &  truth)   # false negative: predicted died, did survive
tn = np.sum(~prediction & ~truth)   # true negative:  predicted died, did die

print("                   predicted: died  survived")
print(f"actually died:                {tn:4d}      {fp:4d}")
print(f"actually survived:            {fn:4d}      {tp:4d}")
assert tp + fp + fn + tn == len(survived)
```

```
predicted: died  survived
actually died:                468        81
actually survived:            109       233
```

This 2×2 table is called the **confusion matrix**. It is nothing more than a cross-tabulation of two nominal variables, ground truth against prediction – exactly the kind of summary measure the chapter Descriptive statistics allows for nominal features. Three summary measures in common use can be read off it:

- **Accuracy**: the proportion of correct predictions, $\frac{tp + tn}{n}$.
- **Precision**: when we predict "survived", how often is that right? $\frac{tp}{tp + fp}$.
- **Recall** (in medicine *sensitivity*): of those who actually survived, how many do we catch? $\frac{tp}{tp + fn}$.

Precision and recall are conditional probabilities, just read in opposite directions: precision is $P(\text{survived} \mid \text{predicted})$, recall is $P(\text{predicted} \mid \text{survived})$. That they are two different numbers is something we will meet again under Bayes' theorem.

```
n = len(survived)
```

```
def summary_measures(prediction):
    tp = np.sum( prediction &  truth); fp = np.sum( prediction & ~truth)
    fn = np.sum(~prediction &  truth); tn = np.sum(~prediction & ~truth)
    accuracy  = (tp + tn) / n
    precision = tp / (tp + fp) if tp + fp > 0 else np.nan   # undefined without a positive
    prediction
    recall    = tp / (tp + fn)
    return accuracy, precision, recall

for name, pred in [("women survive", sex == 'female'),
                   ("nobody survives", np.zeros(n, dtype=bool))]:
    acc, prec, rec = summary_measures(pred)
    print(f"{name:18s} Accuracy {acc:.3f}   Precision {prec:.3f}   Recall {rec:.3f}")
```

```
women survive      Accuracy 0.787   Precision 0.742   Recall 0.681
nobody survives    Accuracy 0.616   Precision nan   Recall 0.000
```

The second row is the stupidest classifier there is: it predicts the most frequent class for every row. Even so, it is right 62% of the time, because only 38% of the passengers survived. This shows that accuracy on its own says little; it always has to be held up against the **base rate** of the most frequent class. Recall and precision expose the rule at once: recall 0, precision undefined.

The stupidest classifier is, incidentally, an old acquaintance. In the chapter Expected value and variance, the median and the mean were the best *constant* models for the absolute and the squared error. For a nominal label the best constant model is the **mode**, the most frequent class, and the error it minimises is the misclassification rate 1 − Accuracy.

The rule can be refined – following the maxim "women and children first", for instance:

```
acc, prec, rec = summary_measures((sex == 'female') | (~missing & (age < 13)))
print(f"Women and children Accuracy {acc:.3f}   Precision {prec:.3f}   Recall {rec:.3f}")
```

```
Women and children Accuracy 0.792   Precision 0.724   Recall 0.743
```

Accuracy barely changes, but recall rises and precision falls: the new rule catches more of the survivors, but it also mistakes more of the dead for survivors. Which of the two measures matters depends on the question. If you are planning lifeboats you want to overlook no survivor (recall); if you are sorting out spam you want to throw away no genuine email (precision). Because a single number is often what is needed, the two are combined into the **F1 score**, the harmonic mean of precision and recall. In the chapter Model selection we let scikit-learn compute this table for us.

We have allowed ourselves one thing here that must never be repeated later on: we assessed the rule on the very data we dreamt it up on. How to hold data back so that a rule can be assessed honestly is also in the chapter Model selection, under *train-test split*.

Two directions The confusion matrix holds one more lesson, which we only look at here and will not prove until the chapter Covariance and dependence. The precision of the rule "women survive" is the proportion of survivors among the women; the recall is the proportion of women among the survivors:

```
p_survived_if_female = np.mean(survived[sex == 'female'])         # P(survived | female) =
    precision
p_female_if_survived = np.mean(sex[survived == 1] == 'female')    # P(female | survived) =
    recall
p_female = np.mean(sex == 'female')    # base rate: proportion of women
p_survived = np.mean(survived)         # base rate: survival rate

print(f"P(survived | female) = {p_survived_if_female:.3f}")
print(f"P(female | survived) = {p_female_if_survived:.3f}")
print(f"Conversion: {p_female_if_survived:.3f} * {p_survived:.3f} / {p_female:.3f}"
      f" = {p_female_if_survived * p_survived / p_female:.3f}")
```

```
P(survived | female) = 0.742
P(female | survived) = 0.681
Conversion: 0.681 * 0.384 / 0.352 = 0.742
```

The two conditional probabilities differ, and the conversion factor between them is the ratio of the two base rates. That, with numbers we counted ourselves, is **Bayes' theorem**.

2.6.7 A look at correlations

We have not yet formally introduced what a correlation is, but let us quickly compute the so-called correlation coefficient all the same. How to interpret what we see here can wait until later.

```
is_female = (sex == 'female').astype(int)
is_male = (sex == 'male').astype(int)

corf = np.corrcoef(survived, is_female)[0, 1]
corm = np.corrcoef(survived, is_male)[0, 1]

print(f"Correlation between being female and survival: {corf:.2f}")
print(f"Correlation between being male and survival: {corm:.2f}")

Correlation between being female and survival: 0.54
Correlation between being male and survival: -0.54
```

3 Continuous stochastics and Scipy

3.1 Continuous distributions and density functions

Definition
Let $X\colon \Omega \to \mathbb{R}$ be a random variable, that is, let Ω be a probability space (a set Ω together with a probability measure P_Ω) and let X be a map such that $P(X \in A) := P_\Omega(X^{-1}(A))$ is defined for all events $A \subseteq \mathbb{R}$.

If Ω is not a discrete probability space (because Ω is uncountably infinite, say $\Omega = \mathbb{R}^n$), if X takes uncountably many possible values, and if the distribution of X has a *density* (defined in the next section), then we call X a *continuous random variable*. A random variable with uncountably many values but no density – a mixture of a Dirac and a normal distribution, for instance – is therefore neither discrete nor continuous.

Definition
The *support* of X is the smallest **closed** set $A \subseteq \mathbb{R}$ with $P(X \in A) = 1$,

$$\operatorname{supp}(X) = \bigcap \{A \subseteq \mathbb{R} \text{ closed} : P(X \in A) = 1\}. \tag{77}$$

The word *closed* is not decoration here. Without it the intersection would be empty for every distribution with a density: for $X \sim U([0,1])$ the set $[0,1] \setminus \{x\}$ still has probability 1 for every single x (individual points are null sets – more on that shortly), so no point at all would lie in all of these sets. Insist on closed sets and $[0,1]$ comes out, as it should. (That the intersection itself has probability 1 again, because countably many of the A already suffice, is something we will not need.)

Definition
A set $M \subseteq \mathbb{R}^n$ is called *closed* if it contains the limits of its own convergent sequences: if a sequence (x_k) lies entirely in M and converges to x, then $x \in M$ as well.

Every interval of the form $[a,b]$ is closed, for instance, as is all of $\mathbb{R}$ and every finite set. The interval $(0,1]$, by contrast, is not: the sequence $\frac{1}{k}$ lies in it, its limit 0 does not. The closure of $(0,1]$ is $[0,1]$.

For a discrete distribution with pmf p and finitely many values, the support simplifies to

$$\operatorname{supp}(X) = \{\, x \;\mid\; p(x) > 0 \,\}, \tag{78}$$

that is, to the set of values that can occur at all (with infinitely many values, their accumulation points come along too, since the support is closed).

Example
The uniform distribution on $\Omega = \{1, \ldots, n\}$, read as a distribution *on* Ω, has support Ω itself – every element of it is possible.

The same distribution can also be read as a distribution on $\mathbb{R}$, by giving every other real number probability 0. Then the support is $\{1, \ldots, n\} \subset \mathbb{R}$, a finite and hence closed subset of the reals. So the support depends not only on the distribution, but also on *what* you read it as a distribution on.

The support is defined for discrete real random variables too – the obvious difference being that the support of a discrete random variable contains no interval.

3.1.1 Density functions

Definition
Let $D \subseteq \mathbb{R}^n$ and let $f\colon D \to [0,\infty)$ be an integrable function (with respect to the Lebesgue measure on $\mathbb{R}^n$; Riemann integrability suffices) with $\int_D f(x)dx = 1$. Then a probability measure on D is defined by:

$$\text{for an event } A \subseteq D\colon \quad P(A) := \int_A f(x)dx \tag{79}$$

Whenever a probability measure P can be written in this way, we call the function f a *probability density function* (pdf), or a *density* for short (not to be confused with the probability mass function, pmf, of the discrete case).

So for a real random variable $X\colon \Omega \to \mathbb{R}$ whose distribution P_X has a density f, we have for $A \subseteq \mathbb{R}$

$$\int_{X^{-1}(A)} dP(\omega) = P(X \in A) = P_X(A) = \int_A f(x)dx \tag{80}$$

The integral on the right is now an integral over the range of X. In the same way we can also compute the expected value and the variance:

Definition
Let X be a continuous random variable whose distribution is given by a density $f\colon D \to [0,\infty)$. Then

$$\mathbb{E}(X) = \int_\Omega X(\omega)\, dP(\omega) = \int_D x f(x)dx \tag{81}$$

is called the *expected value* of X, and

$$\mathbb{V}(X) = \int_\Omega (X(\omega) - \mathbb{E}(X))^2\, dP(\omega) = \int_D (x - \mathbb{E}(X))^2\, f(x)dx \tag{82}$$

the *variance* of X, provided these integrals exist. For $D = \mathbb{R}$, for example, $\mathbb{E}(X) = \int_{-\infty}^{\infty} x f(x)dx$.

This is the continuous counterpart of the sum over the probability mass function in the chapter Expected value and variance: the density takes the place of the pmf, the integral the place of the sum. That other functions of X may be integrated in the same way is what the following theorem says.

Theorem (transformation formula)
Let X be a continuous random variable with density $f\colon D \to [0,\infty)$ and let $g\colon D \to \mathbb{R}$ be a (measurable) function. Then

$$\mathbb{E}(g(X)) = \int_D g(x)f(x)dx, \tag{83}$$

provided the integral exists. For a discrete random variable with pmf p, the same statement reads $\mathbb{E}(g(X)) = \sum_x g(x)\, p(x)$.

We do not prove the theorem (it follows from the change-of-variables theorem for integrals), but we use it constantly: the definition of the variance is precisely the special case $g(x) = (x - \mathbb{E}(X))^2$, and with $g(x) = x^2$ we obtain, as in the discrete case,

$$\mathbb{V}(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \int_D x^2 f(x)dx - (\mathbb{E}(X))^2. \tag{84}$$

In general, $\mathbb{E}(X^k) = \int_D x^k f(x)dx$ is called the *k-th moment* of X.

Example
Let $a < b \in \mathbb{R}$. Take $D = \mathbb{R}$ and $f\colon \mathbb{R} \to [0,\infty)$ given by

$$f(x) = \begin{cases} \frac{1}{b-a}, & \text{if } x \in [a,b] \\ 0 & \text{otherwise} \end{cases} \tag{85}$$

This f defines a probability measure on $\mathbb{R}$, the *continuous uniform distribution* on the interval $[a, b]$. Let X be a random variable with this distribution; we write $X \sim U([a, b])$. Its expected value (using $\int x dx = \frac{x^2}{2}$) is $\mathbb{E}(X) =$

$$\int_{-\infty}^{\infty} x f(x) dx = \int_a^b \frac{x dx}{b-a} = \frac{1}{b-a}\left[\frac{x^2}{2}\right]_a^b = \frac{b^2-a^2}{2(b-a)} = \frac{(b+a)(b-a)}{2(b-a)} = \frac{a+b}{2} \tag{86}$$

The second moment is

$$\mathbb{E}(X^2) = \int_{-\infty}^{\infty} x^2 f(x) dx = \left[\frac{x^3}{3(b-a)}\right]_a^b = \frac{b^3-a^3}{3(b-a)} = \frac{a^2+ab+b^2}{3} \tag{87}$$

The variance is

$$\mathbb{V}(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \frac{a^2+ab+b^2}{3} - \frac{(a+b)^2}{4} = \frac{(a-b)^2}{12} \tag{88}$$

Remark

Null sets. For $X \sim U([a, b])$ and $a \le c < d \le b$ we have $P(c \le X \le d) = \int_c^d \frac{dx}{b-a} = \frac{d-c}{b-a}$, and the same integral comes out for $P(c < X < d)$, $P(c \le X < d)$ and $P(c < X \le d)$: whether or not the endpoints belong to the interval makes no difference to the integral. In particular $P(X = c) = \int_c^c f(x)dx = 0$ for every single point c, even though $f(c) = \frac{1}{b-a} \neq 0$.

This holds for every distribution with a density: individual points have probability 0; by finite additivity so does every finite set; and by the σ-additivity from the chapter Deriving the mathematical formalism of stochastics so does every countable set $A = \{a_1, a_2, \dots\}$, since $P(X \in A) = \sum_i P(X = a_i) = 0$. Sets with probability 0 are called *null sets*. Change a density f on a null set (at finitely many points, say) and not one integral $\int_A f(x)dx$ changes – and so neither does the probability measure it defines.

Example

The set $\mathbb{Q} \cap [0, 1]$ of rational numbers between 0 and 1 is countable, and hence a null set for $X \sim U([0, 1])$: a uniformly distributed random number from $[0, 1]$ is irrational with probability 1. A puzzle: is there any distribution on $[0, 1]$ at all that gives every single point positive probability? (No: for each k there are at most k points with probability at least $\frac{1}{k}$, so at most countably many points in all with positive probability.)

Example

Not every probability measure can be represented by a density: consider the distribution on $\mathbb{R}$ with

$$P(A) = \begin{cases} 1 & \text{if } 0 \in A \\ 0 & \text{otherwise} \end{cases} \tag{89}$$

This distribution is called the *Dirac distribution* with mass at 0. There can be no density f for it: the one-point set $\{0\}$ is a null set for every density, so we would have $\int_{\{0\}} f(x)dx = 0 \neq 1 = P(\{0\})$. The same argument hits every discrete distribution on $\mathbb{R}$: it concentrates its whole mass 1 on its countable support, and that is a null set for every density. If you go to great lengths to build something density-like anyway, you end up in the theory of distributions – generalised functions – from functional analysis (physicists know them). And there are probability distributions for which not even a generalised function will do.

Example

Conversely, a continuous distribution is easily built out of a discrete one. Let Y be a discrete

random variable with values in $\{5,6,7\}$ and $P(Y=5)=\frac{4}{7}$, $P(Y=6)=\frac{2}{7}$, $P(Y=7)=\frac{1}{7}$, and let $Z\sim U([0,1))$ be independent of Y (formally: $\Omega=\{5,6,7\}\times[0,1)$ with the product measure, $X(y,z)=y+z$). Then $X:=Y+Z$ is a continuous random variable with values in $[5,8)$ and density

$$f(x)=\begin{cases}\frac{4}{7}, & \text{if } x\in[5,6)\\ \frac{2}{7}, & \text{if } x\in[6,7)\\ \frac{1}{7}, & \text{if } x\in[7,8)\\ 0 & \text{otherwise,}\end{cases} \tag{90}$$

since for $x\in[5,6)$ we have $P(5\le X\le x)=P(Y=5)\cdot P(Z\le x-5)=\frac{4}{7}(x-5)$, and likewise on the other two subintervals. The density integrates to $\frac{4}{7}+\frac{2}{7}+\frac{1}{7}=1$; what value f takes at the points 6 and 7 does not matter, by the remark on null sets. For example $P(5\le X\le 6)=\int_5^6 f(x)dx=\frac{4}{7}$ and $P(6\le X\le 6.5)=\frac{2}{7}\cdot\frac{1}{2}=\frac{1}{7}$. The expected value is

$$\mathbb{E}(X)=\int_5^8 x f(x)dx=\frac{4}{7}\cdot\frac{11}{2}+\frac{2}{7}\cdot\frac{13}{2}+\frac{1}{7}\cdot\frac{15}{2}=\frac{85}{14}=6+\frac{1}{14}, \tag{91}$$

since on every subinterval $[k,k+1)$ we have $\int_k^{k+1} x\,dx=k+\frac{1}{2}$ (the same comes out of $\mathbb{E}(Y)+\mathbb{E}(Z)=\frac{39}{7}+\frac{1}{2}$). A distribution like this – a discrete choice first, then a continuous one within the chosen class – is called a *mixture distribution*; another example is the contaminated normal distribution in the chapter The most important continuous distributions, where this very distribution on $[5,8)$ is simulated too.

Definition
Let (Ω,P) be a probability space. Then for a random variable $X\colon\Omega\to\mathbb{R}$ we call the function

$$F\colon\mathbb{R}\to[0,1],\qquad x\mapsto P(X\le x) \tag{92}$$

the *cumulative probability distribution function*, or *distribution function*, or *cdf* for short. Unlike the density (pdf), it always exists.

Example
If P_X has a density $f\colon\mathbb{R}\to[0,\infty)$, so that $P(a\le X\le b)=\int_a^b f(x)dx$, then

$$F(x)=\int_{-\infty}^{x} f(t)dt. \tag{93}$$

Whether you write $P(X\le x)$ or $P(X<x)$ here makes no difference, by the remark on null sets; some books define the cdf with $P(X<x)$, and for distributions with a density that is the same function.

Theorem
If P_X has a density f, then the distribution function F is an antiderivative of f: by the fundamental theorem of calculus, F is differentiable at every point x at which f is continuous, with $F'(x)=f(x)$. For every random variable (with or without a density) the cdf also gives the probability of intervals,

$$P(a<X\le b)=F(b)-F(a), \tag{94}$$

and with a density that is precisely $\int_a^b f(t)dt$.

So the cdf climbs steeply exactly where the density is large. In the chapter The most important continuous distributions the pdf and the cdf are always plotted together; look out for it there.

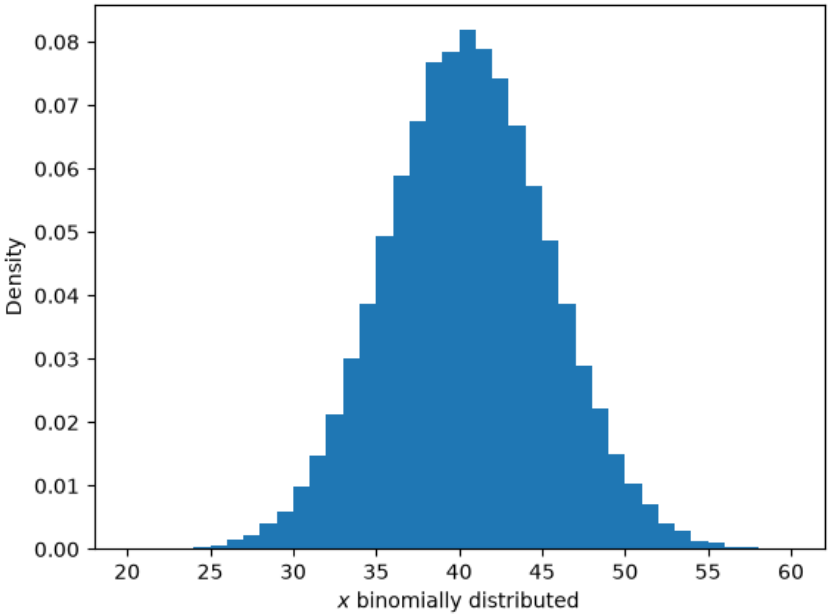
3.2 The Gaussian normal distribution and the central limit theorem

3.2.1 Approximating the binomial distribution

Consider a sequence of Bernoulli distributed random variables X_i with the same parameter p, and let $S_n := \sum_{i=1}^{n} X_i$ be the sum of the first n of them. From the chapter The most important discrete distributions we know that $S_n \sim \mathrm{Bin}(n,p)$: the sum is binomially distributed. As a reminder, this means the pmf of S_n is $p(k) = \binom{n}{k} p^k (1-p)^{n-k}$.

For the concrete values $n = 100,\ p = 0.4$, a histogram of 100000 samples looks something like this:

```
import numpy as np
from scipy.stats import binom, norm
from matplotlib import pyplot as plt
np.random.seed(123123)
binom_samples = binom.rvs(n=100,
                          p=0.4,
                          size=100000)
plt.hist(binom_samples, bins=40, density=True)
plt.xlabel("$x$ binomially distributed")
plt.ylabel("Density")
plt.show()
```



Since large samples always produce this shape, whatever p may be (try it out!), you might suspect that the distribution can be approximated by a function that is easier to compute than the binomial coefficients $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Aside: there is in fact *Stirling's formula* for approximating the factorial,

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{95}$$

which for binomial coefficients gives us directly: $\binom{n}{k} \approx \frac{\sqrt{n}}{\sqrt{2\pi}\sqrt{k}\sqrt{n-k}} \frac{n^n}{k^k (n-k)^{(n-k)}}$

With this, an approximation can be proved (though it is not entirely easy):

$$P(S_n = k) = \binom{n}{k} p^k (1-p)^{n-k} \sim \frac{1}{\sqrt{np(1-p)}} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad \text{for } n \to \infty \tag{96}$$

where on the right-hand side the sequence $x = x_n$ with $x_n = x_n(k) = \frac{k-np}{\sqrt{np(1-p)}}$ has to be substituted for x. The symbol $\sim$ means asymptotically equal: the quotient converges to 1.

The approximation is useful because we can now tabulate the function $e^{-\frac{x^2}{2}}$ and read off concrete values for any given n, p – considerably more efficient than computing binomial coefficients. The whole observation is known as the **theorem** of de Moivre–Laplace, and for now it serves as our motivation for examining the function $e^{-\frac{x^2}{2}}$ more closely.

3.2.2 An integral

To turn the function $e^{-\frac{x^2}{2}}$ into a probability density, its integral has to come out as 1. Since it will be very useful later, we compute a slightly more general integral right away:

Lemma
Let $v \in \mathbb{R}$ with $v > 0$. Then

$$\int_{-\infty}^{\infty} e^{-\frac{x^2}{2v}} dx = \sqrt{2\pi v}. \tag{97}$$

Proof
The square root on the right-hand side suggests squaring both sides of the equation. We then keep rewriting the left-hand side until we can switch an integral over $\mathbb{R}^2$ to polar coordinates and, after a substitution, find antiderivatives easily:

$$\begin{aligned}
&\left(\int_{-\infty}^{\infty} e^{-\frac{x^2}{2v}} dx\right)^2 \\
=&\left(\int_{-\infty}^{\infty} e^{-\frac{x^2}{2v}} dx\right)\left(\int_{-\infty}^{\infty} e^{-\frac{y^2}{2v}} dy\right) \\
=&\int_{-\infty}^{\infty} e^{-\frac{x^2}{2v}} \left(\int_{-\infty}^{\infty} e^{-\frac{y^2}{2v}} dy\right) dx \\
=&\int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} e^{-\frac{x^2}{2v}} e^{-\frac{y^2}{2v}} dy\right) dx \\
=&\int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} e^{-\frac{x^2+y^2}{2v}} dy\right) dx \\
=&\int_{\mathbb{R}^2} e^{-\frac{x^2+y^2}{2v}} d(x,y) \\
=&\int_0^{\infty} \int_0^{2\pi} r e^{-\frac{r^2}{2v}} d\phi dr \\
=&\int_0^{\infty} r e^{-\frac{r^2}{2v}} \int_0^{2\pi} d\phi dr \\
=&\int_0^{\infty} r e^{-\frac{r^2}{2v}} [\phi]_0^{2\pi} dr \\
=&\int_0^{\infty} r e^{-\frac{r^2}{2v}} 2\pi dr \\
=&2\pi \int_0^{\infty} e^{-\frac{r^2}{2v}} r dr \\
=&2\pi \int_0^{-\infty} e^u (-v) du \\
=&2\pi v \int_{-\infty}^{0} e^u du \\
=&2\pi v [e^u]_{-\infty}^{0} \\
=&2\pi v.
\end{aligned} \tag{98}$$

Since integrals turn up rather often in statistics, it is worth working through this calculation and seeing exactly what happens at each step, and why.

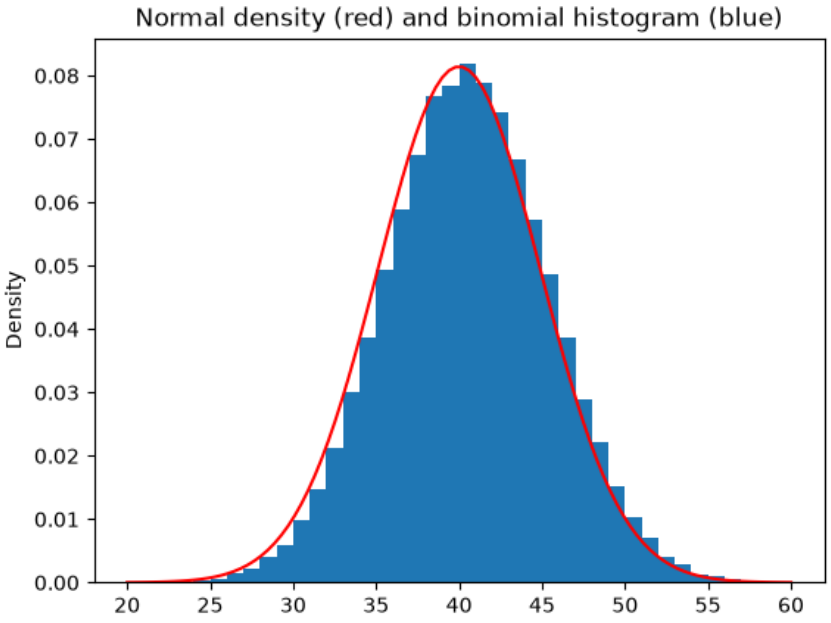
3.2.3 The one-dimensional normal distribution

Definition
Let $\mu, \sigma \in \mathbb{R}$ with $\sigma > 0$. Then the distribution with density function

$$\phi(x) = \phi(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{99}$$

is called the *normal distribution*, and for a real random variable X with this distribution we also write $X \sim \mathcal{N}(\mu, \sigma^2)$.

```
x = np.linspace(20,60,100)
plt.hist(binom_samples, bins=40, density=True)
plt.plot(x, norm(loc=40, scale=np.sqrt(24)).pdf(x), color="red")   # expected value n p = 40,
    variance n p (1 -p) = 24
plt.ylabel('Density')
plt.title('Normal density (red) and binomial histogram (blue)')
plt.show()
```



Proposition
If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $\mathbb{E}(X) = \mu$ and $\mathbb{V}(X) = \sigma^2$.

Proof
We substitute the density for X into the formula for $\mathbb{E}(X)$ and transform the integral over x into an integral over $x + \mu$:

$$\begin{aligned}
\mathbb{E}(X) &= \int_{-\infty}^{\infty} x\phi(x)dx \\
&= \int_{-\infty}^{\infty} x \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \\
&= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} x e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \\
&= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} (x+\mu) e^{-\frac{x^2}{2\sigma^2}} dx \\
&= \frac{1}{\sigma\sqrt{2\pi}} \left(\left(\int_{-\infty}^{\infty} x e^{-\frac{x^2}{2\sigma^2}} dx \right) + \mu \left(\int_{-\infty}^{\infty} e^{-\frac{x^2}{2\sigma^2}} dx \right) \right) \\
&= \frac{1}{\sigma\sqrt{2\pi}} \left(\left(\int_{-\infty}^{\infty} x e^{-\frac{x^2}{2\sigma^2}} dx \right) + \mu\sigma\sqrt{2\pi} \right) \\
&= \mu + \frac{1}{\sigma\sqrt{2\pi}} \left(\int_{-\infty}^{\infty} x e^{-\frac{x^2}{2\sigma^2}} dx \right) = \mu.
\end{aligned} \tag{100}$$

The expression after the $\mu+$ is simply 0, because the function is point-symmetric about 0 (the factor x is obviously odd, and the other factor depends only on $|x|$).

The variance is handled in exactly the same way, by simplifying this expression:

$$\begin{aligned}\mathbb{V}(X) &= \int_{-\infty}^{\infty} x^2\phi(x)dx - (\mathbb{E}(X))^2 \\ &= \int_{-\infty}^{\infty} x^2 \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx - \mu^2.\end{aligned} \tag{101}$$

After the same substitution $x \to x+\mu$ and expanding $(x+\mu)^2$, what is left is $\frac{1}{\sigma\sqrt{2\pi}} \int x^2 e^{-x^2/2\sigma^2}\, dx$ (the mixed term is again odd, and the μ^2 term cancels), and integration by parts with $u = x$, $v' = xe^{-x^2/2\sigma^2}$ yields σ^2 for it; we leave the details to you.

Definition
A random variable with $X \sim \mathcal{N}(0,1)$ is also called *standard normally distributed*

Proposition
If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $\frac{X-\mu}{\sigma} \sim \mathcal{N}(0,1)$.

Proof
Idea: It is immediately clear that $\mathbb{E}(X-\mu) = 0$ and that $\mathbb{V}\left(\frac{X-\mu}{\sigma}\right) = 1$. What is less clear is that the new random variable really is normally distributed; that can be proved with the moment generating function, for instance.

Standardisation is a special case of a more general statement that you will need all the time:

Proposition
If $X \sim \mathcal{N}(\mu, \sigma^2)$ and $a \neq 0$, $b \in \mathbb{R}$, then

$$aX + b \sim \mathcal{N}\left(a\mu + b,\ a^2\sigma^2\right). \tag{102}$$

So normality is preserved under affine-linear transformations.

Proof
Idea: Expected value and variance follow from the rules of calculation in the chapter Expected value and variance. That the shape is preserved can be read off the density: if a random variable X has density p_X, then $Y = aX + b$ has density

$$p_Y(y) = \frac{1}{|a|}\, p_X\left(\frac{y-b}{a}\right) \tag{103}$$

(the transformation formula: the factor $\frac{1}{|a|}$ makes sure the integral comes out as 1 again). Substituting the normal density, the exponent becomes $-\frac{((y-b)/a-\mu)^2}{2\sigma^2} = -\frac{(y-(a\mu+b))^2}{2a^2\sigma^2}$ and the prefactor $\frac{1}{|a|\sigma\sqrt{2\pi}}$ – exactly the density of $\mathcal{N}(a\mu+b, a^2\sigma^2)$.

Two special cases are worth remembering. With $a = \frac{1}{\sigma}$ and $b = -\frac{\mu}{\sigma}$ we get the standardisation from above. And conversely: if $Z \sim \mathcal{N}(0,1)$, then $\sigma Z + \mu \sim \mathcal{N}(\mu, \sigma^2)$ – so every normal distribution arises from the standard normal distribution by stretching and shifting. That is exactly how they are generated on a computer, too: `mu + sigma * rng.standard_normal(n)` needs only one random number generator for all normal distributions. The chapter The multivariate normal distribution takes this as its starting point: there the same construction is carried out with a matrix instead of a number.

We also use the fact that the sum of independent random variables with densities f, g again has a density, namely the *convolution* $f * g$. Rather than introduce and prove that rigorously, we merely record the result for normal distributions, which can be checked using the convolution.

Theorem
Among all **continuous** distributions of real random variables X with **support all of** $\mathbb{R}$ and fixed expected value $\mu = \mathbb{E}(X)$ and fixed variance $\sigma^2 = \mathbb{V}(X)$, the normal distribution is the one with maximal entropy.

Even to state this theorem precisely, we need a notion of entropy for continuous distributions. Rather than set that up now, we return to it later, when we also discuss relative entropy, its continuous version and likelihood.

Proposition
If $X \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $Y \sim \mathcal{N}(\mu_2, \sigma_2^2)$ are independent, then $X + Y \sim \mathcal{N}(\mu_1 + \mu_2,\ \sigma_1^2 + \sigma_2^2)$. More generally, every linear combination $aX + bY$ is again normally distributed, with expected value $a\mu_1 + b\mu_2$ and variance $a^2\sigma_1^2 + b^2\sigma_2^2$.
Without proof

Expected value and variance follow from the rules of calculation in the chapter Expected value and variance; the remarkable part is that the *shape* is preserved too – the sum is not just some distribution with these moments but a bell curve again. That is far from obvious: the sum of two uniformly distributed variables is triangularly distributed, not uniformly distributed. The normal distribution is said to be *stable under convolution*. The chapter on the multivariate normal distribution shows the generalisation: there the same holds for *dependent* X, Y too, as soon as they are jointly normally distributed, with an extra covariance term in the variance.

What matters, though, is the theorem's take-home message: if nothing is known about a continuous distribution on the real numbers beyond its expected value and variance, the corresponding normal distribution is the most sensible assumption. In this sense the normal distribution is a good continuous substitute for the discrete uniform distribution, but now on all of $\mathbb{R}$ (the normal distribution has support $\mathbb{R}$).

The restriction to support $\mathbb{R}$ is no formality here; it is the actual modelling assumption. If the support is known more precisely, a different distribution has maximal entropy – and is hence the most sensible assumption:

Support	what else is known	distribution with maximal entropy
$\{1, \dots, n\}$	nothing	discrete uniform distribution, $p(k) = \frac{1}{n}$
$[a, b]$	nothing	continuous uniform distribution, $f(x) = \frac{1}{b-a}$
$[0, \infty)$	expected value μ	exponential distribution, $f(x) = \lambda e^{-\lambda x}$ with $\lambda = \frac{1}{\mu}$
$\mathbb{R}$	expected value and variance	normal distribution
$[a, b]$	expected value and variance	truncated normal distribution (`scipy.stats.truncnorm`)

The profiles of the three continuous distributions are in the chapter The most important continuous distributions, that of the discrete uniform distribution in the chapter The most important discrete distributions. The truncated normal distribution is exactly what it sounds like: a normal distribution whose density is set to 0 outside $[a, b]$ and renormalised inside. You need it when a quantity is bounded by its very nature – proportions between 0 and 1, readings at the display limit of an instrument – and a normal model would otherwise put probability mass where no values can occur at all.

The pattern behind this is always the same: maximal entropy means "assume as little extra as possible". What you know goes in as a constraint; everything else you leave to chance.

3.2.4 The central limit theorem

Theorem
If X_i is a sequence of identically distributed, mutually independent (iid = independent identically distributed) random variables with expected value $\mathbb{E}(X_i) = 0$ and variance $\mathbb{V}(X_i) = \sigma^2$, and we consider the scaled sum $Z_N := \frac{1}{\sigma\sqrt{N}} \sum_{i=1}^{N} X_i$ (with $\mathbb{E}(Z_N) = 0$ and $\mathbb{V}(Z_N) = 1$), then we have convergence in distribution:

$$Z_N \xrightarrow{N\to\infty} \mathcal{N}(0,1) \tag{104}$$

The letter Z is a reminder that Z_N is the sum $S_N = \sum_{i=1}^{N} X_i$ we looked at above for Bernoulli variables, normalised to expected value 0 and variance 1.

Convergence in distribution means that the cumulative distribution function of Z_N converges to that of a standard normal distribution.

There are also weaker versions of this theorem that allow certain dependencies among the X_i.

Remark
The theorem matters to us because it lets us approximate a sum of independent random variables by a normal distribution. That situation arises often in practice when you take measurements on physical or techno-social systems, which are as a rule subject to a long list of largely independent disturbances. The sum of these sources of error is (given enough independent sources of error) approximately normally distributed.

Two concrete examples: the error in position fixing with GNSS systems such as GPS is normally distributed (even though the atmospheric disturbances that make the value less accurate are not). In the industrial manufacture of components, likewise, sources of error throughout the production process add up to a normal distribution in the end – one whose variance has to be kept small enough for the component to do its job. A Lego brick, for instance, must not deviate too much from an identically built Lego brick, or the structure will not hold together properly afterwards.

Remark
The standard error. The application that matters most to us: the mean $\overline{X}_N$ of a sample of size N is itself a (scaled) sum, and hence approximately normally distributed for large N, with expected value μ and standard deviation $\sigma/\sqrt{N}$. This standard deviation of the sample mean is called the **standard error**. Since a normally distributed quantity deviates from its expected value by less than 1.96 standard deviations with probability 95 %, the true expected value lies in the interval $\overline{X}_N \pm 1.96\,\sigma/\sqrt{N}$ for about 95 % of all samples. That is sharper than Chebyshev's inequality from the chapter Expected value and variance, which guarantees only 75 % for two standard errors, and it is the recipe by which the chapter Statistical tests reasons from the sample to the population. Titanic example: of 314 women, 74.2 % survived; the standard error of this proportion is $\sqrt{0.742 \cdot 0.258/314} \approx 0.025$, so the interval is about 74 ± 5 percentage points.

Aside: the speed of the convergence can be estimated too, which tells us how good the approximation by a normal distribution is:

Theorem (Berry–Esseen)
There exists a constant C with $0.41 \le C \le 0.48$ and the following property:

If X_i is a sequence of iid random variables with expected value 0 and variance σ^2 whose third absolute moments $\mathbb{E}|X_i|^3 = \rho < \infty$ also exist, and if we write F_N for the cumulative distribution function of $Z_N = \frac{1}{\sigma\sqrt{N}} \sum_{i=1}^{N} X_i$ and Φ for that of the standard normal distribution $\mathcal{N}(0,1)$, then for all N:

$$\sup_{x\in\mathbb{R}} |F_N(x) - \Phi(x)| \;\le\; \frac{C\rho}{\sigma^3\sqrt{N}}. \tag{105}$$

The exact value of C is not known; the bounds given are the current state of research. So for concretely known σ, ρ we can always find an N that makes the left-hand side as small as we like. In other words: for any desired accuracy, we can compute how many of the iid random variables have to be added up (and scaled) for Z_N to be approximated by a standard normal distribution.

Naturally, if the expected value is not 0, a variant follows as a corollary (as with the central limit theorem), with convergence to $\mathcal{N}(\mu, 1)$ – and with a different rescaling, to $\mathcal{N}(\mu, \sigma^2)$ as well. In practice it looks more like this: every unknown error term is modelled by a normal distribution whose expected value and variance are estimated empirically.

3.2.5 A practical example: The "N ≥ 30" myth

Many textbooks, statistics courses and analysis protocols offer a rule of thumb: *from $N = 30$ on you may work with a normal distribution.* The rule is so widespread that it is rarely questioned – and in that form it is wrong. And Berry–Esseen tells us why.

The bound above reads $\frac{C\rho}{\sigma^3\sqrt{N}}$. In it, N appears only under a square root; the decisive quantity is the factor ρ/σ^3, the third absolute moment relative to the cube of the standard deviation – a measure of how skewed the individual random variable is. For a symmetric distribution this factor is small; for a skewed one it can be arbitrarily large, and then $N = 30$ is simply not enough.

The most important case in practice is a binary feature: "side effect occurred yes/no", "click yes/no", "survived yes/no". Such a feature is Bernoulli distributed with parameter p, and for it we can compute the factor. For $X \sim \text{Bernoulli}(p)$, centred to have expected value 0,

$$\sigma^2 = p(1-p), \qquad \rho = \mathbb{E}\,|X-p|^3 = p(1-p)\left[(1-p)^2 + p^2\right], \tag{106}$$

hence

$$\frac{\rho}{\sigma^3} = \frac{(1-p)^2 + p^2}{\sqrt{p(1-p)}}. \tag{107}$$

For a balanced feature, $p = 0.5$, this is exactly 1, and the Berry–Esseen bound at $N = 30$ comes out (with $C \approx 0.44$) at $0.44/\sqrt{30} \approx 0.08$: the cumulative distribution functions differ by at most eight percentage points – usable. For a rare event, $p = 0.01$, by contrast, the same factor is ≈ 9.85 and the bound at $N = 30$ becomes

$$\frac{0.44 \cdot 9.85}{\sqrt{30}} \approx 0.79. \tag{108}$$

A bound of 0.79 on the deviation between two cumulative distribution functions that both lie between 0 and 1 says nothing at all – it is useless. Only at $N = 1000$ does it drop to ≈ 0.14. (The Berry–Esseen bound is an *upper* bound; the actual error may be smaller. But a rule of thumb that cannot even be justified from above is not one you should use.)

Rule of thumb

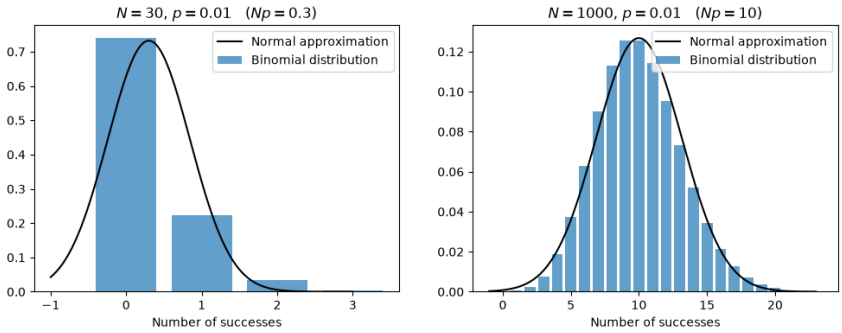
The Laplace condition. The binomial distribution $\text{Bin}(N, p)$ may be approximated by a normal distribution if

$$Np \geq 10 \qquad \text{and} \qquad N(1-p) \geq 10, \tag{109}$$

that is, if *both* outcomes occur at least ten times on average. For $p = 0.5$ that means $N \geq 20$, but for $p = 0.01$ it means $N \geq 1000$. Some textbooks write $Np(1-p) \geq 9$ instead; that comes to the same thing.

So the difference from the $N \geq 30$ rule is not academic: for an event that hits one case in a hundred, the correct threshold is 33 times higher. That is exactly the situation with rare side effects in medical studies.

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

for ax, n in ((ax1, 30), (ax2, 1000)):
    p = 0.01
    k = np.arange(0, int(n * p + 4 * np.sqrt(n * p * (1 - p))) + 2)
    ax.bar(k, binom(n, p).pmf(k), label="Binomial distribution", alpha=0.7)
    xs = np.linspace(k.min() - 1, k.max(), 200)
    ax.plot(xs, norm(loc=n * p, scale=np.sqrt(n * p * (1 - p))).pdf(xs),
            color="black", label="Normal approximation")
    np_text = f"{n * p:g}"
    ax.set_title(f"$N = {n}$, $p = 0.01$   ($Np = {np_text}$)")
    ax.set_xlabel("Number of successes")
    ax.legend()

plt.show()
```



On the left, at $N = 30$, almost all the probability mass sits on the values 0 and 1 – the distribution is cut off at zero and heavily skewed to the right, while the normal approximation runs placidly off into negative territory and spreads probability where there are no outcomes at all. On the right, at $N = 1000$, the Laplace condition is just satisfied, with $Np = 10$, and the bell shape is recognisable.

The moral is more general than the example: a rule of thumb that names only the sample size and says nothing about the distribution cannot work. What counts is the ratio of skewness to sample size.

3.3 The most important continuous distributions in SciPy

Now that we have discussed the Gaussian distribution at length, we want an overview of some of the most important continuous distributions, much as we did for the discrete ones in the chapter The most important discrete distributions. For this we use the Python module `scipy.stats`.

3.3.1 SciPy Statistics

```
import numpy as np
np.random.seed(12321)

import scipy.stats as st

# Create a random variable with an N(3,2^2) distribution:
X = st.norm(loc=3, scale=2)
# draw samples:
Xsamples = X.rvs(size=4)
print("some examples:",Xsamples)
```

```
some examples: [4.02801895 1.9033209  6.49477125 1.83362523]
```

```
# Check the distribution:
Xsamples = X.rvs(size=1000)
print("mean:", np.mean(Xsamples), "expected", X.mean()) # 3
print("std:", np.std(Xsamples), "expected", X.std()) # 2
print("3rd moment:", X.moment(3))
print("4th moment:", X.moment(4))
print("entropy:", X.entropy())
```

```
mean: 3.0004822746684243 expected 3.0
std: 1.9994507762900595 expected 2.0
3rd moment: 62.99999999999999
4th moment: 344.99999999999994
entropy: 2.112085713764618
```

```
# Fit the data:
mu, sigma = st.norm.fit(Xsamples)
print(mu, sigma)
```

```
3.0004822746684243 1.9994507762900595
```

There are further useful methods, such as `st.norm.sf`, the 'survival function', defined as 1-cdf, or `st.norm.logpdf` for when you want to work directly with the logarithm of the density function (both are potentially faster than computing it by hand).

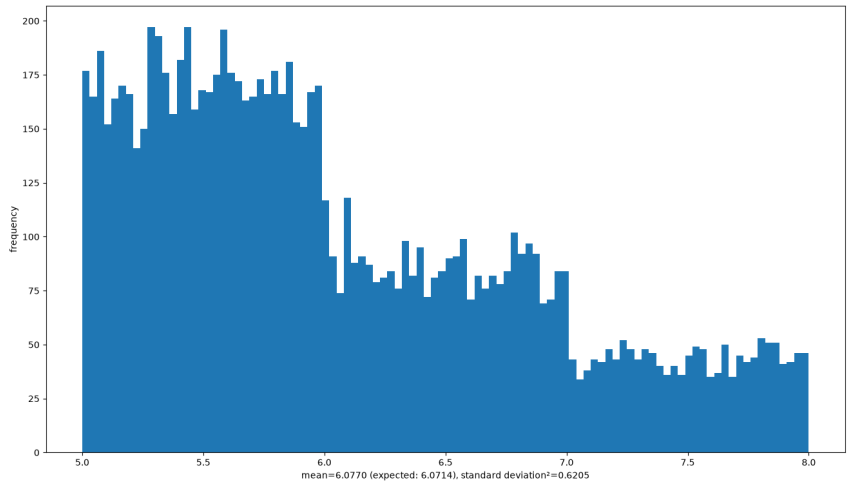
3.3.2 Matplotlib

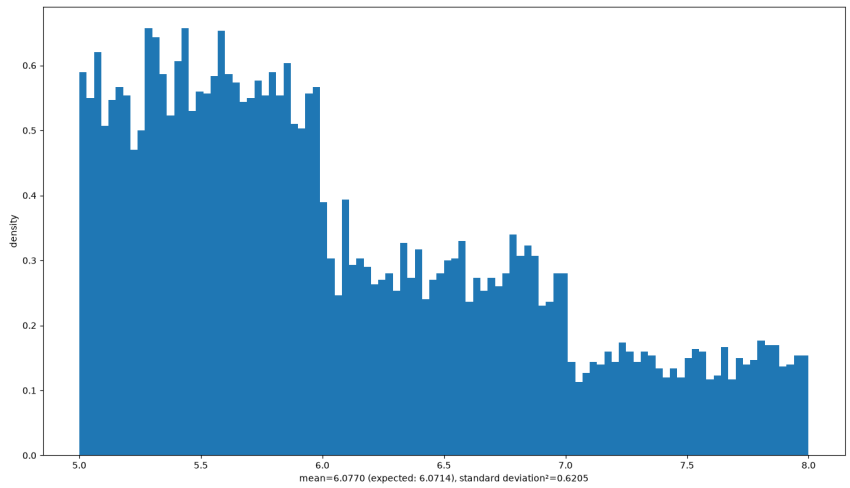
For each distribution we will look at plots of the density function (pdf) and the cumulative distribution function (cdf). For this we use the Python module `matplotlib.pyplot`, which emulates the behaviour of MATLAB. We will come back to Matplotlib in more detail.

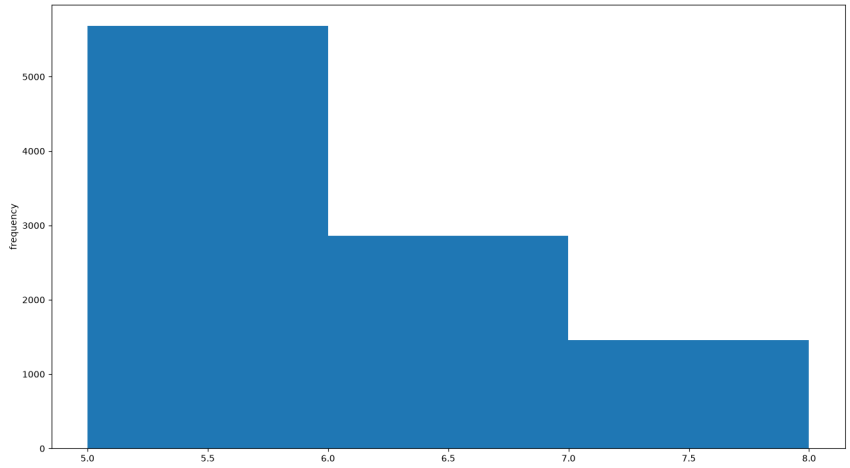
As a first example we simulate the continuous distribution on $[5, 8)$ from the chapter Continuous distributions: first one of the numbers $5, 6, 7$ is chosen with weights $\frac{4}{7}, \frac{2}{7}, \frac{1}{7}$, then a uniformly distributed number from $[0, 1)$ is added. That chapter also works out the expected value $6 + \frac{1}{14}$, which we can compare with the mean of the sample.

```
from matplotlib import pyplot as plt

# Adjust the size of the plot output:
plt.rcParams["figure.figsize"] = (16,9)

# Sample from the distribution on [5,8) from the chapter Continuous distributions:
sample = np.random.choice(3, 10000, p=[4/7,2/7,1/7]) + 5 + np.random.random(10000)
print("sample[:10] =", sample[:10])
```

```
# Plot the histogram:
plt.hist(sample, bins=100)
plt.ylabel("frequency")
plt.xlabel(f"mean={np.mean(sample):.4f} (expected: {6+1/14:.4f}), standard deviation²={np.std(
    sample)**2:.4f}")
plt.show()

plt.hist(sample, bins=100, density=True)
plt.ylabel("density")
plt.xlabel(f"mean={np.mean(sample):.4f} (expected: {6+1/14:.4f}), standard deviation²={np.std(
    sample)**2:.4f}")
plt.show()

# Histogram with few bins:
plt.hist(sample, bins=3)
plt.ylabel("frequency")
plt.show()

sample[:10] = [5.96301841 5.60942173 5.72908826 5.80068973 5.59819708 5.35377228
 6.093922   6.36189002 5.42430139 5.87334923]
```

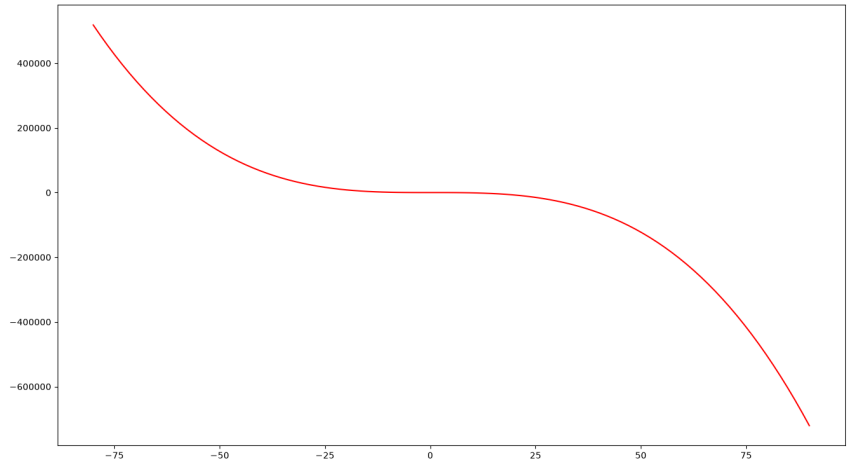






To plot a function we define it as a Python function and evaluate it at enough points; the plot is then interpolated between them:

```
# Plot the function x |-> 3+2x+x^2 -x^3 on the interval [ -80,90]:
def f(x):
    return 3 + 2*x + x**2 - x**3

x = np.linspace( -80, 90, 100)
plt.plot(x, f(x), color='red')
plt.show()
```



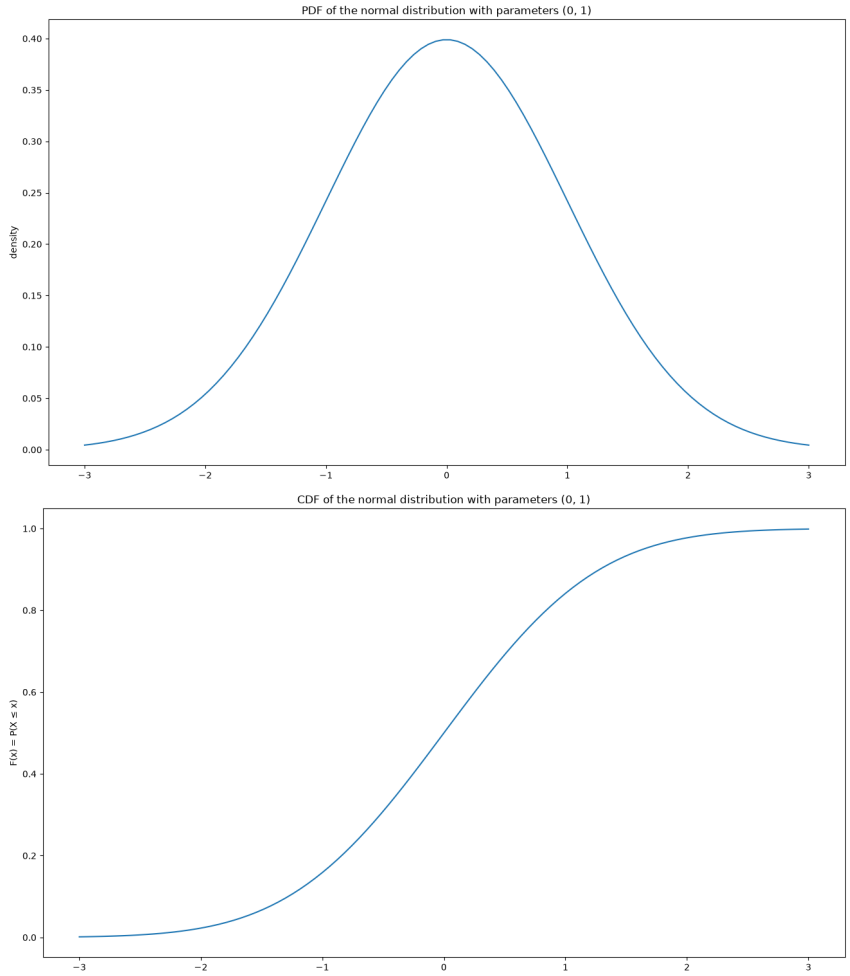
With that, we turn to the most important continuous distributions:

3.3.3 Normal distribution

- Parameters μ, σ, notation $\mathcal{N}(\mu, \sigma^2)$
- Density $\phi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
- Expected value μ
- Variance σ^2
- All odd central moments are 0 (the density is symmetric, so an odd moment $\neq 0$ is a sign of asymmetry)
- Maximum entropy for a fixed expected value, a fixed variance and support all of $\mathbb{R}$
- Central limit theorem: for iid X_i with expected value μ and variance σ^2, the standardised sum $Z_N = \frac{1}{\sigma\sqrt{N}} \sum_{i=1}^{N} (X_i - \mu)$ is approximately $\mathcal{N}(0,1)$-distributed for large N, equivalently $\frac{1}{N} \sum_{i=1}^{N} X_i \approx \mathcal{N}\left(\mu, \frac{\sigma^2}{N}\right)$ (chapter The normal distribution)

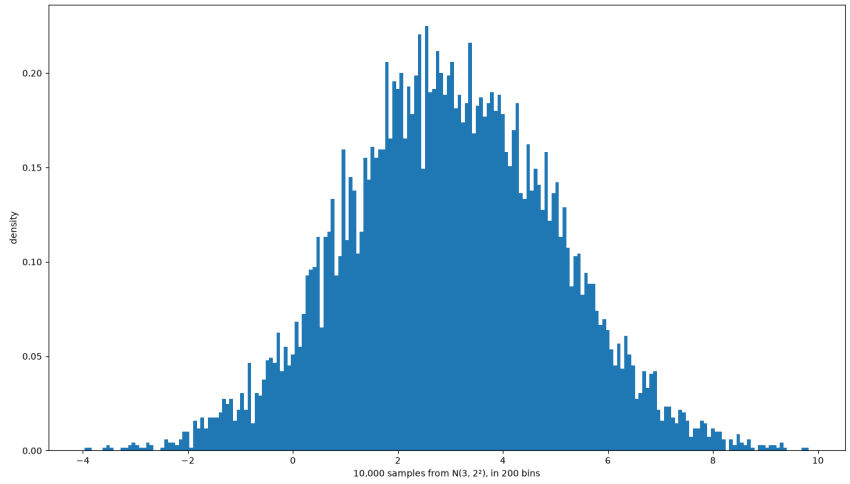
- Example applications:
  - Measurement errors in physical systems
  - Manufacturing precision
  - Biological quantities (body height etc.)

```
# For the plots:
def plotX(x, X, name, params):
    # First the density function:
    plt.plot(x, X.pdf(x))
    plt.ylabel('density')
    plt.title(f"PDF of the {name} with parameters {params}")
    plt.show()
    # Then the cumulative distribution function:
    plt.plot(x, X.cdf(x))
    plt.ylabel('F(x) = P(X \leq x)')
    plt.title(f"CDF of the {name} with parameters {params}")
    plt.show()

x = np.linspace( -3,3,100)
mu, sigma = 0, 1
X = st.norm(loc=mu, scale=sigma)
name = "normal distribution"
plotX(x, X, name, (mu, sigma))
```




```
X = st.norm(loc=3, scale=2)
Xsamples = X.rvs(size=10000)
plt.hist(Xsamples, bins=200, density=True)
plt.xlabel("10,000 samples from N(3, 2²), in 200 bins")
```

```
plt.ylabel("density")
plt.show()
```

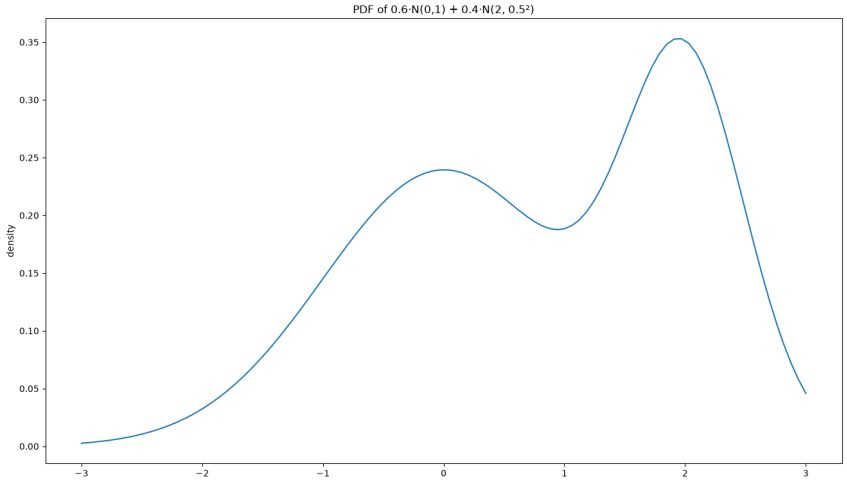


3.3.4 Contaminated normal distribution

The contaminated normal distribution is a simple case of a *mixture model*, more precisely a *Gaussian mixture*.

- Parameters $\varepsilon, \mu_1, \sigma_1, \mu_2, \sigma_2$, notation $\varepsilon\mathcal{N}(\mu_1, \sigma_1^2) + (1-\varepsilon)\mathcal{N}(\mu_2, \sigma_2^2)$
- Density $\phi(x) = \frac{\varepsilon}{\sigma_1\sqrt{2\pi}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + \frac{1-\varepsilon}{\sigma_2\sqrt{2\pi}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}}$
- Expected value $\varepsilon\mu_1 + (1-\varepsilon)\mu_2$
- Variance $\varepsilon\sigma_1^2 + (1-\varepsilon)\sigma_2^2 + \varepsilon(1-\varepsilon)(\mu_1-\mu_2)^2$
- Example applications: a two-stage process in which the outcome of a Bernoulli experiment decides which normal distribution is sampled from. One concrete experimental setup would be measuring the height of the participants in a computer science lecture. Gender is (approximately!) Bernoulli distributed (presumably not uniformly), and within each gender group height is then normally distributed (but with different means). The example also shows that every act of modelling has the potential to render invisible a part of reality that you may not want invisible at all. Fortunately, mixture models can also be set up with more than one parameter ε.

```
name = "0.6·N(0,1) + 0.4·N(2, 0.5²)"
def f(x, epsilon=0.6, mu1=0, sigma1=1, mu2=2, sigma2=0.5):
    norm1 = st.norm(loc=mu1, scale=sigma1)
    norm2 = st.norm(loc=mu2, scale=sigma2)
    return epsilon*norm1.pdf(x) + (1 -epsilon)*norm2.pdf(x)

x = np.linspace( -3,3,100)
plt.plot(x, f(x))
plt.ylabel('density')
plt.title(f"PDF of {name}")
plt.show()
```

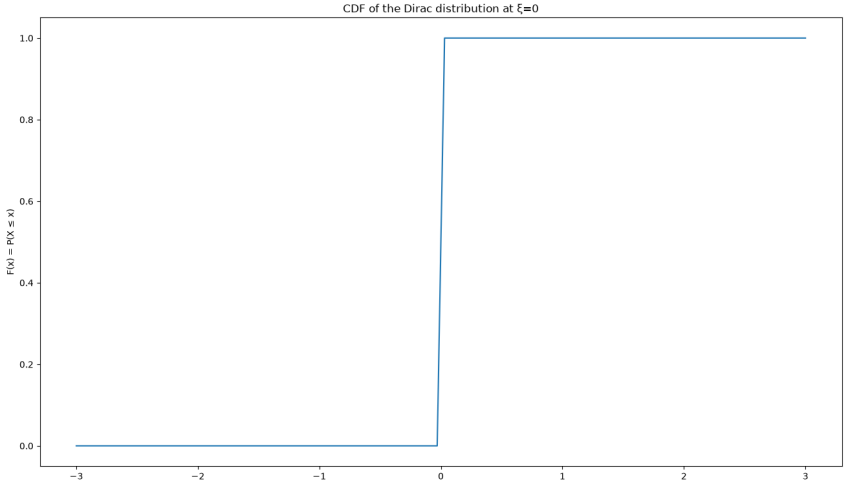


3.3.5 Dirac distribution

- Parameter ξ, notation $\delta(\xi)$
- No density, but $P(X = \xi) = 1$
- Expected value ξ
- Variance 0
- Entropy 0 (read as a discrete distribution; it has no differential entropy)
- Example applications:
  - Writing down 'density functions' in the continuous sense for discrete distributions
  - Photographs of the night sky (taken with Hubble, say) – every star is a smeared-out heap of pixels (normally distributed), but the 'actual' distribution of the stars behind it is a sum of Dirac measures, since every star sits at exactly one point (the star's true extent is irrelevant at that distance). Some super-resolution techniques work with this model.

Strictly speaking the Dirac distribution cannot be plotted (there is no density, after all). Some people draw an arrow of length 1 at ξ by way of illustration. The cumulative distribution function is a Heaviside step function:

```
name = "Dirac distribution at \xi=0"
plt.plot(x, np.heaviside(x, 1))
plt.ylabel('F(x) = P(X \leq x)')
plt.title("CDF of the "+name)
plt.show()
```

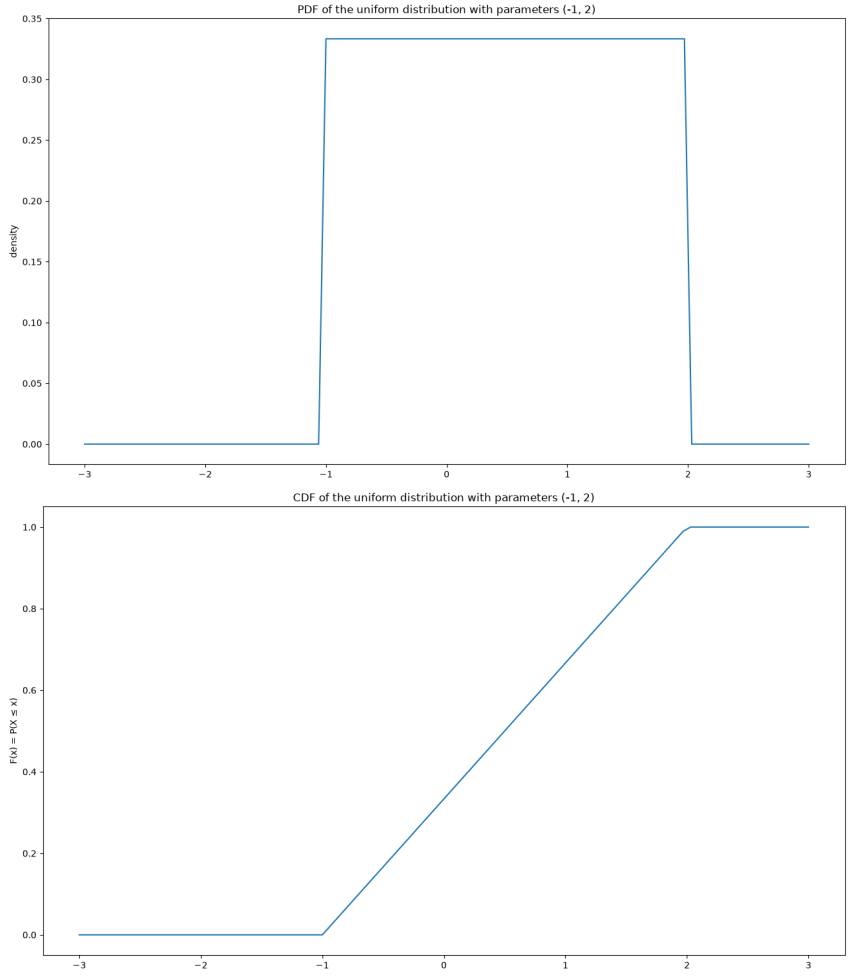


3.3.6 Continuous uniform distribution

- Parameters a, b, notation $U([a, b])$
- Density $\frac{1}{b-a}$
- Expected value $\frac{a+b}{2}$
- Variance $\frac{(a-b)^2}{12}$
- Maximum entropy among all distributions with support $[a, b]$
- Example applications:
  - If a bus arrives at your stop extremely punctually every hour but you do not know when the last one left, your waiting time at the stop, in minutes, is uniformly distributed on $[0, 60]$.
  - The temperature of your CPU from the third decimal place on, `(T * 100 - int(T * 100))`, is approximately uniformly distributed.

```
a, b = -1, 2
X = st.uniform(loc=a, scale=b -a)
print(X.mean(), X.var())
name = "uniform distribution"
plotX(x, X, name, (a, b))
```

```
0.5 0.75
```

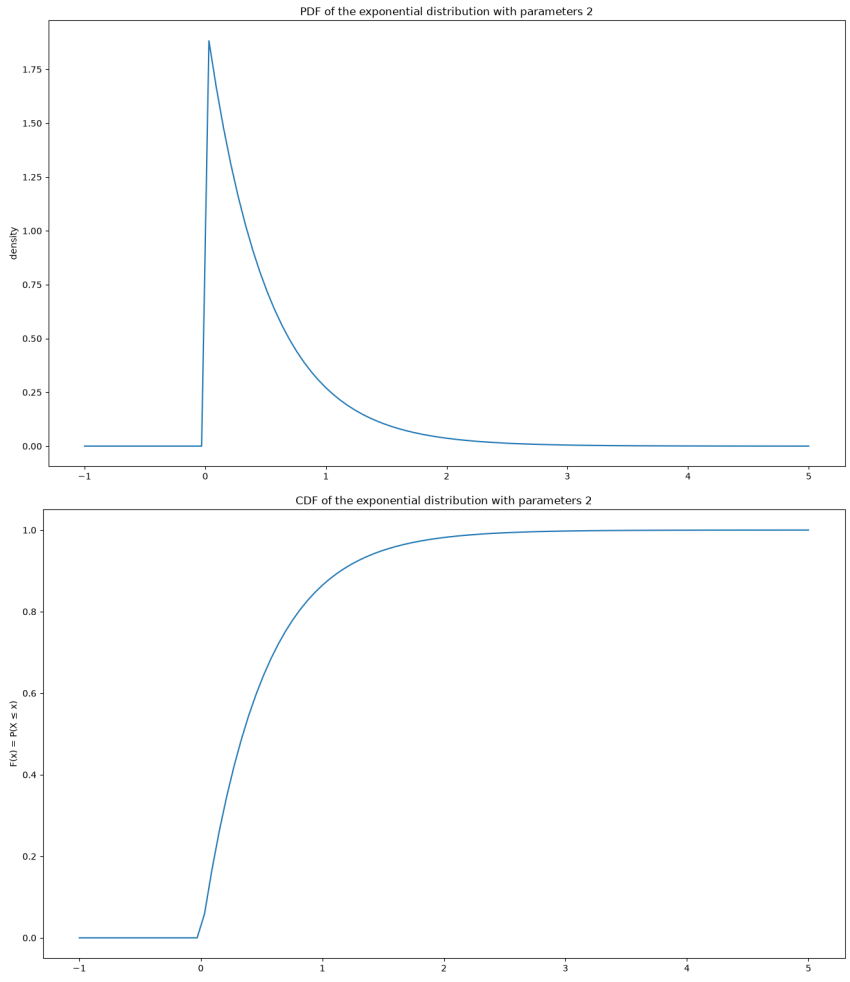


3.3.7 Exponential distribution

- Parameter λ, notation $Exp(\lambda)$
- Density $\lambda e^{-\lambda x}$ for $x \geq 0$ (and 0 for $x < 0$)
- Expected value $\frac{1}{\lambda}$
- Variance $\frac{1}{\lambda^2}$
- Maximum entropy among all distributions with support $[0, \infty)$ and expected value $\frac{1}{\lambda}$
- Example applications:
  - How long does it take until the next packet arrives at the server?

The exponential distribution is the continuous flip side of the Poisson distribution from the chapter The most important discrete distributions: count the events per unit of time and there are Pois(λ) many of them; ask instead for the waiting time until the next event, and it is Exp(λ)-distributed. In both cases λ is the same rate.

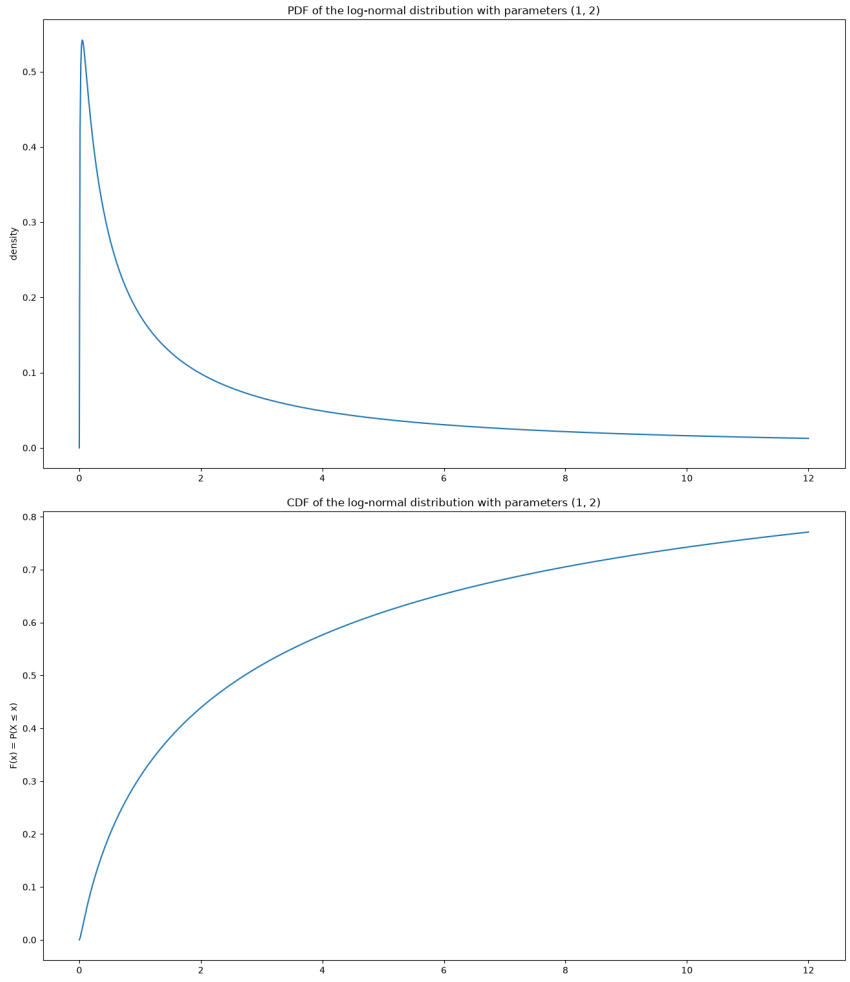
```
name = "exponential distribution"
lamda = 2   # 'lambda' is a keyword in Python (anonymous functions), hence the spelling
X = st.expon(scale = 1/lamda)
x = np.linspace( -1,5,100)
plotX(x,X,name,(lamda))
```



3.3.8 Log-normal distribution

- Parameters μ, σ, notation $Lognormal(\mu, \sigma^2)$
- Density $\frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\log(x)-\mu)^2}{2\sigma^2}}$
- Expected value $e^{\mu+\frac{\sigma^2}{2}}$, but $\mathbb{E}(\log X) = \mu$
- Variance $(e^{\sigma^2}-1)e^{2\mu+\sigma^2}$, but $\mathbb{V}(\log X) = \sigma^2$
- Maximum entropy among all distributions with support $(0, \infty)$ and $\mathbb{E}(\log X) = \mu$ and $\mathbb{V}(\log X) = \sigma^2$
- Example applications:
  - The length of a game of chess is log-normally distributed
  - Many (!) more examples are listed in Wikipedia on the log-normal distribution

```
name = "log -normal distribution"
mu, sigma = 1, 2
X = st.lognorm(s=sigma, scale = np.exp(mu))
x = np.linspace(0, 12, 1000)
plotX(x,X,name,(mu, sigma))
```

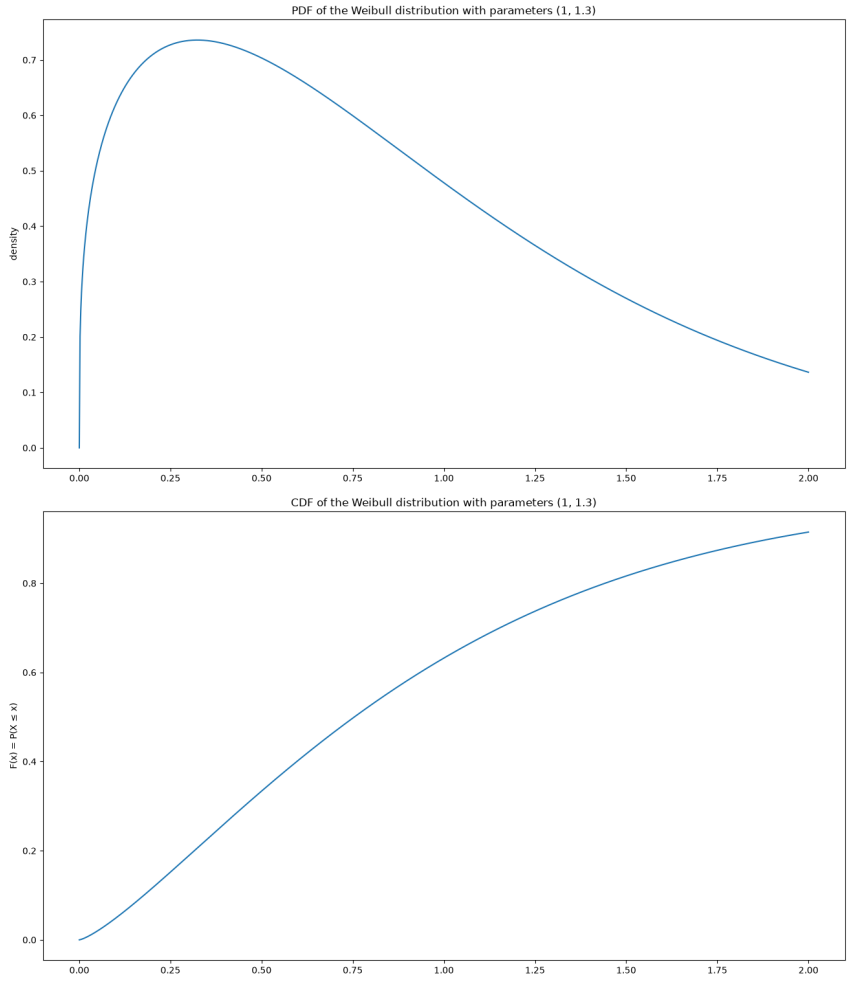


3.3.9 Weibull distribution

- Parameters $\lambda > 0, k > 0$, notation $Weibull(\lambda, k)$
- Density $f(x) = \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-\left(\frac{x}{\lambda}\right)^k}$ for $x \geq 0$ and $f(x) = 0$ for $x < 0$

- Expected value $\lambda\Gamma(1+\frac{1}{k})$ (where $\Gamma(n+1)=n!$)
- Variance $\lambda^2\left[\Gamma(1+\frac{2}{k})-\Gamma(1+\frac{1}{k})^2\right]$
- Maximum entropy among all distributions with support $[0,\infty)$, fixed $\mathbb{E}(X^k)$ and fixed $\mathbb{E}(\log X)$ – so the shape parameter k says which moment is held fixed
- Example applications:
  - How long does a heat pump run before it breaks down?
  - What is the wind speed?
  - What is the largest amount of precipitation to fall on a single day in the year?

```
name = "Weibull distribution"
lamda, k = 1, 1.3   # lamda: see above
X = st.weibull_min(c=k, loc=0, scale = lamda)
x = np.linspace(0,2,1000)
plotX(x,X,name,(lamda, k))
```



3.3.10 Data to plot and fit

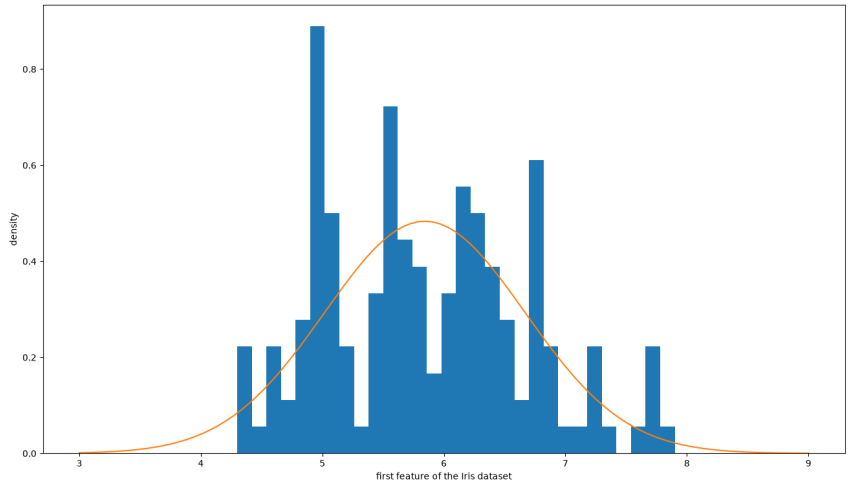
To get a little practice in plotting data and fitting distributions to it (and then visualising the fit in a plot), we need some toy data.

A very simple, and therefore very instructive, dataset is Fisher and Anderson's 'iris flower dataset', which we can get from scikit-learn, for instance:

```
from sklearn import datasets
iris = datasets.load_iris()
print(iris["data"][:5])
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
```

```
first_feature = iris["data"][:,0]
plt.hist(first_feature, bins=30, density=True)
# Now the best -fitting normal distribution on top:
mu, sigma = st.norm.fit(first_feature)
X = st.norm(loc=mu, scale=sigma)
x = np.linspace(3,9,100)
plt.plot(x, X.pdf(x))
plt.xlabel("first feature of the Iris dataset")
plt.ylabel("density")
plt.show()
```



One source of reasonably well-maintained and much-discussed toy data is Kaggle. To work with these data you usually have to turn CSV files into Numpy arrays. That is particularly painless with Pandas (which builds on Numpy), which we will come to; for now, Numpy's built-in `np.genfromtxt` will do.

In its Climate Data Center (in German), the German weather service DWD keeps a large collection of well-documented weather and climate data, some of it updated daily. It holds spatially distributed variables and time series of all manner of physical quantities and statistical distributions. One interesting (and answerable) question, for instance, is whether heavy rainfall events in Düsseldorf have become more frequent over the last few decades.

Have a look at these (and other!) data, work out how the variables might be distributed, plot the distributions and fit them. Comparing the histogram with the density function of the fitted distribution is the best way to see how close we are.

Since a great deal of data fits none of the distributions listed here, we need more complex models as well. Mostly, though, these are assembled in stages from the simple models, as with the contaminated normal distribution. One of the most important will later be the mixture of multivariate normal distributions, which lets us look at some of the principles of machine learning.

If you want to dig further into the relationships between the distributions – and have noticed that several of them can perfectly well be parametrised in different ways (the normal distribution with μ, σ, say, but also with μ, σ^2) – then ProbOnto may be just the thing. It is an ontology and knowledge base originally developed for pharmacology. The overview diagram alone is already a good memory aid:

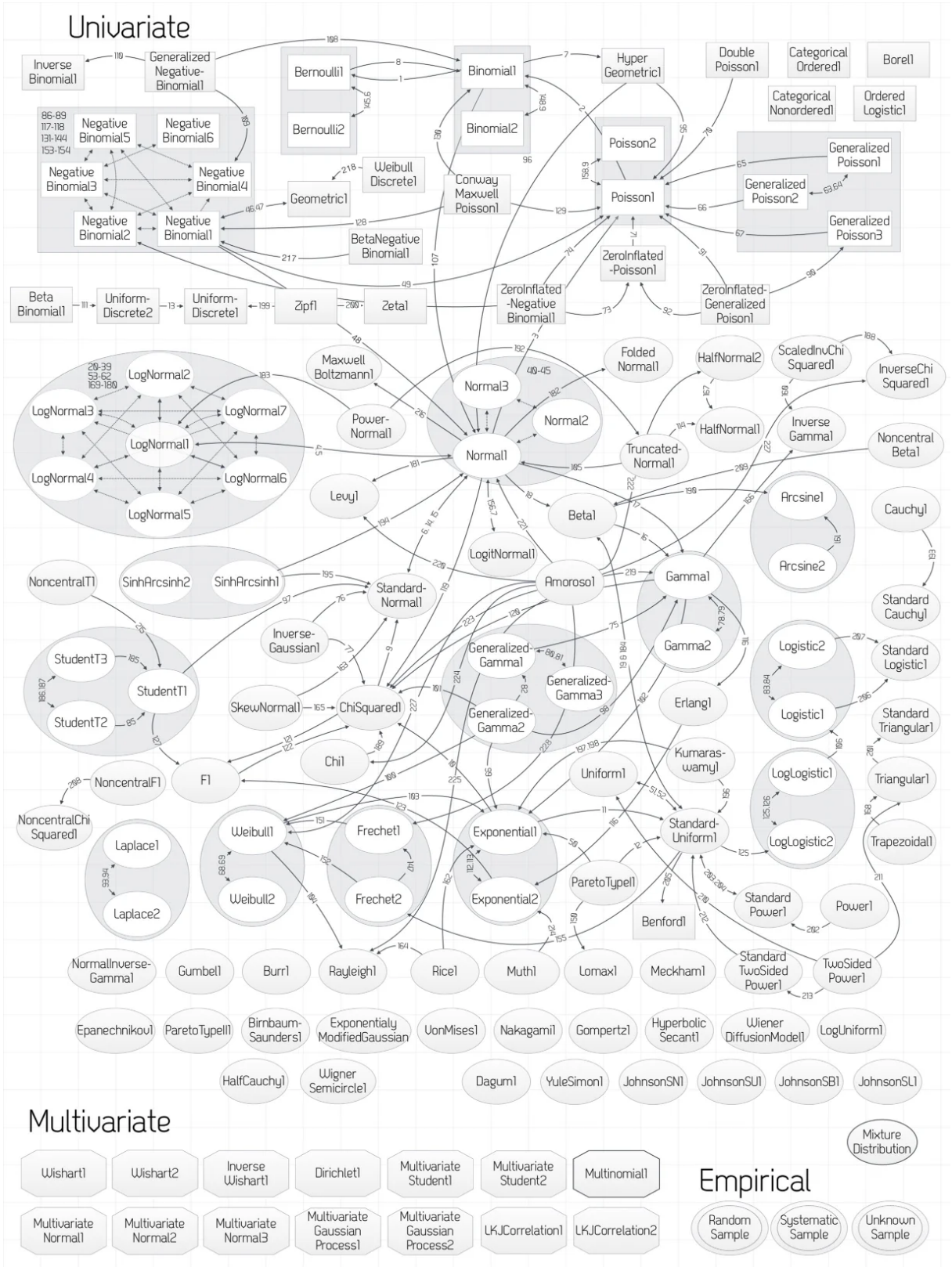


Figure 11: ProbOnto 2.5 CC-BY-SA Wikipedia user Fuzzyrandom

4 Exploratory data analysis

4.1 Descriptive statistics and visualisation

Schutt and O'Neil [2013], p. 41 shows schematically how we go about 'data science':

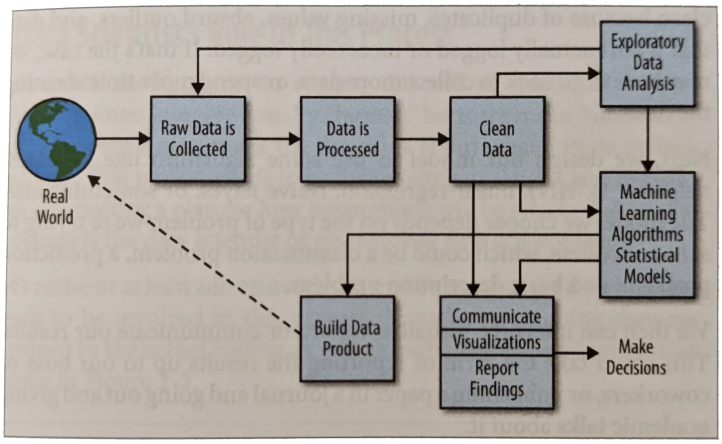


Figure 12: The "data science process". Figure after Schutt and O'Neil [2013], p. 41 (Cathy O'Neil and Rachel Schutt, O'Reilly 2013).

Our focus here: exploratory data analysis

We will go over what we have learned about descriptive statistics so far and put it in a new context – summary statistics and visualisation in particular.

This revision doubles as a vocabulary lesson: we want to understand the language of statistical data analysis and the language of machine learning alike, and to be able to translate between them.

4.1.1 Statistics vs. machine learning

There are two "cultures": statistics with statistical learning, and machine learning with predictive modelling. Machine learning is a subfield of artificial intelligence, statistics a subfield of mathematics.

- Corresponding terms, statistics – machine learning
  - model – model (or, where neural networks are involved, the net)
  - parameters – weights
  - fitting – learning
  - performance on the test set – generalisation
  - regression, classification – supervised learning
  - density estimation, clustering – unsupervised learning
  - error function (loss function) – loss
  - estimating the parameters – training
  - sample – training data (training set); one pass through all the training data is called an epoch

The right-hand column is the vocabulary of the course "Introduction to Deep Learning", which uses it throughout; anyone who knows the left-hand column will quickly find their way around there.

Exploratory data analysis means computing summary statistics and visualisations (descriptive statistics, that is) in order to get a better sense of the information in a dataset. It is a human-driven, iterative process, and it needs knowledge of the field of application – without some idea of which values are plausible and which questions are worth asking, you produce diagrams instead of insights. *Data mining* is about automating this step of a data science project: the automatic discovery of patterns and relationships, often following a scheme fixed in advance.

The cultures of modelling can be divided more finely still, as C. Molnar does in the book "Modeling Mindsets: The Many Cultures Of Learning From Data" Molnar [2022]; to follow that, though, it helps to already know the maximum likelihood approach and Bayes' theorem.

4.1.2 Descriptive statistics

Descriptive statistics covers the data analysis that happens *before* inferential statistics. Mostly that means describing the data through various summary measures.

- A sample has a size = length = number of elements / rows / instances / outcomes (in Numpy that is the length of axis 0, see Numpy)

  Distinct from the sample is the *population*: the totality of all possible outcomes, with some distribution.

- With tabular data, the columns are also called variables, attributes, characteristics or features, and the possible values are called *levels*. A dataset with n rows and k columns therefore has n samples and k features. Not every column is a feature in the statistical sense, though: a serial number, a student number or a primary key from a database only identifies the row and carries no information about what was measured – any analysis that treats an ID as a feature will find, at best, chance in it and, at worst, the order in which the data was collected. The *target variable* you want to predict is a column too, but it does not count when you count the features. Depending on the kind of feature, several *levels of measurement* (also called scale types) are distinguished; a feature is said to be measured on a nominal, an ordinal or a cardinal scale:

  - Nominal scale: categorical, discrete, no fixed order (e.g. sex, postcode, religious affiliation). Careful: mean and median make no sense here, but frequencies do. The postcode is the classic case of "looks like a number, but is nominal" – a `mean()` over a column of postcodes gives you a number that means nothing.
  - Ordinal scale: values can be ordered, but with no measure of distance (e.g. rankings in sport, military ranks, the Mohs hardness of minerals). Careful: here too a mean makes no sense – Mohs hardness 5 does not lie "halfway" between 4 and 6; it is merely harder than 4 and softer than 6.
  - Interval scale (a cardinal scale): values (discrete or continuous) can be ordered and the distances between them stated, but the zero point is arbitrary (e.g. temperature in degrees Celsius, dates, the water level of the Rhine, Likert scales such as "does not apply at all …applies fully"). Differences are meaningful, ratios are not: 20 °C is not "twice as warm" as 10 °C.
  - Ratio scale: a cardinal scale with a natural zero point, so that the ratio of two values is a meaningful quantity (e.g. age, income, temperature in kelvin, speed in km/h, the number of people in a household). Here, for three values a, b, c, the ratios $a : b$ and $a : c$ can be compared with each other.

  Which summary measures, and which arithmetic, make sense for a feature depends on its level of measurement – see the summary below, in the section Level of measurement and permissible summary measures.

- Univariate descriptions

  are summary measures that refer to a single variable and not to how the variables relate to one another. They can be divided further:

  - central tendency (what used to be called 'averages')
    - The arithmetic mean (our estimate of the expected value of the underlying distribution)
    - The median (half the data points lie above the median, and in that sense it is the middle) Incidentally, it is always the case that $|\texttt{median}(x) - \texttt{mean}(x)| \leq \sigma$ On an ordinal scale (that is, when distances between the values cannot be stated) the median is a meaningful quantity
    - The mid-range is the arithmetic mean of the maximum and the minimum.
    - The geometric mean
      is $\sqrt[n]{x_1 \cdots x_n}$, the n-th root of the product of the values (in Numpy `np.exp(np.mean(np.log(x)))`, or directly as `scipy.stats.gmean` in Scipy). It is the right mean when values compound multiplicatively – for growth rates, say, the geometric mean of the yearly factors is the constant factor that would lead to the same result after n years. It presupposes

a ratio scale, because it needs a natural zero point: the geometric mean of 10 °C and 20 °C is $\sqrt{200} \approx 14.14$ °C, but convert the same two temperatures to kelvin and you get $\sqrt{283.15 \cdot 293.15} \approx 288.11$ K ≈ 14.96 °C – two different answers to the same question. For negative values it is not defined at all.

* The mode
  is defined as the most frequent value. The term on its own is not very useful, but it is often used in the sense that a distribution is unimodal or bimodal. Unimodal means that there is one clearly recognisable mode which describes the distribution well. Bimodal means that the distribution is a mixture distribution whose two components have different modes, which you then see as two peaks. In general it is not clear-cut whether a dataset should be regarded as unimodal, bimodal or multimodal.
  For variables on a nominal scale the mode is a meaningful measure of location
* Several of the measures of central tendency come, in a sense, with a measure of dispersion of their own (see Dispersion), a fact you can remember as "dispersion precedes location". The Wikipedia article 'Central tendency' is recommended on this.

– Dispersion, spread

  Measures of spread characterise how far the data scatters around a measure of location – that is, how strongly it deviates from it. Since this amounts to characterising how far the individual data points deviate from one another, measures of spread can also be defined without direct reference to a measure of location.

  * Range: the largest value minus the smallest
  * Variation ratio (also the *non-modal proportion*): the fraction of the sample whose value is not the mode
  * Quartiles: by analogy with the median, the lower quartile $Q1$ is the value below which exactly a quarter of the data lies, and the upper quartile $Q3$ the value above which exactly a quarter lies. The median is $Q2$. Percentiles are defined in the same way. The difference $Q3-Q1$ is the **interquartile range** (IQR); it is the measure of spread that belongs to the median, and unlike the range it is insensitive to outliers – which is exactly why a boxplot draws its box from $Q1$ to $Q3$.
  * Sample variance: mean squared error about the mean
  * Sample standard deviation
    Once we calculate with units, a problem with the variance shows up: if a random variable X or a vector x (a column of a table, say) is measured in metres (a ratio scale), then computing the variance means summing, or integrating, over squares, so the variance is in square metres. This also shows that the variance is scale-dependent in general: comparing two variables, one in metres and the other in centimetres, by their variances is no comparison at all. Taking the square root fixes the first problem and yields, in the standard deviation, a measure in the same unit as the expected value.
    Careful: two formulas go by the name "sample standard deviation", one with n in the denominator and one with $n-1$. This is also known as the *Bessel correction*, and we will come back to it as part of estimation theory. In these notes the n-version is meant unless stated otherwise (which is also how `numpy.std` calculates); pandas (`DataFrame.std`) and `scipy.stats`, by contrast, divide by $n-1$ by default (`ddof=1`).
  * Coefficient of variation
    The standard deviation has the same unit as the data, but that still does not help when comparing two different features: is a spread of 3 hours a lot? The **coefficient of variation** $\frac{\sigma}{\mu}$ is unitless and answers the question as a fraction of the mean (in Numpy `np.std(x)/np.mean(x)`, in Scipy `scipy.stats.variation`). It only makes sense on a ratio scale, because it depends on the zero point: computed in Celsius it can come out negative or infinite, in kelvin it cannot. Its reciprocal $\frac{\mu}{\sigma}$ is the **signal-to-noise ratio**, used in exactly the same way in engineering – the larger it is, the more clearly the signal stands out from the noise.
  * dispersion precedes location: every measure of spread has a measure of location that belongs to it, namely the value that minimises that measure of spread – which one depends on the norm you use to measure the error:
    · The mode minimises the variation ratio (L_0)

        - The median minimises the mean absolute error (L_1)
        - The mean minimises the mean squared error (L_2)
        - The mid-range minimises the maximum absolute deviation (L_∞)

        The two middle statements are proved in the chapter Expected value and variance, section "Quantifying deviation".

    - Shape
        * Skewness is the third central moment, normalised by the standard deviation (μ_3/σ^3)
        * Kurtosis is the fourth central moment, normalised by the standard deviation (μ_4/σ^4)
        * Excess kurtosis is the difference between the kurtosis and the kurtosis of a normal distribution – which is exactly 3, whatever the parameters

- Multivariate descriptions:

    - Correlations

      We will have to deal with covariance and correlation soon. Covariance is to variance as an inner product is to a norm. The (Pearson) correlation coefficient is normalised so as to be scale-invariant.

For a short and relaxed overview of part of the question "what actually are data", and of the question of scales in particular, there is a video from the HeiCAD programme "KI für Alle" (in German).

Level of measurement and permissible summary measures Not every summary measure above makes sense for every feature. A mean over postcodes is a number, but it is not information; a standard deviation over school grades presupposes that the distance from "good" to "very good" is the same as the distance from "sufficient" to "deficient". The level of measurement decides what is allowed:

Level of measurement	Example	Meaningful comparisons	Measure of location	Measure of spread
nominal	degree programme, colour, postcode	equal / not equal	mode	variation ratio (non-modal proportion)
ordinal	school grade, Mohs hardness, rank	additionally: smaller / larger	median, quartiles	range, interquartile range
interval	temperature in °C, date, Likert scale	additionally: differences	arithmetic mean, mid-range	variance, standard deviation
ratio	age, price, temperature in K	additionally: quotients	additionally: geometric mean	additionally: coefficient of variation

Every level of measurement also permits all the summary measures of the rows above it: for a cardinal feature you may report the median and the mode as well, for an ordinal one the mode. The other way round does not work – the mean and variance of a nominal variable are meaningless, even when its levels are stored as numbers. So a column of data type `int` is a long way from being on a cardinal scale: an ID is nominal, a school grade ordinal.

Two nominal features. There is no correlation coefficient for the relationship between two nominal features, but there is an equally simple summary measure: the **cross-tabulation** (contingency table), which counts how often each combination of levels occurs. A cross-tabulation of ground truth against prediction is called a confusion matrix; the chapter Titanic EDA reads the quality of a classifier off one, and pandas produces it with `pd.crosstab`.

Encoding categorical features. Most machine learning methods (linear regression, say) calculate with numbers, and how to translate a nominal or an ordinal feature into numbers depends on the same table:

- An **ordinal** feature may be encoded as integers along its order ("low" = 1, "medium" = 2, "high" = 3; *integer* or *ordinal encoding*). The order is real; only the equal spacing is a modelling assumption, and one to keep an eye on.

- A **nominal** feature may *not* be: encode degree programmes as $1, 2, 3$ and you invent an order and distances that do not exist, and a linear model would calculate with them. Instead you give every level a column of its own, which is 1 when that level is present and 0 otherwise (*one-hot encoding*; the columns are also called *dummy variables*). A single column "degree programme" with three levels thus becomes three columns of zeros and ones.

The chapter Linear regression shows how to do this with pandas, and what it means for a regression, on the Melbourne dataset. How to unify different spellings of a category beforehand is covered in the chapter Data cleaning.

A worked example Take the $n = 8$ values $x = (7,\ 2,\ 9,\ 4,\ 4,\ 12,\ 5,\ 3)$ – the number of emails per day over a week plus a day, say.

- **Sorting**: $(2,\ 3,\ 4,\ 4,\ 5,\ 7,\ 9,\ 12)$.
- **Mode**: 4, the only value that occurs more than once.
- **Median**: n is even, so we take the mean of the two middle values 4 and 5: $\texttt{median}(x) = 4.5$.
- **Quartiles**: the lower half $(2, 3, 4, 4)$ has median $Q_1 = 3.5$, the upper half $(5, 7, 9, 12)$ has median $Q_3 = 8$; the interquartile range is $Q_3 - Q_1 = 4.5$. Careful: there are several conventions for quartiles. `numpy.percentile` interpolates linearly between the sorted values by default and gives $Q_1 = 3.75$ and $Q_3 = 7.5$ here.
- **Range**: $12 - 2 = 10$. **Mid-range**: $(12 + 2)/2 = 7$.
- **Mean**: $\texttt{mean}(x) = \frac{7+2+9+4+4+12+5+3}{8} = \frac{46}{8} = 5.75$.
- **Variance** (n-version): the deviations from the mean are $(1.25,\ -3.75,\ 3.25,\ -1.75,\ -1.75,\ 6.25,\ -0.75,\ -2.75)$; their squares $(1.5625,\ 14.0625,\ 10.5625,\ 3.0625,\ 3.0625,\ 39.0625,\ 0.5625,\ 7.5625)$ add up to 79.5, so $\sigma^2 = 79.5/8 = 9.9375$.
- **Standard deviation**: $\sigma = \sqrt{9.9375} \approx 3.15$. With the Bessel correction: $\sqrt{79.5/7} \approx 3.37$.
- **Coefficient of variation**: $\sigma/\texttt{mean}(x) = 3.15/5.75 \approx 0.55$ – the spread amounts to a good half of the mean, so the days are very uneven. **Geometric mean**: $\sqrt[8]{7 \cdot 2 \cdot 9 \cdot 4 \cdot 4 \cdot 12 \cdot 5 \cdot 3} \approx 4.95$, smaller than the arithmetic mean 5.75 (which is no accident: the geometric mean is never larger than the arithmetic one).
- Check: $|\texttt{median}(x) - \texttt{mean}(x)| = 1.25 \leq \sigma$. The mean lies above the median because the outlier 12 pulls it up: the distribution is skewed to the right.

The same in Numpy (with the quartiles as `numpy.percentile` interpolates them):

```
import numpy as np
x = np.array([7, 2, 9, 4, 4, 12, 5, 3])
print(np.median(x))            # 4.5
print(np.percentile(x, [25, 75]))  # [3.75 7.5 ]
print(np.mean(x), np.var(x))   # 5.75 9.9375
print(np.std(x))               # 3.1524... (n -version)
print(np.std(x, ddof=1))       # 3.3701... (Bessel correction, n -1)
print(np.std(x) / np.mean(x))  # 0.5482... coefficient of variation
print(np.exp(np.mean(np.log(x))))  # 4.9542... geometric mean
```

4.1.3 Simpson's paradox

Before we come to the pictures, a warning about aggregation that needs nothing more than percentages and still shows why data analysis is more than arithmetic. It comes from a medical study of two treatments for kidney stones Charig et al. [1986]; we will simply call them A and B. Each was given to 350 patients, and "success" is whatever the clinicians defined it to be.

Treatment	Patients	Successes	Success rate
A	350	273	78%
B	350	289	83%

Imagine you have kidney stones. Which treatment do you choose? Most people say B.

But there is no such thing as simply "kidney stones": there are small ones and large ones. Let us split the table along that line:

Treatment	small stones	large stones
A	81 of 87, so 93%	192 of 263, so 73%
B	234 of 270, so 87%	55 of 80, so 69%

For small stones A is better, for large stones A is better. If you have small stones you want A; if you have large stones you want A. And only if you do not know how large your stones are would you want B? That cannot be right, because your stones are either small or large.

The resolution lies in the absolute numbers, not the percentages. Treatment A was used almost exclusively on large stones (263 of 350), treatment B almost exclusively on small ones (270 of 350). Large stones are harder to treat, so the large group of difficult cases drags A's overall rate down, while B profits from its large group of easy ones. The doctors chose the therapy by stone size, and stone size determines success. A variable like this, one that influences both which group you end up in and the outcome, is called a **confounder**. The punchline: you should choose B only if it also lets you choose to have small stones.

This phenomenon is called **Simpson's paradox**: a relationship that holds overall can reverse in every single subgroup. It occurs not only in tables of percentages but just as readily with continuous features:

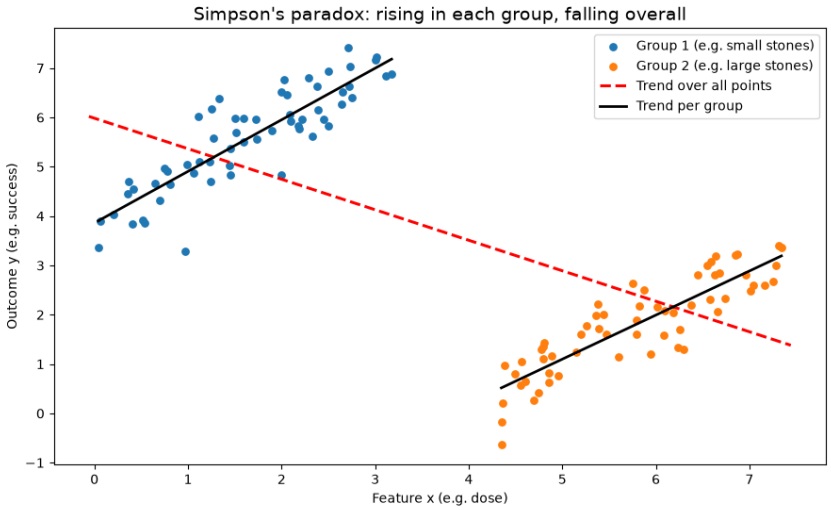


Figure 13: Simpson's paradox with continuous features: within each group y rises with x, over all the points together the trend falls.

In the kidney stone example we knew the stone size. Often, though, the variable that creates the groups is not in the table at all: then all we see is the cloud of points on the right, without the colours, and the falling trend looks entirely convincing. Such an unobserved variable is called a **latent variable** – nothing mysterious about the word, it just means "not observed". The overall distribution is then a **mixture** of two group distributions, and the question of whether a cloud of points can be decomposed into such groups will meet us throughout this book: with the contaminated normal distribution, with clustering by k-means, and finally with Gaussian mixture models. That covariance and correlation walk straight into this trap is something we will see in the chapter Covariance.

Take-home message

Before you aggregate a table or read a trend off it, ask whether the data is secretly stitched together from several samples. If it is, calculate group by group. And check the absolute numbers, not only the percentages.

Exercise: Simpson in the Iris dataset
The Iris dataset (which we use often from the chapter Iris onwards) records, among other things, sepal length and sepal width for 150 plants of three species. Determine the relationship between the two measurements once across all 150 plants and once per species. What do you observe?

Solution
Across all the plants the relationship is weakly negative (correlation coefficient about -0.12): wider sepals go with shorter ones. Within each species it is clearly positive (setosa about 0.74, versicolor 0.53, virginica 0.46). The species is the confounder: *setosa* has short, wide sepals, the other two long, narrow ones. Anyone who does not know the species sees a trend that holds in no group at all. How the numbers are computed is shown in the remark *For data* in the chapter Covariance.

4.1.4 Visualisation

Why visualise? The first reason: summary numbers can lie without letting it show. Alberto Cairo invented the 'Datasaurus' to point out the difference between summary statistics and visualisation:

> Never trust summary statistics alone; always visualize your data.
>
> Alberto Cairo, 2016

Justin Matejka and George Fitzmaurice developed it into the 'Datasaurus Dozen' Matejka and Fitzmaurice [2017]: twelve datasets that all share the same summary statistics (the same means and standard deviations of x and y to two decimal places, and the same correlation coefficient of about -0.06 as well; what that measures comes in the chapter Covariance and dependence) but look very different indeed.

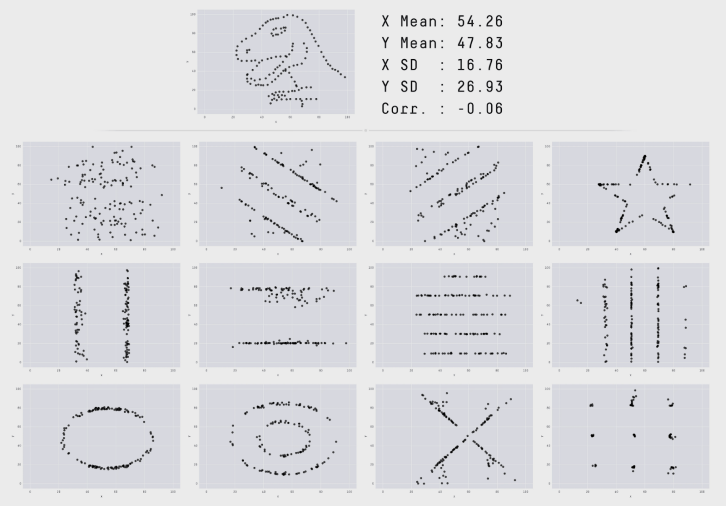


Figure 14: The Datasaurus Dozen, inspired by Anscombe's quartet. Figure: Justin Matejka and George Fitzmaurice, Autodesk Research 2017 Matejka and Fitzmaurice [2017].

The second reason to always plot is Simpson's paradox from above: a single summary measure over all the data can say the opposite of what holds in every group, and in a plot you can see the groups.

- The most important kinds of plot
  - Histograms
    To visualise the frequency distribution of a feature on a cardinal scale, you can put the feature on the x-axis and draw rectangles side by side, each with an area that represents the frequency of one class – the sum of the frequencies over the interval beneath it on the x-axis. The length of that interval is called the bucket size, bin size or class width. Choosing the right bin size is a science of its own, and there are various algorithms for computing a "good" one.

Histograms are thus a special kind of bar chart or column chart. For features on a nominal scale you can look at the corresponding bar chart directly: rectangles of equal width whose heights give the frequency of each level. For that, some arbitrary ordering of the levels has to be chosen as well.

The ordering is arbitrary only for *nominal* features, though; for an ordinal feature the x-axis should follow the order, otherwise you throw away the information it carries. To show several groups in the same diagram, you stack the bars on top of one another (`plt.bar(..., bottom=...)`) – a *stacked bar chart*. That makes sense precisely when the sum of the groups means something in itself; otherwise bars placed side by side are easier to compare, because they all stand on the same baseline. A simple example with `plt.bar` is in the chapter Titanic EDA.

So a histogram is really a statistical transformation of continuous data (on a cardinal scale) into discrete data, which is then drawn as a bar chart.

Recommended: the article "Histogram" on Wikipedia, without code, and the histogram examples in the Matplotlib documentation, with code

– Scatter plot

A set of vectors in $\mathbb{R}^2$ can be displayed by drawing a symbol – a filled circle, say – at each point's coordinates. With tabular data, you typically pick out two variables, one for the x-axis and one for the y-axis.

In Matplotlib that is a single line, and if the data is already an $(N, 2)$ array, the two variables are its two columns:

```
plt.scatter(x, y, marker="o")
plt.scatter(data[:, 0], data[:, 1])                  # the same for an (N,2) array
plt.scatter(data[:, 0], data[:, 1], c=data[:, 2])    # third column as colour
```

The third line is the trick by which a scatter plot takes on a third variable: `c=` colours every point by its value, and `cmap=` decides which colour scale is used. For a categorical feature with few levels this gives groups in different colours – that is how the Iris scatter plot in the chapter Iris is coloured (there with `cmap=plt.cm.cividis_r`), and in the chapter Exploratory analysis: examples the colour carries the grape variety in the wine dataset.

– Boxplot, also box-and-whiskers plot

Shows the median, quartiles and outliers of a distribution in one picture: the box runs from Q_1 to Q_3, the "whiskers" reach to the last value within 1.5 interquartile ranges, and everything beyond that is drawn as an individual point. In Matplotlib, `plt.boxplot(x)`; the chapter Data cleaning uses the same rule for outliers.

– Line plot

When you plot the graphs of one or more functions (as we did with the PDFs), those are line plots. Data on a cardinal scale can be displayed with a line plot just as well, but connecting the points suggests a wealth of data that is not necessarily there.

Conversely, the line plot is exactly the right tool when the data really does come from a curve: you sample points along the curve and connect them. A circle, for instance, is drawn like this:

```
angle = np.linspace(0, 2 * np.pi, 360)
plt.plot(np.cos(angle), np.sin(angle), linewidth=2)
plt.axis("equal")    # otherwise the circle turns into an ellipse
```

The same principle is behind every density plot in this book: the density function is evaluated at a few hundred points and the values are connected.

– Pie chart

As a rule: wherever a pie chart will do, a bar chart will do too, and it is easier to read. Pie charts are therefore of value mainly when you want to lighten up a text without going into the statistic shown in any detail. Features on a nominal or an ordinal scale are in principle suited to them, since the pie shows the share of each level in the whole – it is a bar chart drawn in an unusual way.

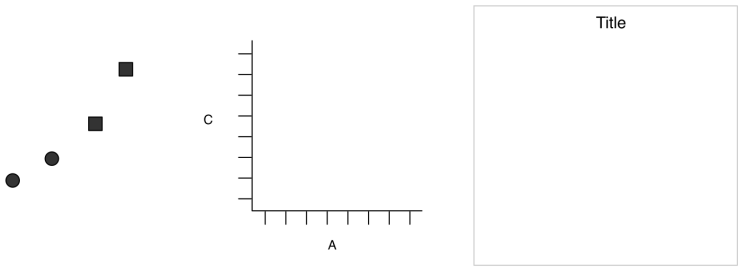
– Violin plot

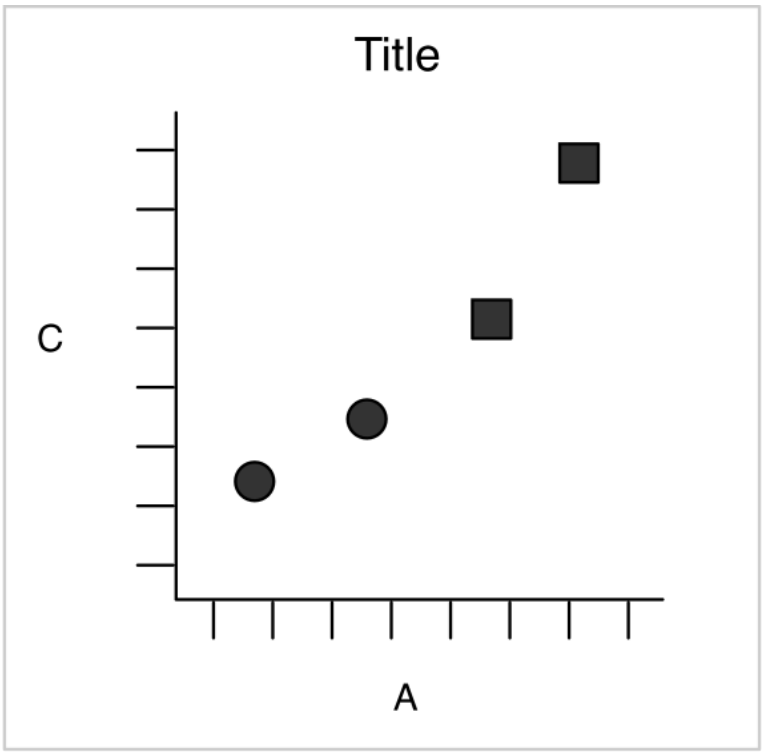
Similar to a boxplot, but in place of the rectangle the density is drawn in outline. One disadvantage is that it is rarely used, so readers are not accustomed to this kind of plot.

  - Map plot
    With the image of a map (or some other kind of map, a network topology, say) in the background, features can be displayed spatially – as the colour of a bounded area, for instance. This is akin to a histogram: instead of a scatter plot, two-dimensional regions (rather than intervals) are aggregated over before the aggregated data is displayed. A special case is a "map" that is simply a grid onto which frequencies are plotted – a two-dimensional histogram.
  - countless others, used more rarely or not fitting this scheme – which means of communication is most suitable always depends on the case at hand.

- The most important tools for plotting with Python
  - `matplotlib` – this is what we use; it is the most widely used package for Python
  - The object-oriented interface of `matplotlib` – recommended by the developers
  - Seaborn – a layer on top of `matplotlib` that makes far-reaching assumptions about what might look good, with a focus on ease of use
  - Plotnine – an implementation of the grammar of graphics in Python, analogous to R's `ggplot2`
  - Bokeh – interactive web plots, e.g. the demo on the IMDB dataset
  - Pygal – generates SVG files, which together with HTML5 can also be used for animations.

- The grammar of graphics
  - History
    1983: Bertin: Semiology of Graphics
    2000: Wilkinson, Anand, Grossman: The Grammar of Graphics
    2005: Wickham: `ggplot2` R package
    2010: Wickham: A Layered Grammar of Graphics
  - Layers of the grammar
    * Data – what is to be visualised?
    * Aesthetics – axes, dimensions
    * Scales – should the values be scaled? Nominal or ordinal?
    * Geometry – points, lines, polygons
    * Statistics – means, for example
    * Facets – subplots
    * Coordinate systems – Cartesian, polar or logarithmic
  - Combining the parts:





→

- A layer consists of
  - Data and aesthetic mappings (aes)
  - Statistical transformations (stat)
  - Geometric objects (geom)
  - Position adjustments
- A plot consists of
  - A default dataset and mappings of the variables in it to aesthetics
  - One or more layers (not necessarily with additional data)
  - One scale per aesthetic mapping
  - A coordinate system
  - A facet specification
- A statistical transformation is typically a *summary* of data.

  Example: smoothing, which turns a set of points into a curve that is smooth (infinitely differentiable).

  We have already met some of the most important statistical transformations implemented in ggplot2:
  - bin: divide a continuous range into bins (classes) and count the points in each class.
  - boxplot: compute the statistics for a boxplot
  - contour: compute contour lines
  - density: compute a one-dimensional density estimate
  - identity: the identity map, f (x) = x
  - jitter: shift every value by a small random amount
  - smooth: smoothed mean of y given x
  - summary: summed values of y for each x
  - unique: remove duplicates
- A scale, in the sense of the grammar of graphics, controls how data is mapped to aesthetic attributes.

  Example: a real variable between 0 and 1 can be displayed as a change of colour from blue to red. The scale is the mapping of all the values in $[0, 1]$ to colour values; its inverse can be used to draw a *legend*.

Example: a factor variable (the species in the Iris dataset, say) is displayed with different symbols (in a scatter plot, say). The mapping from level to symbol is the scale.

You can also map several variables onto one aesthetic – by mapping different values onto saturation and hue in the HSV colour model, for instance.

– Colours

When choosing colour scales you have to think about distinguishability, but also about the people who will look at the plots later. Two reasons never to use colour as the only distinguishing feature:

First, the circumstances. A plot gets looked at in sunlight on a laptop, projected through a badly calibrated beamer, or printed out – in the past routinely in greyscale. What survives is the lightness. A colour scale should therefore still work once you take the colour away, and curves are additionally told apart by line style (`linestyle="--"`, dotted, different line widths) or by markers (`marker="x"`, circles, triangles).

Second, the viewers themselves. About one man in twelve and one woman in two hundred perceive colours differently from the majority, most often red and green. In a lecture with two hundred listeners there are therefore typically several people for whom a red-green scale is no scale at all. That is often enough to make it a habit rather than an exception.

A colour palette that works well even with restricted colour perception is this one:



That it works can be checked by calculation: a colour vision deficiency can be simulated as a linear map in colour space, and applied to the eight colours above, that looks like this.

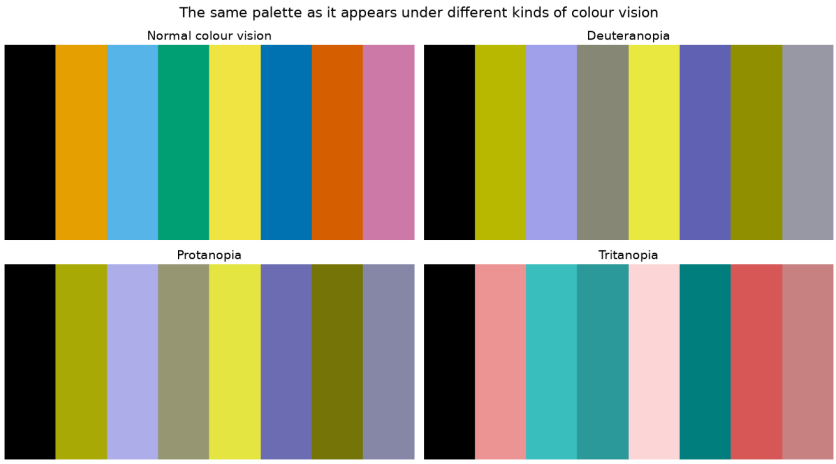


Figure 15: The same palette, simulated for normal colour vision and for the three kinds of dichromacy (method after Viénot, Brettel and Mollon 1999). The colours stay distinguishable in all four panels – that is the property that matters.

Two things can be seen from it. First, the palette does lose its colourfulness, but no two patches merge – every colour keeps a lightness of its own. Second, a palette that set red against green would merge here, and that is exactly what happens with the widespread traffic-light coding.

Research into suitable colour palettes is not finished – at the SciPy conference in 2022, for instance, there was the talk Development of Accessible, Aesthetically-Pleasing Color Sequences by Matthew Petroff, whose results went into Matplotlib version 3.10.0 in December 2024.

4.1.5 Exploratory data analysis as a process

The tools of this chapter – levels of measurement, measures of location and spread, visualisation – are means to an end. The end is called *exploratory data analysis* (EDA): opening up a dataset step by step until you can say something about it. “Step by step” is meant literally here, and above all: in a circle.

You go through the following five steps not once but again and again, because every pass throws up new questions.

1. **Formulate the research questions.** What do you actually want to know? What levels of measurement do the features have, what distributions do you expect, which relationships do you consider plausible? Without this step you produce diagrams instead of insights. In the chapter Exploratory analysis: examples this step comes expressly first.

2. **Visualise the data.** The quickest way to see a distribution, an outlier or a relationship is a picture – that was the previous section, and the Datasaurus showed why summary numbers alone are not enough.

3. **Clean the data.** Check outliers and remove them where necessary, impute or interpolate missing values, repair errors in the coding. In the chapter Titanic EDA missing ages are replaced by the median; the chapter Data cleaning turns that into a programme of its own.

4. **Transform the data.** Select subsets, rescale features, build new features – age groups out of age (*binning*), say, or an area out of length and width. Such new columns are features too, and finding them is often where the real insight lies.

5. **Model the data.** Choose a statistical model and fit it to the data – a normal distribution or a regression line, for instance. The chapters that follow are about that.

The five steps are a finer resolution of the whole picture that Figure 12 showed at the start of the chapter. And because they run in a circle, it is also clear why EDA cannot be automated: step 1 needs somebody with a question, and step 3 somebody who knows which values are implausible in this field of application. We practise that in the chapters that follow, on several datasets.

4.2 Exploratory data analysis: wine, heavy rainfall, Titanic

We now work through the first steps of exploratory data analysis – as much of it as we have learned so far – on a few examples.

```
import numpy as np
from scipy import stats as st
import matplotlib.pyplot as plt
```

4.2.1 Wine

We begin with an old dataset, which can be downloaded from the UC Irvine Machine Learning Repository, among other places:

Wine. (1991). UCI Machine Learning Repository.

```
url = "https://archive.ics.uci.edu/ml/machine -learning -databases/wine/wine.data"
```

According to the documentation the dataset has 178 rows (called *instances* here, as they often are) and 14 variables. The first variable is categorical, encoded as an `int` from `{1, 2, 3}`; the others are continuous real variables (hence `float`). The licence is CC-BY-4.0, so we may use the dataset freely, modify it and share our results, provided we name the original source:

> Riccardo Leardi, riclea@anchem.unige.it

The dataset comes as two files:

- `wine.data`
- `wine.names`

Here `wine.data` is the dataset proper, without any metadata, and `wine.names` holds the metadata in human-readable form.

From `wine.names` we take the following in particular:

> These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines.

- The attributes are
  1. Alcohol
  2. Malic acid
  3. Ash
  4. Alcalinity of ash
  5. Magnesium
  6. Total phenols
  7. Flavanoids
  8. Nonflavanoid phenols
  9. Proanthocyanins
  10. Color intensity
  11. Hue
  12. OD280/OD315 of diluted wines
  13. Proline
- Number of Instances
  - class 1 59
  - class 2 71
  - class 3 48

Now that we have successfully downloaded the data and read through the metadata that come with them, we can ask what we actually want to do with them.

One obvious task is to find out whether the three classes (that is, the three producers of the wine) can be told apart by the wine's chemical composition. In statistical terms: do the distributions of the 13 variables differ enough from class to class for the individual data points (the instances) to be assigned to a class on that basis?

We can set this up as a game: if we delete the first variable, the class, can it be reconstructed from the rest? In the simple version we do not delete the class yet, but use it to estimate three different distributions for each of the 13 attributes. Then, for every instance and ignoring its class, we judge which distribution it most likely belongs to, and check whether we were right.

In the advanced version we split the data into training and test data, or form the three (or more) classes without using the variable "class" at all. First, though, let us take a look at the data.

Quite apart from the classes, one thing is interesting straight away: how are the data distributed? We ask that now for each variable on its own.

```
# Numpy's genfromtxt assumes whitespace -separated data,
#   so we state the delimiter explicitly:
data = np.genfromtxt(url, delimiter=",")
print(type(data), data.shape)
print(data[0]) # one instance
```

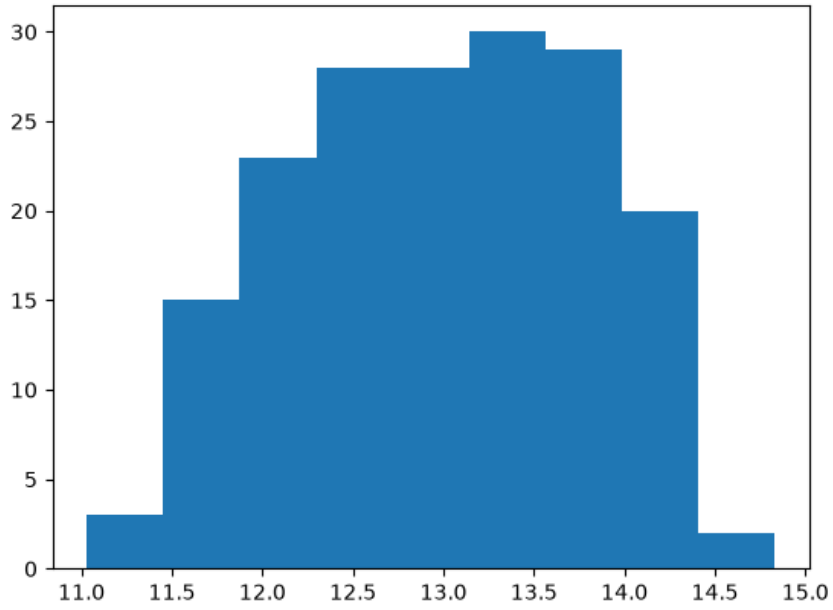
```
<class 'numpy.ndarray'> (178, 14)
[1.000e+00 1.423e+01 1.710e+00 2.430e+00 1.560e+01 1.270e+02 2.800e+00
 3.060e+00 2.800e -01 2.290e+00 5.640e+00 1.040e+00 3.920e+00 1.065e+03]
```

```
wine_class = data[:,0].astype("i8")
# If wine_class were a Python list:
#class_counts = [wine_class.count(d) for d in (1,2,3)]
# In Numpy it goes like this:
unique, counts = np.unique(wine_class, return_counts=True)
print(dict(zip(unique, counts)))
# If counts were a Python list:
#assert counts == [59, 71, 48]
# In Numpy it goes like this:
print(np.equal(counts, [59, 71, 48]))
assert np.all(np.equal(counts, [59, 71, 48]))
```

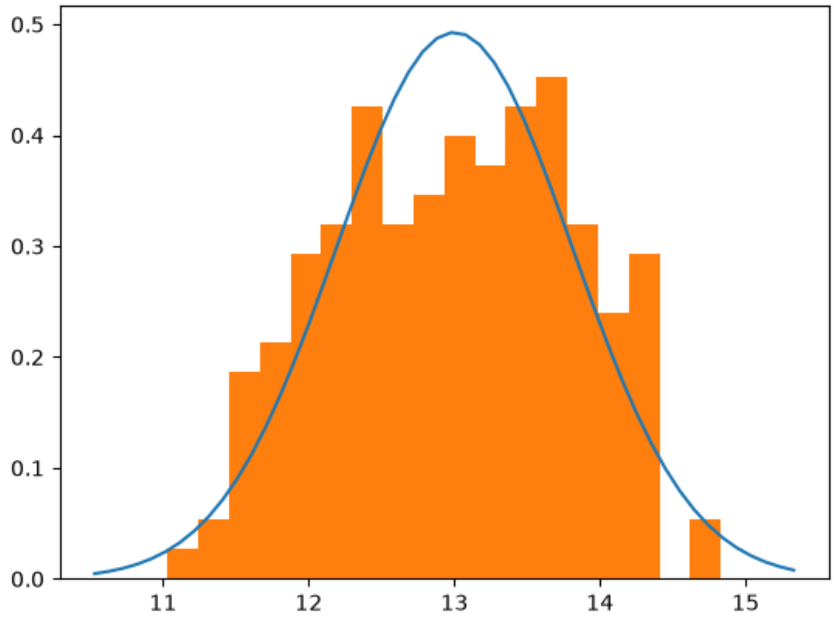
```
{np.int64(1): np.int64(59), np.int64(2): np.int64(71), np.int64(3): np.int64(48)}
[ True  True  True]
```

```
alcohol = data[:,1]
print("min", np.min(alcohol),
      "max", np.max(alcohol),
      "range", np.max(alcohol) - np.min(alcohol),
      "std", np.std(alcohol),
      "n", len(alcohol))
plt.hist(alcohol, bins="auto")
plt.show()
```

```
min 11.03 max 14.83 range 3.8000000000000007 std 0.8095429145285168 n 178
```



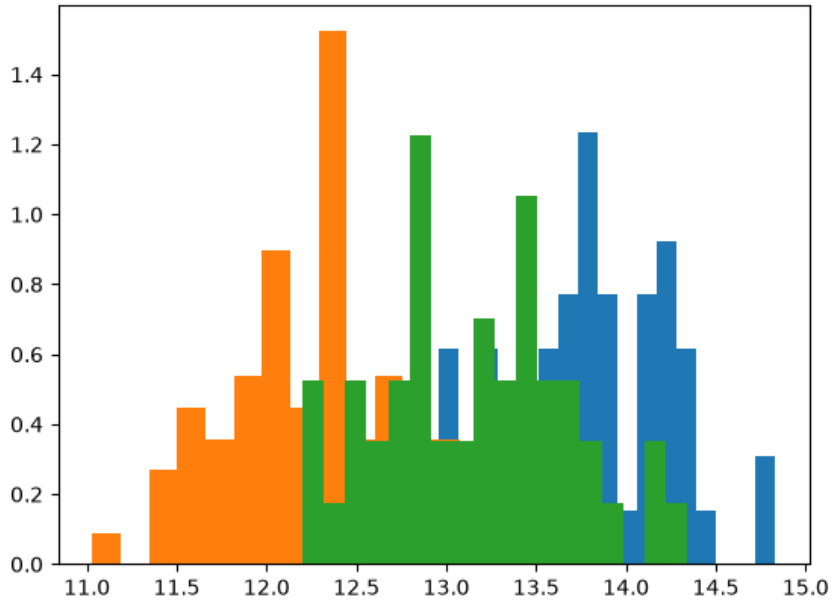

```
# Looks roughly normally distributed, so:
mu, sigma = st.norm.fit(alcohol)
x = np.linspace(np.min(alcohol) -0.5, np.max(alcohol)+0.5, 50)
plt.plot(x, st.norm(mu, sigma).pdf(x))
plt.hist(alcohol, density=True, bins=18)
plt.show()
```



We have to admit that this distribution does not look particularly normal. You could almost make out a bimodal distribution – a mixture, that is, of two normal distributions with expected values near 12 and 13.5. Rather than go on guessing from the picture, let us note that we shall need a method for testing such guesses rigorously.

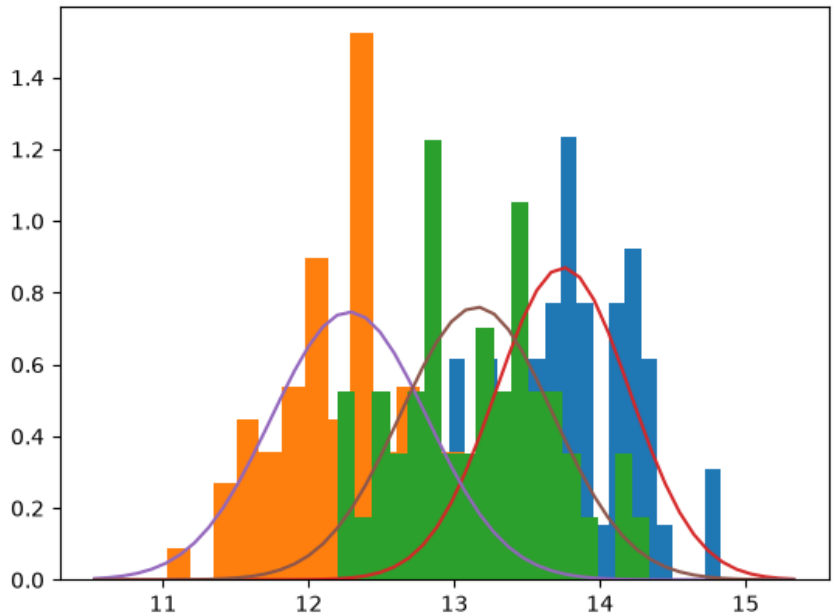
In our example we can press the class into service as an auxiliary variable:

```
alcohol_by_class = [alcohol[data[:,0] == c] for c in (1,2,3)]
# This is now a list of length 3,
#   each entry holding the alcohol values within one class.
for alcohol_class in alcohol_by_class:
    plt.hist(alcohol_class, density=True, bins=18)

plt.show()
```



And indeed, clear differences show at once. The 'blue' wine (class 1) has more alcohol on average than the others, the 'orange' wine (class 2) less. Let us draw in the corresponding normal densities:

```
for alcohol_class in alcohol_by_class:
    plt.hist(alcohol_class, density=True, bins=18)

x = np.linspace(np.min(alcohol) -0.5, np.max(alcohol)+0.5, 50)
for alcohol_class in alcohol_by_class:
    mu, sigma = st.norm.fit(alcohol_class)
    plt.plot(x, st.norm(mu, sigma).pdf(x))

plt.show()
```

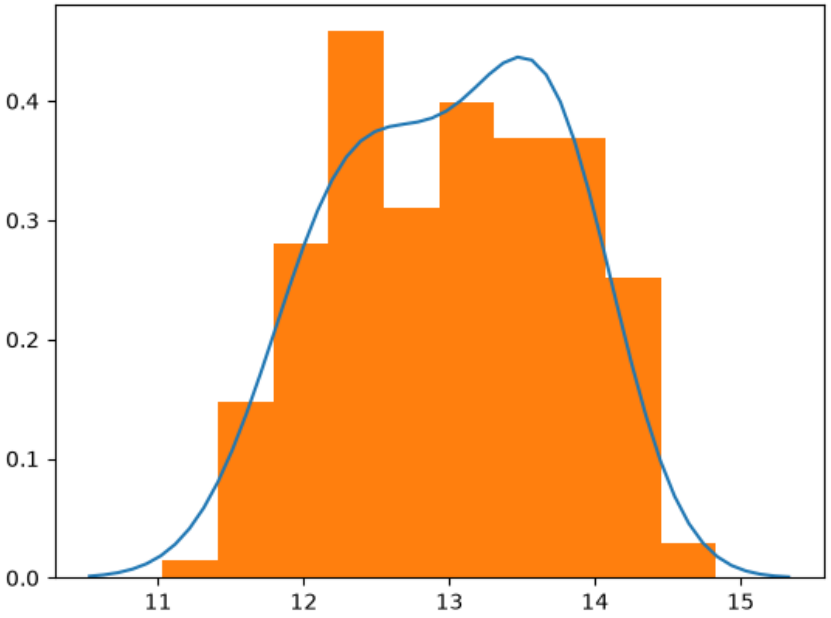


Unfortunately it is just as clear that the overlaps are so large that between 12 and 14.5 per cent alcohol by volume no sensible prediction of the class is possible.

For completeness we also model the mixture distribution:

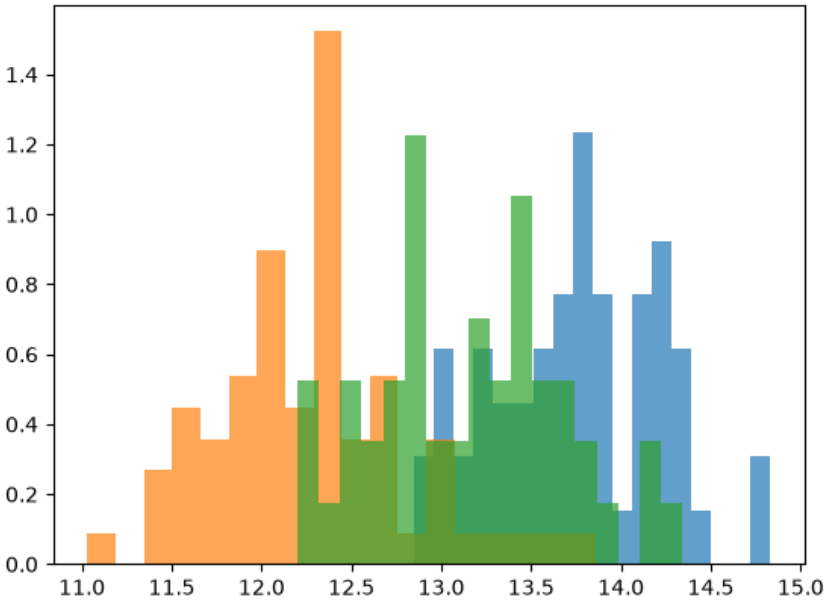
```
class_frequencies = counts / len(alcohol) # (59/178, 71/178, 48/178)
params = [st.norm.fit(alcohol_class) for alcohol_class in alcohol_by_class]
pdfs = [st.norm(*param).pdf for param in params]
```

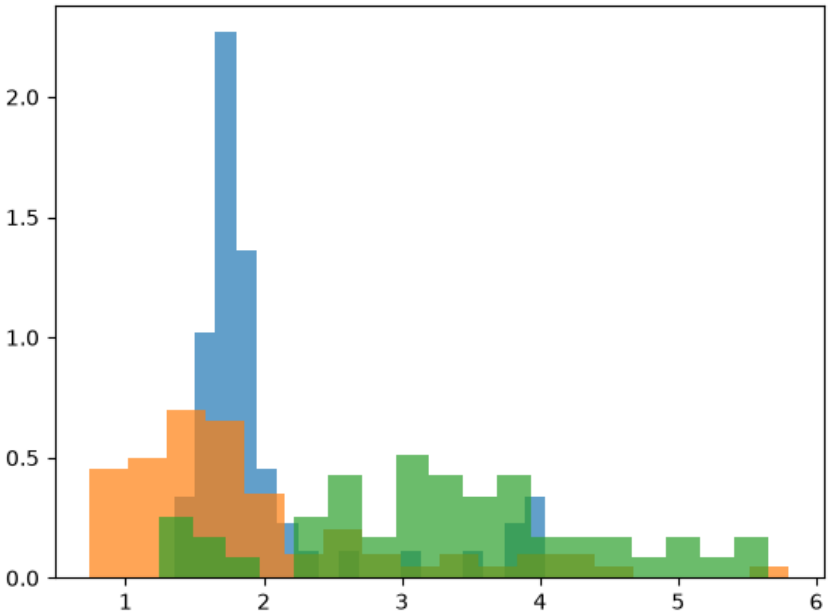
```
def mixed_alcohol(x):
    return sum((class_frequency * pdf(x)
                for class_frequency, pdf
                in zip(class_frequencies, pdfs)))

plt.plot(x, mixed_alcohol(x))
plt.hist(alcohol, density=True)
plt.show()
```

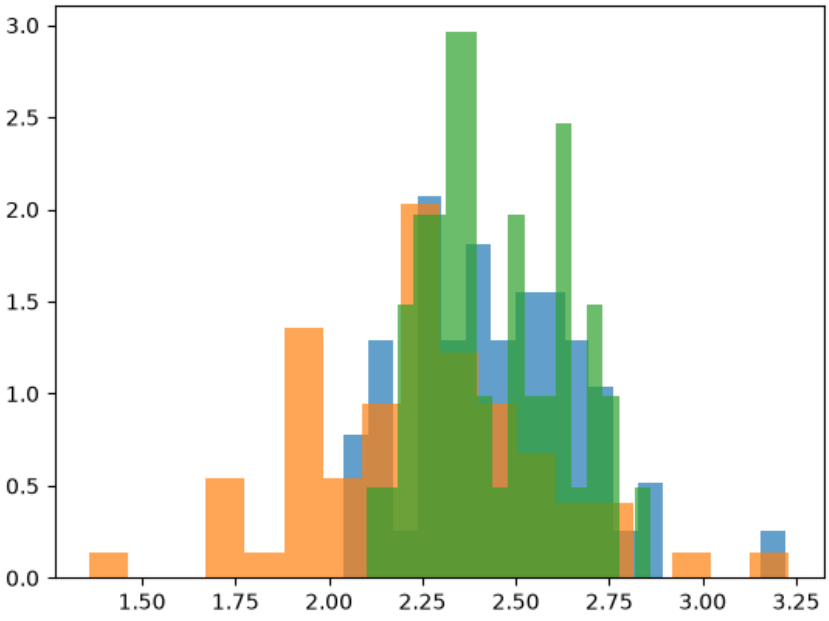


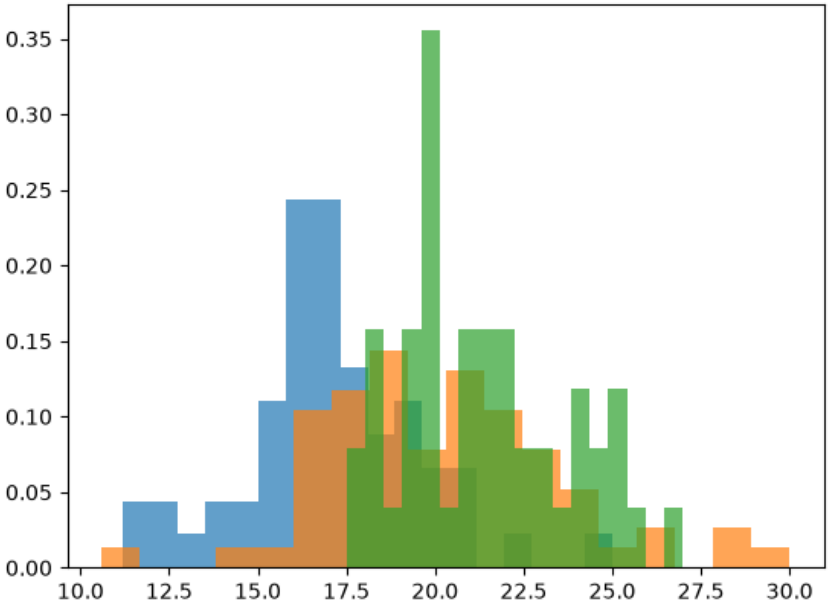
Having seen that the first attribute, the alcohol content, depends on the class, it is natural to examine all the other attributes the same way. So we rewrite the code generically and look at all the variables at once:

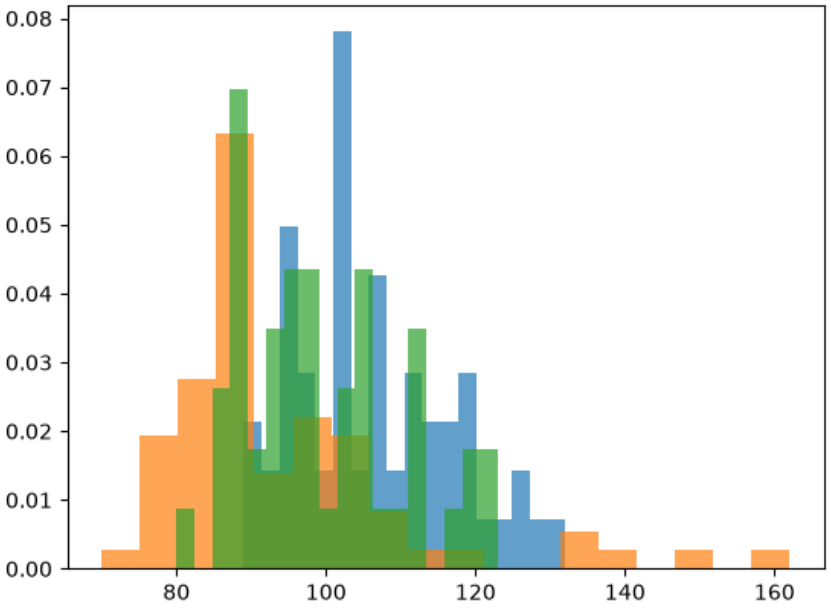
```
attributes_by_class = [[data[:,i][data[:,0] == c]
                        for c in (1,2,3)]
                       for i in range(1,14)]
# This is now a list of length 13,
#   each entry holding a list of length 3

# 13 plots with 3 histograms each:
for attribute in attributes_by_class:
    for attribute_class in attribute:
        plt.hist(attribute_class, density=True,
                 bins=18, alpha=0.7)
    plt.show()
```

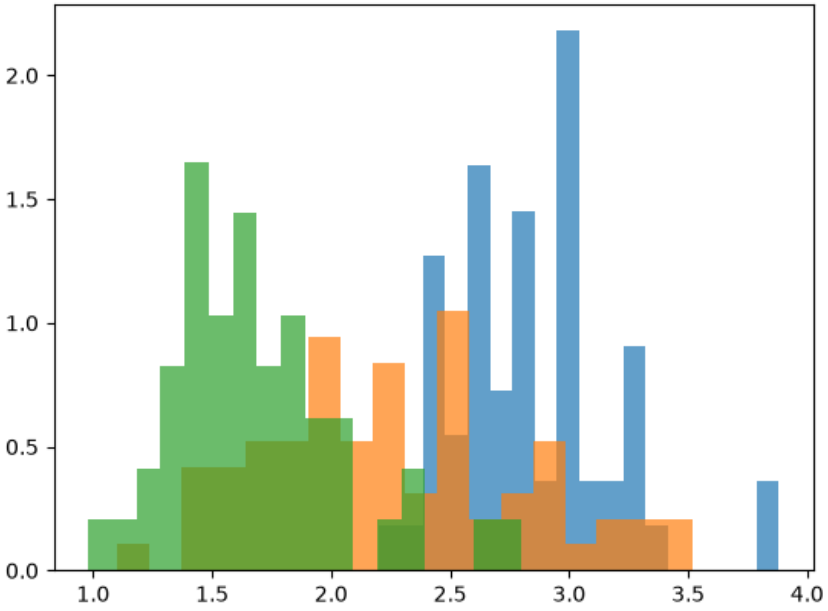

1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
11.0
11.5
12.0
12.5
13.0
13.5
14.0
14.5
15.0

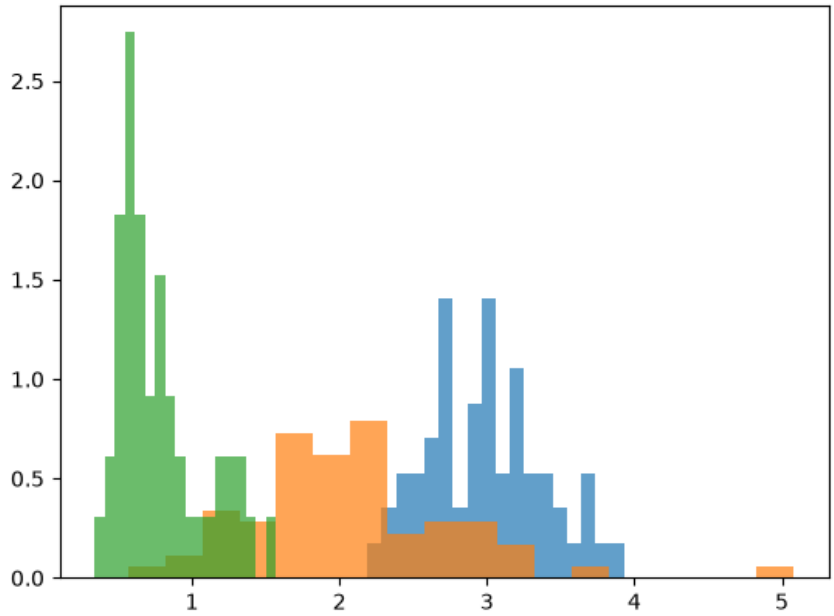

2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6

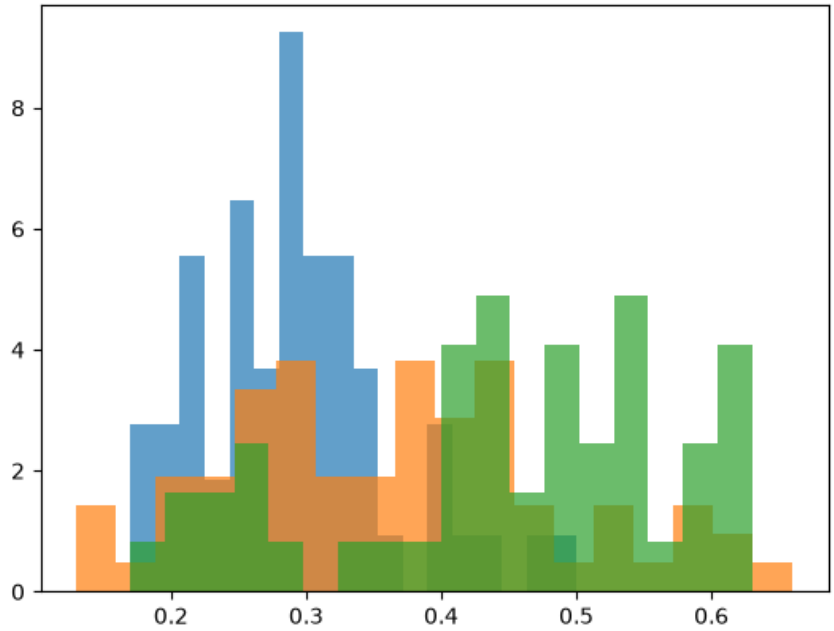

3.0
2.5
2.0
1.5
1.0
0.5
0.0
1.50
1.75
2.00
2.25
2.50
2.75
3.00
3.25

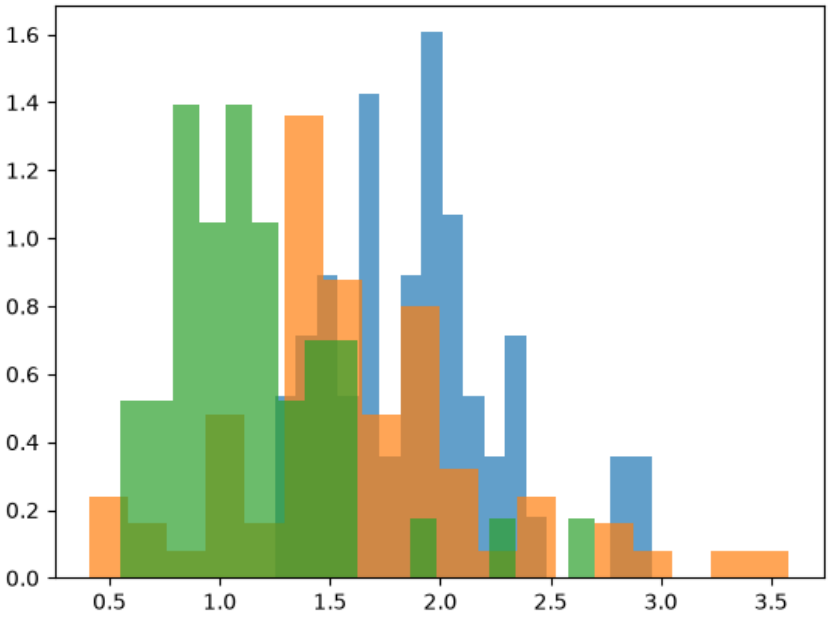

0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
10.0
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0

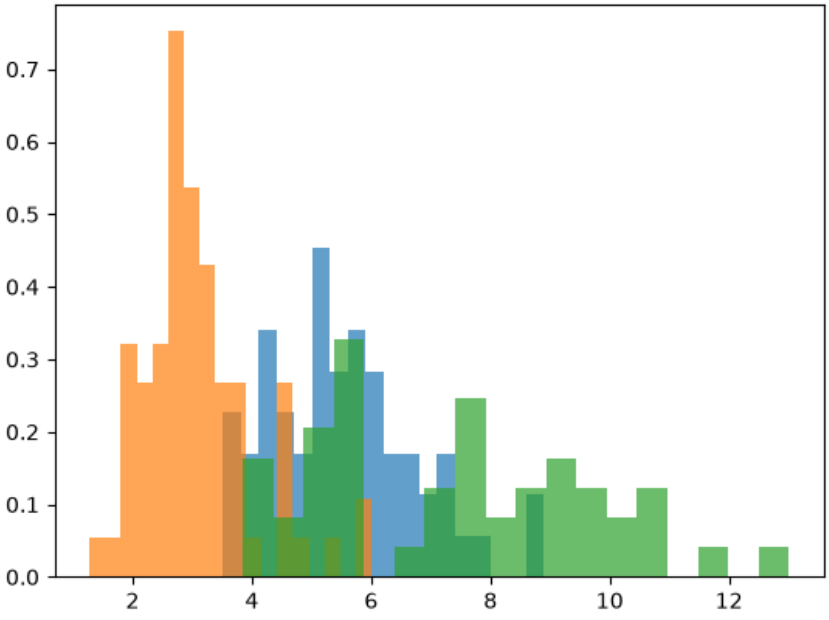

0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
80
100
120
140
160

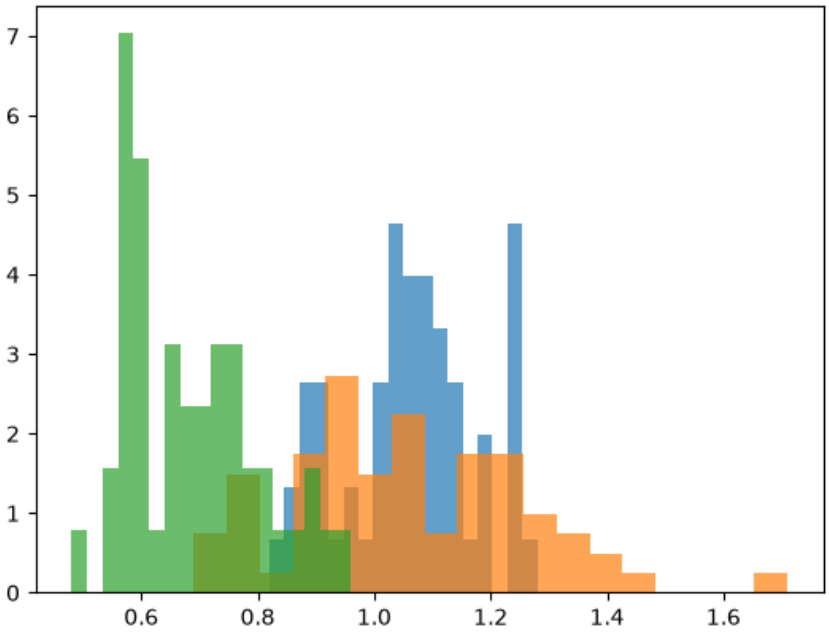

2.0
1.5
1.0
0.5
0.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0

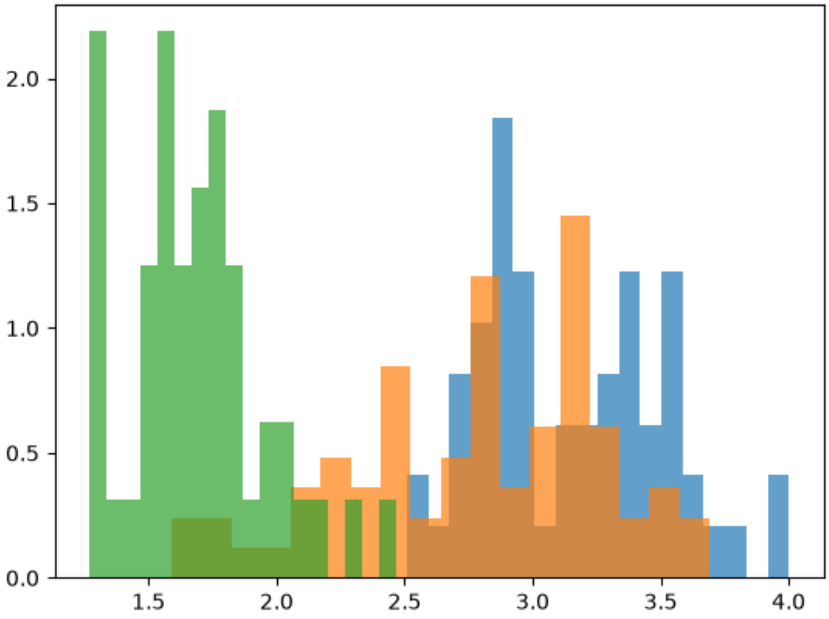

2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5

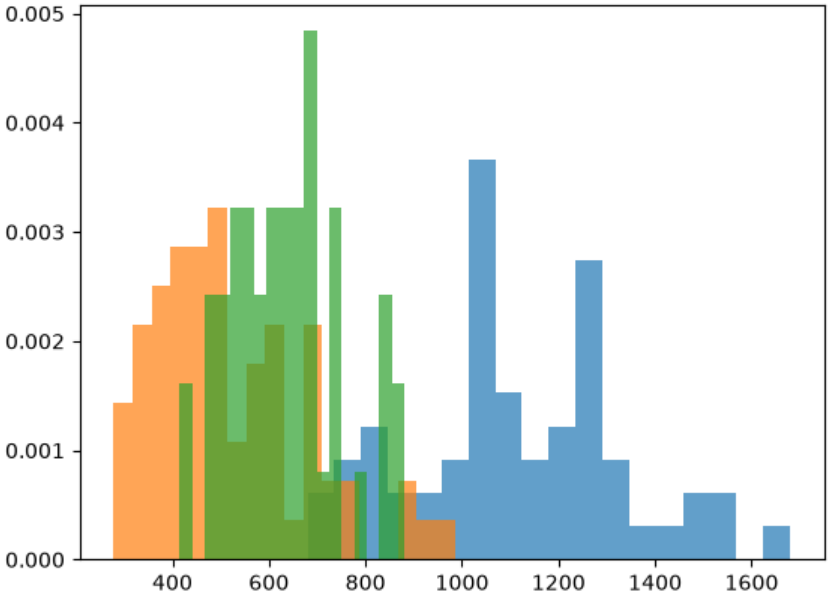

8
6
4
2
0
0.2
0.3
0.4
0.5
0.6


1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
2
4
6
8
10
12

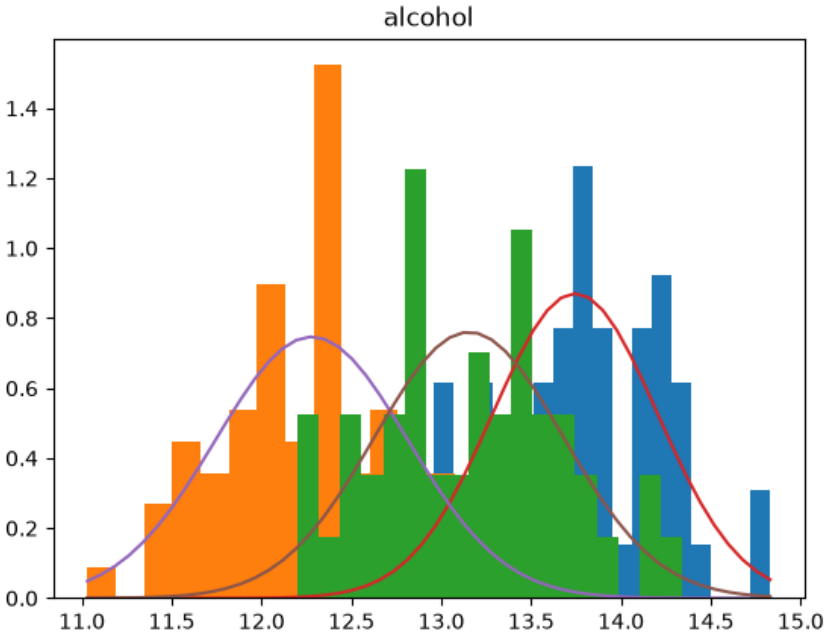

7
6
5
4
3
2
1
0
0.6
0.8
1.0
1.2
1.4
1.6

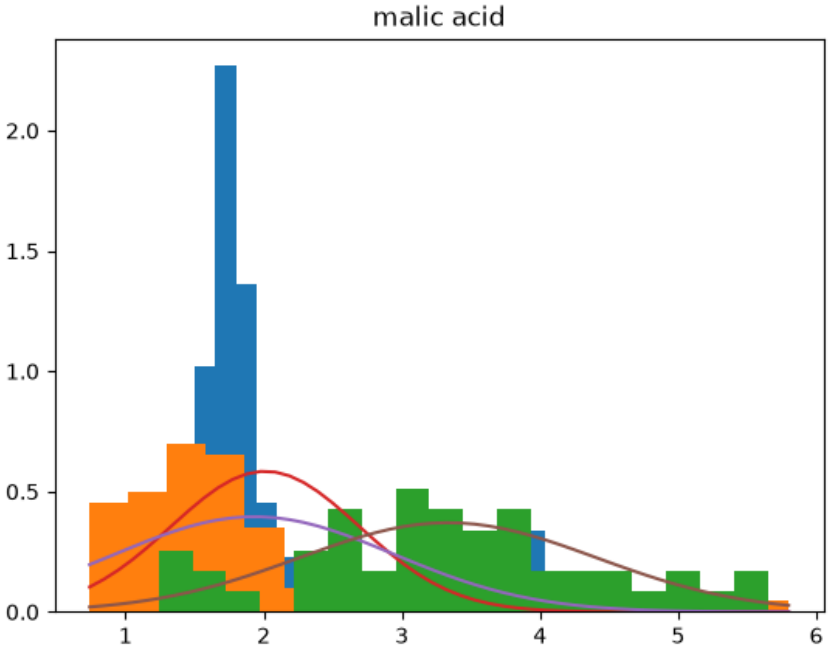

2.0
1.5
1.0
0.5
0.0
1.5
2.0
2.5
3.0
3.5
4.0

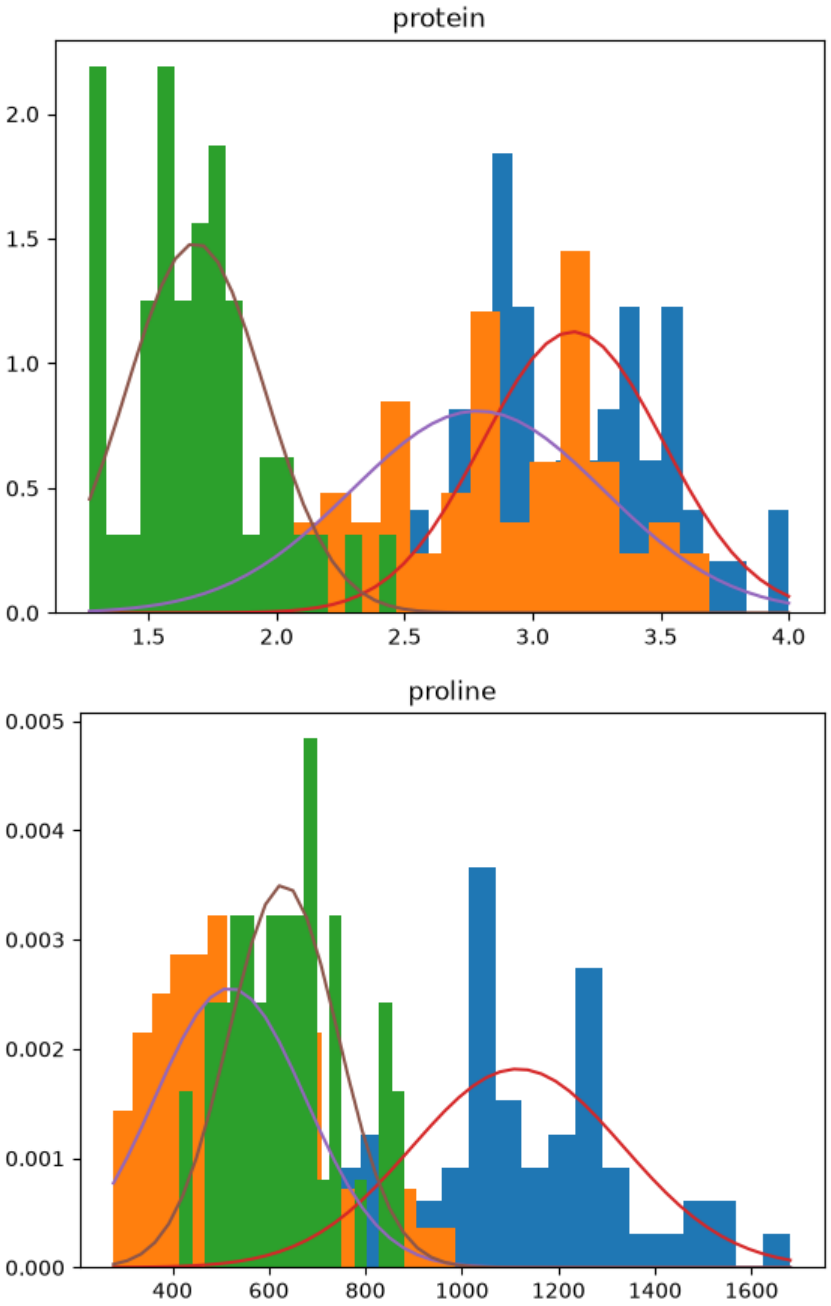


The pictures suggest that, besides attribute 1 (alcohol), attributes 2 (malic acid), 12 (OD280/OD315 of diluted wines, the protein content) and 13 (proline, an amino acid) might also serve to keep the classes apart.

```
# uses attributes_by_class
def plot_attribute_by_class(i, title=""):
    """use numbering from dataset, so alcohol = 1"""
    plt.title(title)
    attribute_by_class = attributes_by_class[i -1]
    attribute = data[:,i]
    xmin, xmax = np.min(attribute), np.max(attribute)
    for attribute in attribute_by_class:
        plt.hist(attribute, density=True, bins=18)
    x = np.linspace(xmin, xmax, 50)
    for attribute in attribute_by_class:
        mu, sigma = st.norm.fit(attribute)
        plt.plot(x, st.norm(mu, sigma).pdf(x))
    plt.show()

plot_attribute_by_class(1, "alcohol")
plot_attribute_by_class(2, "malic acid")
plot_attribute_by_class(12, "protein")
plot_attribute_by_class(13, "proline")
```


alcohol
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
11.0
11.5
12.0
12.5
13.0
13.5
14.0
14.5
15.0

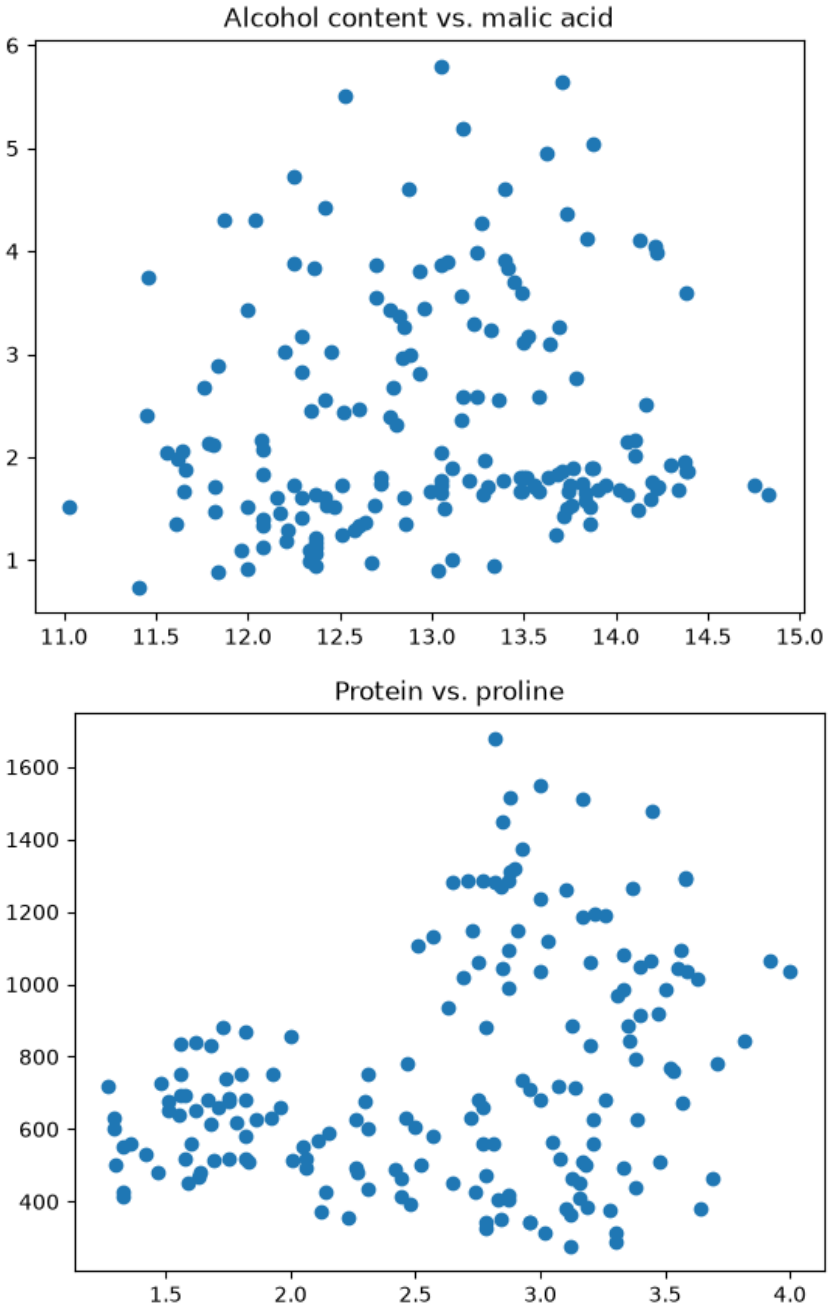

malic acid
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6



From this we see that malic acid is a rather poor indicator of the class, whereas protein and proline content together look considerably better suited to prediction than, say, the alcohol content. More precisely: protein content separates class 3 from classes 1 and 2, and proline content separates class 1 from classes 2 and 3. Neither separation is really clean, however, as the plot already shows.

Finally, we look at the relationship between individual attributes with a scatter plot:

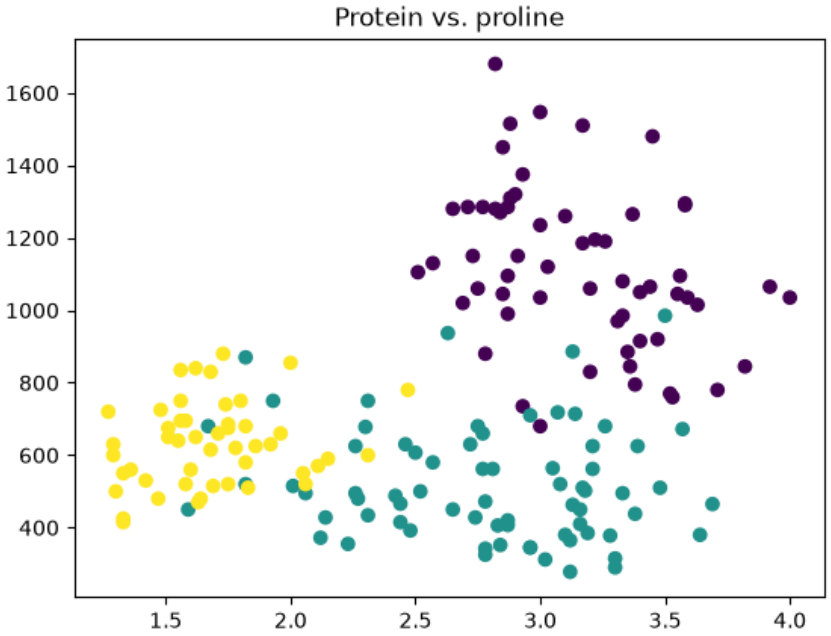
```
plt.title("Alcohol content vs. malic acid")
plt.scatter(data[:,1], data[:,2])
plt.show()
plt.title("Protein vs. proline")
plt.scatter(data[:,12], data[:,13])
plt.show()
```



Whereas alcohol content and malic acid show no obvious relationship, the protein-proline plot has a pronounced white patch: a high proline content goes together with a high protein content. It would now be interesting to compute measures of this relationship, and also to search the specialist literature for an explanation of how it comes about.

To see whether the relationship is only apparent, and perhaps really arises from a dependence on the class, we look at the scatter plot once more, with the classes distinguished by colour:

```
plt.title("Protein vs. proline")
plt.scatter(data[:,12], data[:,13], c=data[:,0])
plt.show()
```



And indeed it is quite plain, as it was in the histograms before, that these two attributes taken together separate the classes reasonably well.

What we would like, of course, is a coordinate-independent procedure for separating the classes, so that we do not have to pick two variables by hand. Later, principal component analysis (PCA) will give us one.

4.2.2 Heavy rainfall events (CatRaRE)

The German weather service (DWD) makes data on heavy rainfall events available:

> Catalogues of radar-based heavy precipitation events, as a basis for analysing heavy rainfall activity and its damage potential in Germany since 2001

- The description of the datasets is to be found at the DWD (in German)
- The metadata (a description of the attributes) are available as an XLSX (in German)
- The data download links have to be cited whenever the data are used (in German)
  Lengfeld, Katharina; Walawender, Ewelina; Winterrath, Tanja; Weigl, Elmar; Becker, Andreas (2021): Starkregenereignisse Version 2021.01 mit Überschreitung der DWD-Warnstufe 3 für Unwetter basierend auf RADKLIM-RW Version 2017.002,
  DOI: 10.5676/DWD/CatRaRE_W3_Eta_v2021.01
- The CSV file we want to look at can be downloaded directly

```
url = "https://opendata.dwd.de/climate_environment/CDC/event_catalogues/germany/precipitation/
    CatRaRE_v2021.01/data/CatRaRE_2001_2020_W3_Eta_v2021_01.csv"
# Column types of the 60 columns in the order of the CSV header row, following CatRaRE's
    metadata table:
# i8 = integer, f8 = floating -point number, S10/S23/... = string with that maximum length
data = np.genfromtxt(url, delimiter=",", names=True,
                     dtype="i8,S10,S10,i8,i8,i8,S23,S23,i8,i8,f8,f8,f8,i8,i8,i8,i8,i8,i8,S23,S23
    ,S23,f8,f8,f8,S12,S12,i8,i8,i8,i8,S6,i8,S6,i8,i8,f8,i8,i8,i8,f8,i8,i8,i8,i8,f8,f8,f8,i8,f8,
    i8,i8,i8,i8,i8,f8,i8,f8,f8,f8,f8")
data[0]
```

```
np.void((1, b'W3_Eta_v20', b'W3_Eta_1', 1, 200101020350, 200101020750, b'2001 -01 -02T03
    :49:59.000', b'2001 -01 -02T07:50:00.000', 4, 51, 44.6, 51.0, 100.0, 1, 286, 183, -157462,
    -4575645, 8315037, b'Baden -W\xfcrttemberg', b'Breisgau -Hochschwarzwal', b'Feldberg (
    Schwarzwald)', 2.6, 51.5, 38.0, b'28', b'4', 5, 3, 0, 19, b'SWAZF', 19, b'SWAZF', 19, 12,
```

```
    16.5, 20, 29, 19, 24.8, 30, 6, 620, 21, 0.1, 0.5, 64.9, 0, 0.4, 85, 312, 312, 1187, 585,
    1102.3, 1462, -26.1, -311.9, -8.6, 223.5), dtype=[('OBJECTID', '<i8'), ('CatVersion', 'S10')
    , ('KEY_FIELD', 'S10'), ('Event_ID', '<i8'), ('Start_Time', '<i8'), ('End_Time', '<i8'), ('
    Date_START', 'S23'), ('Date_END', 'S23'), ('Duration', '<i8'), ('Area', '<i8'), ('Area_LAEA
    ', '<f8'), ('Area_in_G', '<f8'), ('AShare_in_G', '<f8'), ('RRmax_in_G', '<i8'), ('x_RRmax',
    '<i8'), ('y_RRmax', '<i8'), ('x_RRmaxPRJ', '<i8'), ('y_RRmaxPRJ', '<i8'), ('AGS_RRmax', '<i8
    '), ('BDL_RRmax', 'S23'), ('LKS_RRmax', 'S23'), ('GMD_RRmax', 'S23'), ('Eta', '<f8'), ('
    RRmax', '<f8'), ('RRmean', '<f8'), ('Tmax', 'S12'), ('Tmean', 'S12'), ('SRImax', '<i8'), ('
    SRImean', '<i8'), ('MPP_flag', '<i8'), ('WL_START', '<i8'), ('WLc_START', 'S6'), ('WL_END',
    '<i8'), ('WLc_END', 'S6'), ('V3_RRmax', '<i8'), ('V3_MIN', '<i8'), ('V3_AVG', '<f8'), ('
    V3_MAX', '<i8'), ('V4_RRmax', '<i8'), ('V4_MIN', '<i8'), ('V4_AVG', '<f8'), ('V4_MAX', '<i8
    '), ('BVLK_RRmax', '<i8'), ('BVLK_SUMME', '<i8'), ('BVLK_D1km2', '<i8'), ('SDL_RRmax', '<f8
    '), ('SDL_GRAD', '<f8'), ('SDL_MAX', '<f8'), ('VSGL_RRmax', '<i8'), ('VSGL_GRAD', '<f8'), ('
    VSGL_MAX', '<i8'), ('CLC_RRmax', '<i8'), ('CLC_DOMIN', '<i8'), ('SRTM_RRmax', '<i8'), ('
    SRTM_MIN', '<i8'), ('SRTM_AVG', '<f8'), ('SRTM_MAX', '<i8'), ('TPI_RRmax', '<f8'), ('TPI_MIN
    ', '<f8'), ('TPI_AVG', '<f8'), ('TPI_MAX', '<f8')])
```

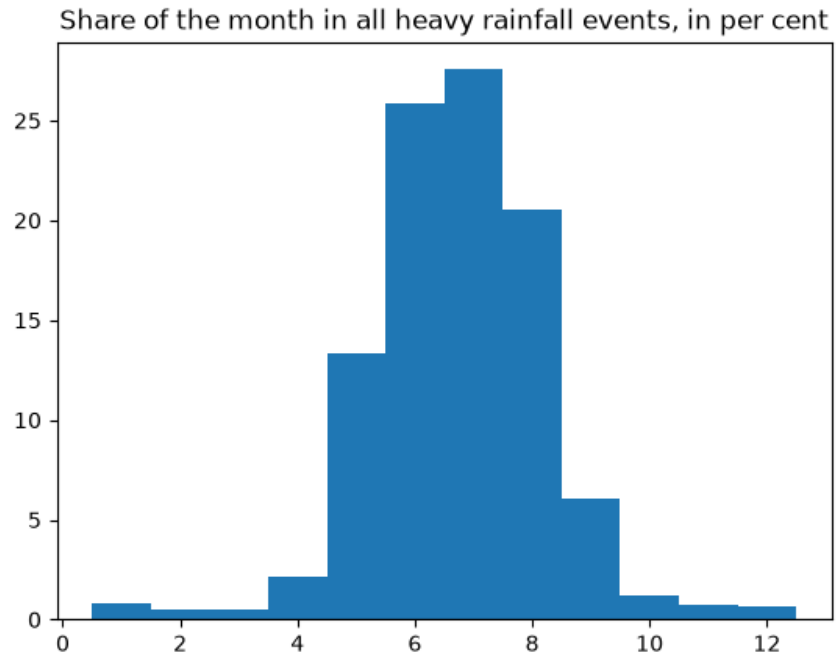
```
print(np.max(data["RRmax"]), np.min(data["RRmean"]))
print(np.mean(data["Duration"]))
```

```
462.0 25.6
5.051172893136403
```

Now we have to settle on a research question. One question is always obvious: what distribution do the variables have? We can once again look at the bare frequency, that is, at how the points in time are distributed. We could follow it over the years, or average over the years and analyse how the month influences the frequency. The latter suggests itself: heavy rain is a weather phenomenon and so depends on the seasons, and we can expect a clear relationship.

At the moment the start time of a heavy rainfall event is encoded as an `int`. We turn that to our advantage: make a `string` out of the `int` `200101020350` and extract the month from it:

```
def getmonthfromintdate(intdate):
    return int(str(intdate)[4:6])

assert getmonthfromintdate(200101020350) == 1

def getmonthcount(data, month):
    mask = [getmonthfromintdate(intdate) == month
            for intdate in data["Start_Time"]]
    return len(data[mask])

months = np.arange(1,13)
counts = np.array([getmonthcount(data, month)
                   for month in months])
densities = 100 * counts / len(data)
plt.title("Share of the month in all heavy rainfall events, in per cent")
plt.bar(months, densities, width=1)
plt.show()
```



So the expected relationship shows clearly: in July and June heavy rainfall events are considerably more frequent than in winter.

Next we can look at the location, say at the variable `GMD_RRmax` and the place `"Düsseldorf"`:

```
# Does not work:
duessel = data[data["GMD_RRmax"] == "Düsseldorf"]
len(duessel)
```

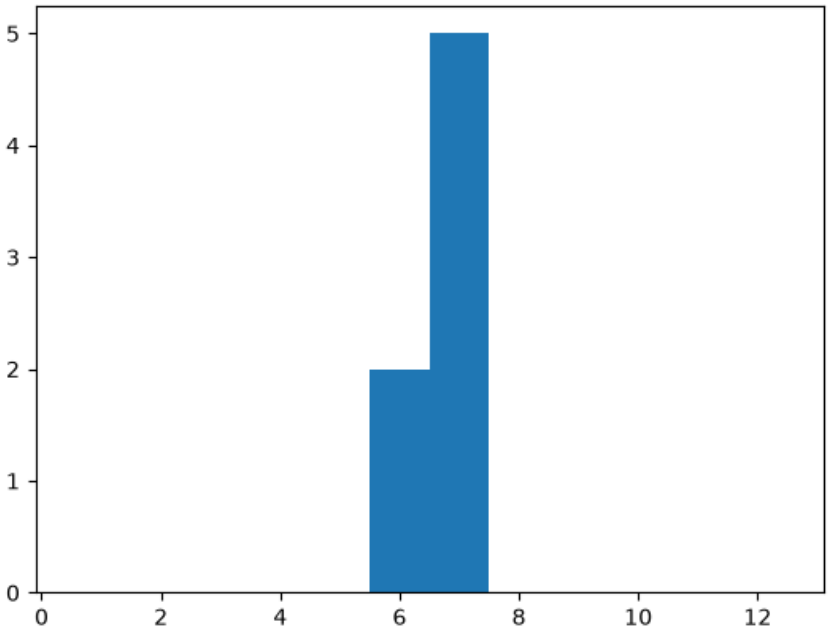
```
0
```

```
# Instead:
for entry in data["GMD_RRmax"]:
    if("sseldorf" in str(entry)):
        print(entry)
        break
```

```
b'D\xfcsseldorf'
```

```
# So (b'...' is a byte string: genfromtxt reads text columns as bytes;
#  with pandas.read_csv, see the chapter Pandas, they would be ordinary strings):
city = b'D\xfcsseldorf'
citydata = data[data["GMD_RRmax"] == city]
print(len(citydata))
citycounts = np.array([getmonthcount(citydata, month)
                       for month in months])
plt.bar(months, citycounts, width=1)
plt.show()
```

```
7
```

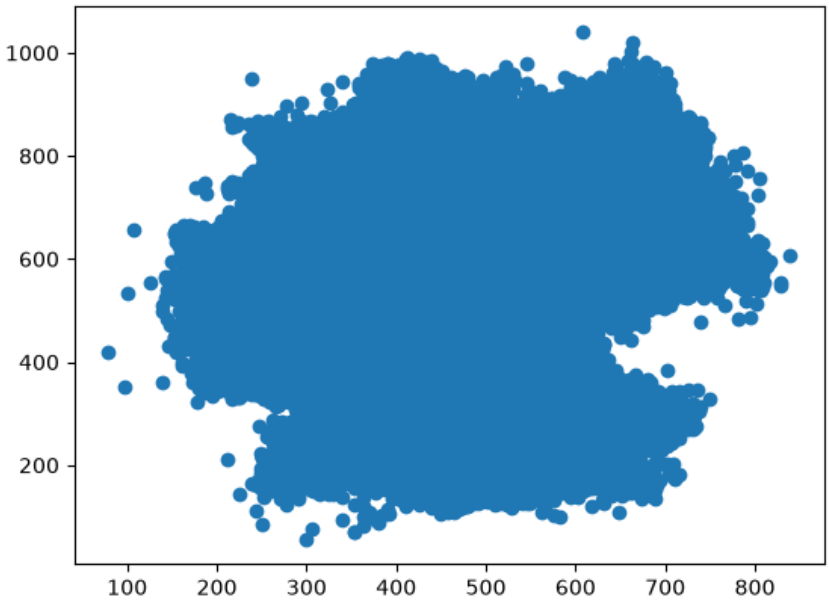


We see that in Düsseldorf, June and July have so far been the only months with heavy rainfall events.

Seven events are too few for looking at further correlations, so there is little to be gained from staying with Düsseldorf.

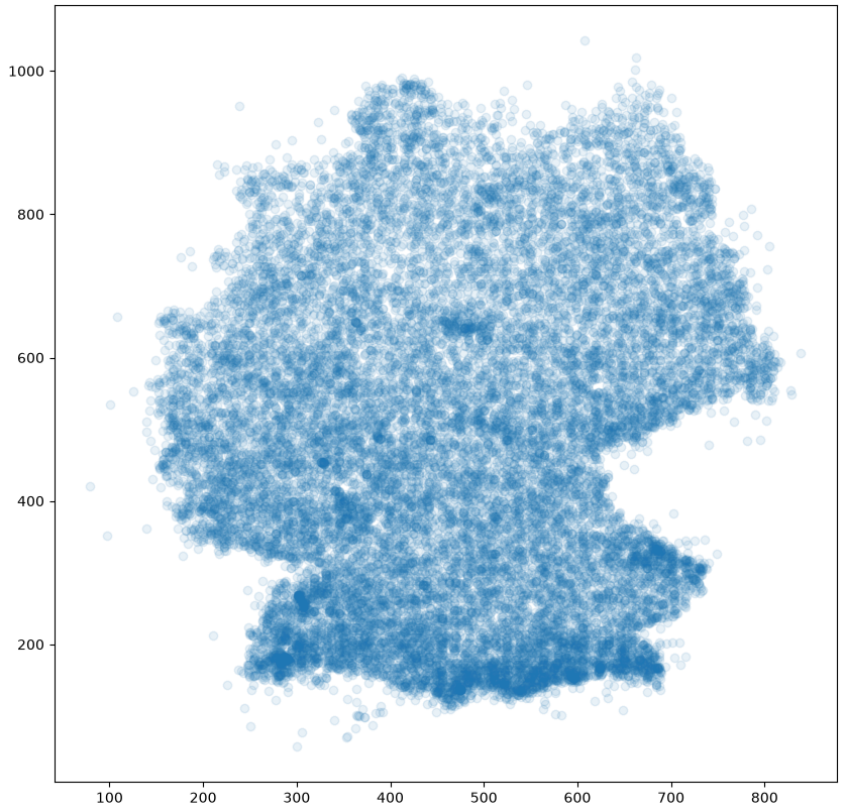
To get an impression of the spatial distribution, we look at a scatter plot of the locations of maximum precipitation in the RADKLIM grid (`x_RRmax, y_RRmax`):

```
plt.scatter(data["x_RRmax"], data["y_RRmax"])
plt.show()
```



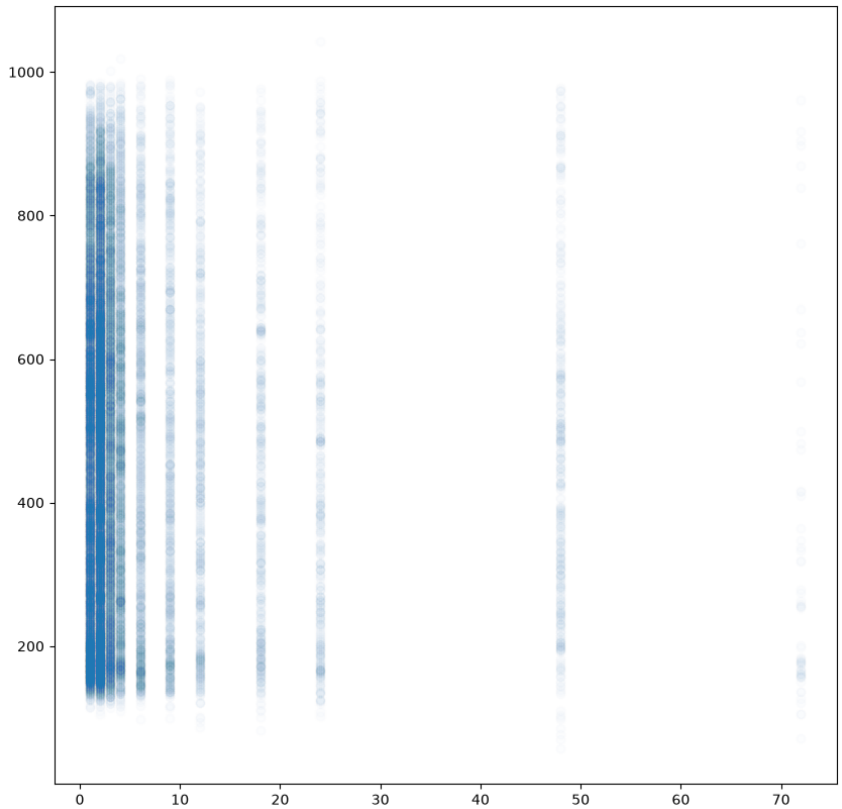
You can almost make out Germany – and, above all, that the y axis needs rescaling. So we read up on the scaling of the RADKLIM grid and learn that every grid point corresponds to exactly $1 \times 1\,\mathrm{km}^2$. The plot therefore merely has to be square. To see a little more of the distribution, we also make every point partly transparent. Large clusters then become visible:

```
plt.rcParams["figure.figsize"] = (10,10)
plt.scatter(data["x_RRmax"], data["y_RRmax"], alpha=0.1)
plt.show()
```



This shows that heavy rainfall events are more frequent in the south than in the north. We might also suspect a relationship with the duration (`Duration`) of the events:

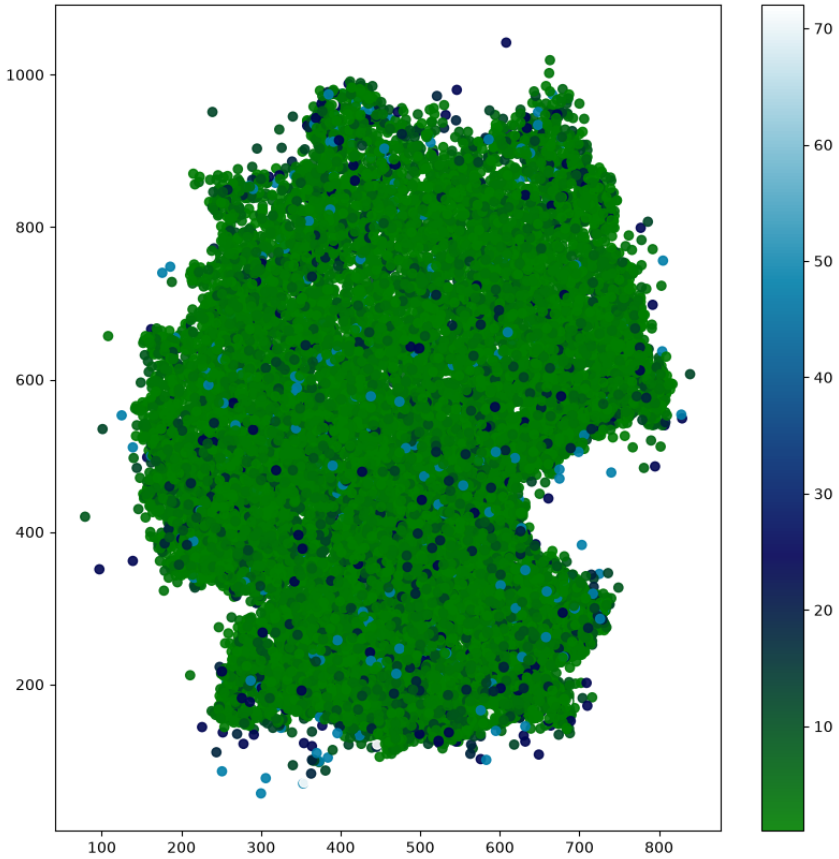
```
plt.scatter(data["Duration"], data["y_RRmax"], alpha=0.01)
plt.show()
```



Instead of the suspected relationship we see something else entirely: the duration is mostly a single-digit number, and longer events are rather rare. We can bring in the spatial relationship as well by

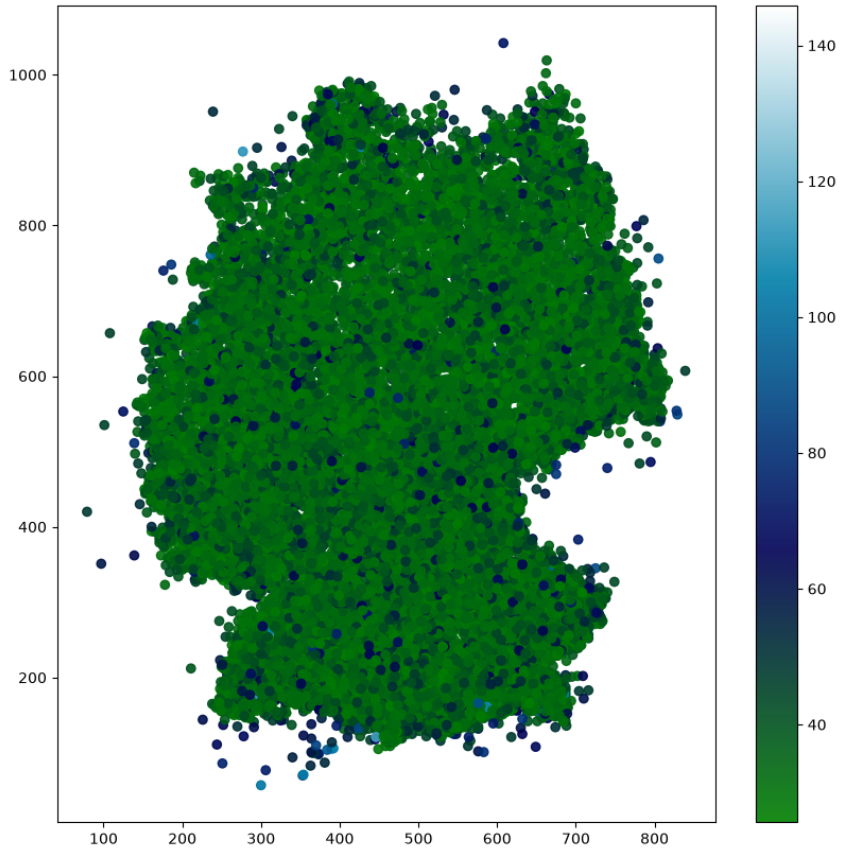
distinguishing by colour:

```
plt.scatter(data["x_RRmax"], data["y_RRmax"],
            alpha=0.9, c=data["Duration"],
            cmap="ocean")
plt.colorbar()
plt.show()
```



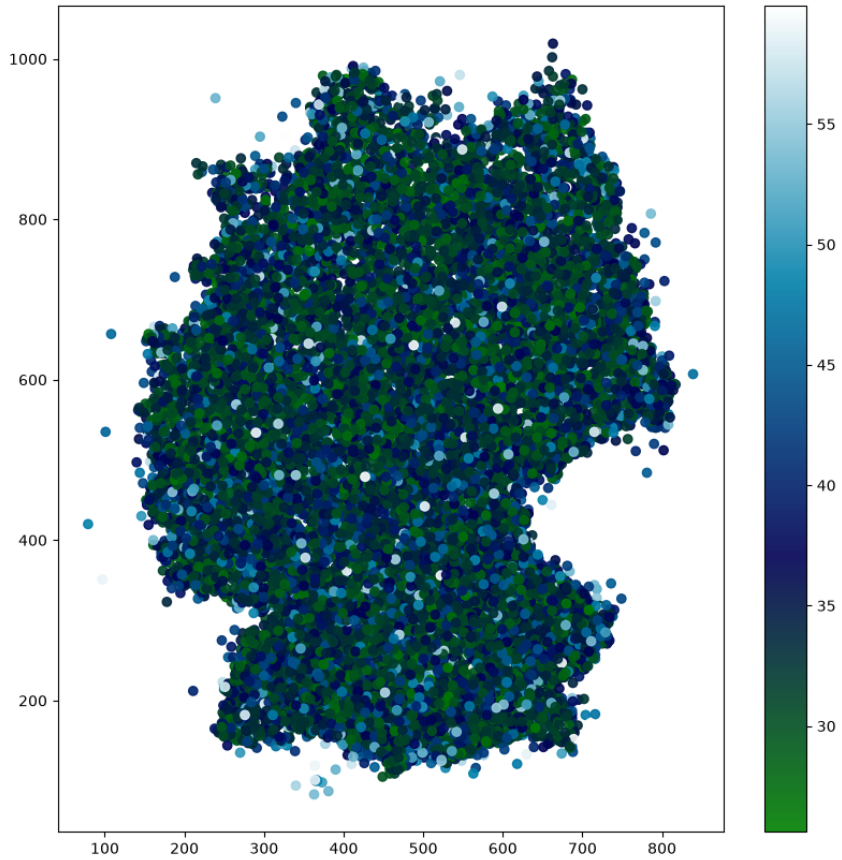
Since there is not much to see here (apart from the rarity of longer events), we look at the analogous plot for the mean precipitation:

```
plt.scatter(data["x_RRmax"], data["y_RRmax"],
            alpha=0.9, c=data["RRmean"],
            cmap="ocean")
plt.colorbar()
plt.show()
```



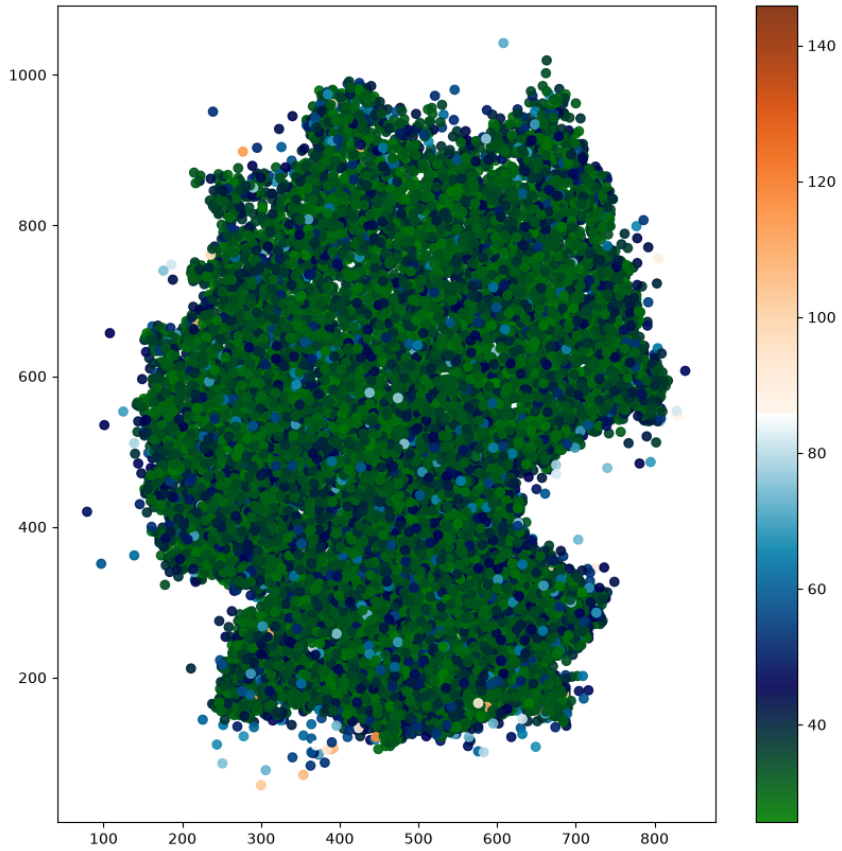
Here we begin to suspect that the scale is hard to read because the more extreme events (in light blue) take up so much of it. Excluding those events shows more:

```
calm_data = data[data["RRmean"] < 60]
plt.scatter(calm_data["x_RRmax"], calm_data["y_RRmax"],
            alpha=0.9, c=calm_data["RRmean"],
            cmap="ocean")
plt.colorbar()
plt.show()
```



But we get the same effect by adjusting the colormap instead:

```
from matplotlib import colormaps
from matplotlib.colors import ListedColormap
top = colormaps['ocean'].resampled(128)
bottom = colormaps['Oranges'].resampled(128)
newcolors = np.vstack((
    top(np.linspace(0, 1, 128)),
    bottom(np.linspace(0, 1, 128))
))
my_colormap = ListedColormap(newcolors, name='My Colormap')
plt.scatter(data["x_RRmax"], data["y_RRmax"],
            alpha=0.9, c=data["RRmean"],
            cmap=my_colormap)
plt.colorbar()
plt.show()
```



That will do on heavy rainfall events for now.

4.2.3 Titanic

Now we get to know matplotlib's object-oriented interface, using the Titanic dataset. On Kaggle it is part of a 'competition'.

We download the training dataset as `titanic.csv`.

```
# This will not work (pandas.read_csv understands quotation marks and
#  reads this file directly, see the chapter Pandas):
try:
    data = np.genfromtxt("titanic.csv",
                         delimiter=",", names=True)
except ValueError:
    print("Problem: ',' in the field 'Name'")

Problem: ',' in the field 'Name'

# We skip the first line and pretend
#   the name consisted of two columns.
data = np.genfromtxt("titanic.csv",
                     delimiter=",", skip_header=1,
                     usecols=(1,2,5,6,7,8),
                     dtype="i8,i8,S10,i8,i8,i8",
                     names=("Survived","Pclass","Sex",
                            "Age","SibSp","Parch"))
data[:4]

array([(0, 3, b'male', 22, 1, 0), (1, 1, b'female', 38, 1, 0),
       (1, 3, b'female', 26, 0, 0), (1, 1, b'female', 35, 1, 0)],
      dtype=[('Survived', '<i8'), ('Pclass', '<i8'), ('Sex', 'S10'), ('Age', '<i8'), ('SibSp',
    '<i8'), ('Parch', '<i8')])
```

```
print("Total survivors:", np.sum(data["Survived"]))
print("Total passengers:", len(data))
print("Survival rate:", np.sum(data["Survived"]) / len(data))
```

```
Total survivors: 342
Total passengers: 891
Survival rate: 0.3838383838383838
```

```
# Missing data:
data["Age"][:10]
```

```
array([22, 38, 26, 35, 35, -1, 54,  2, 27, 14])
```

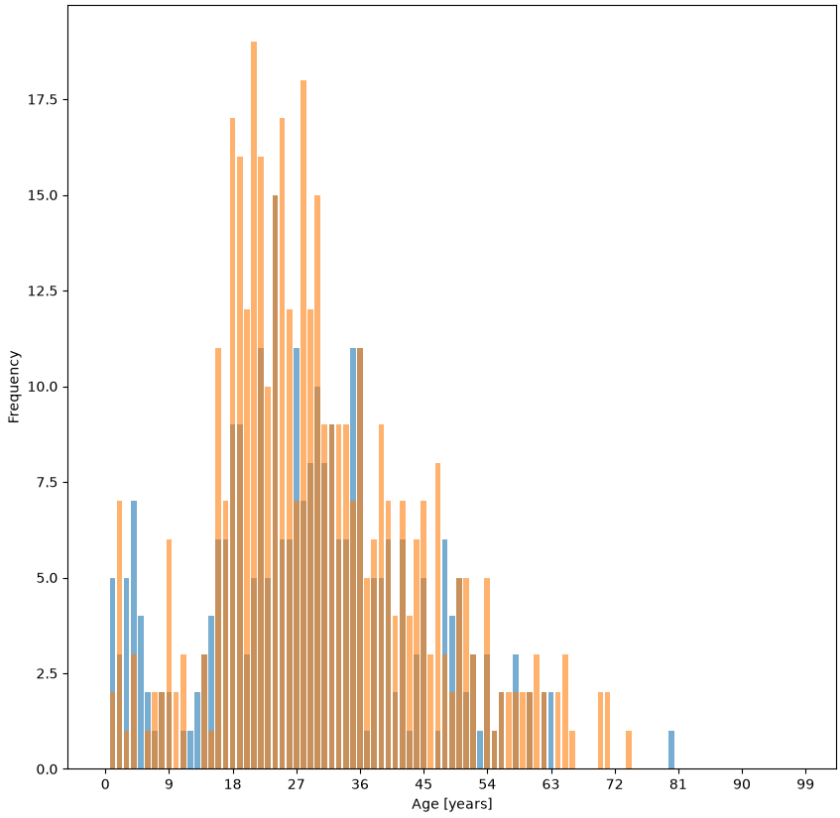
```
aged_data = data[data["Age"] != -1]
print("Wrong mean:", np.mean(data["Age"]))
print("Correct mean:", np.mean(aged_data["Age"]))
```

```
Wrong mean: 22.85409652076319
Correct mean: 29.847605224963715
```

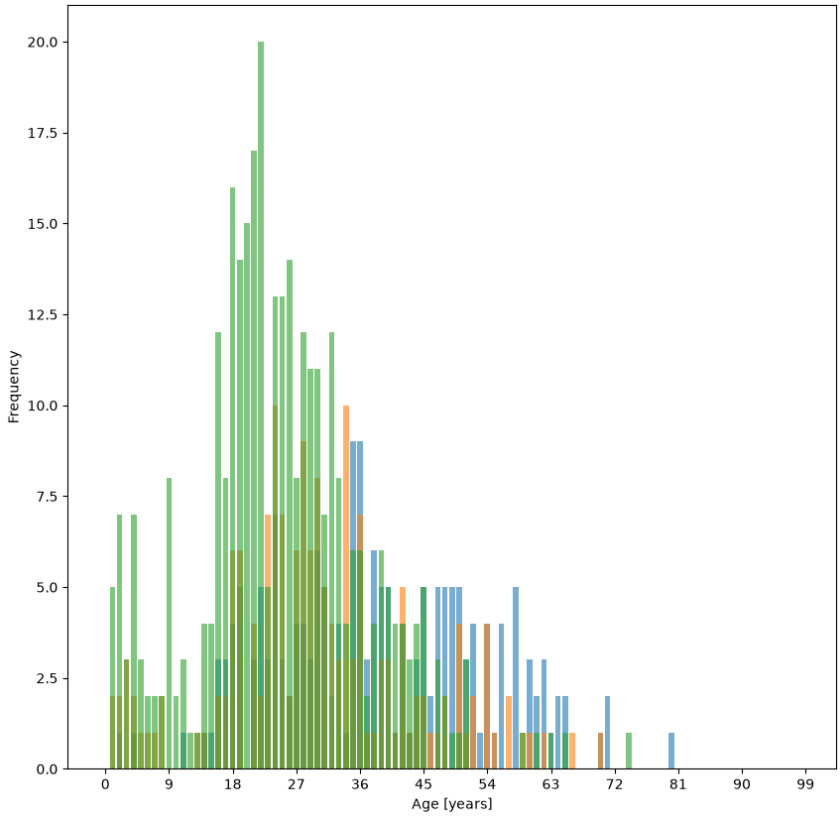
```
# Age correlated with survival?
nonsurvivors, survivors = tuple(aged_data[aged_data["Survived"] == i]
                                for i in range(0,2))
print(np.mean(survivors["Age"]), np.mean(nonsurvivors["Age"]))
```

```
29.01418439716312 30.425061425061426
```

```
fig, ax = plt.subplots()

def ageplot(data, ax):
    ax.set_xlabel("Age [years]")
    ax.set_ylabel("Frequency")
    ages = np.arange(0,99)
    count = np.array([len(data[data["Age"] == age])
                      for age in ages])
    ax.bar(ages, count, alpha=0.6)
    ax.set_xticks(np.linspace(0, 99, 12))

ageplot(survivors, ax)
ageplot(nonsurvivors, ax)
plt.show()
```



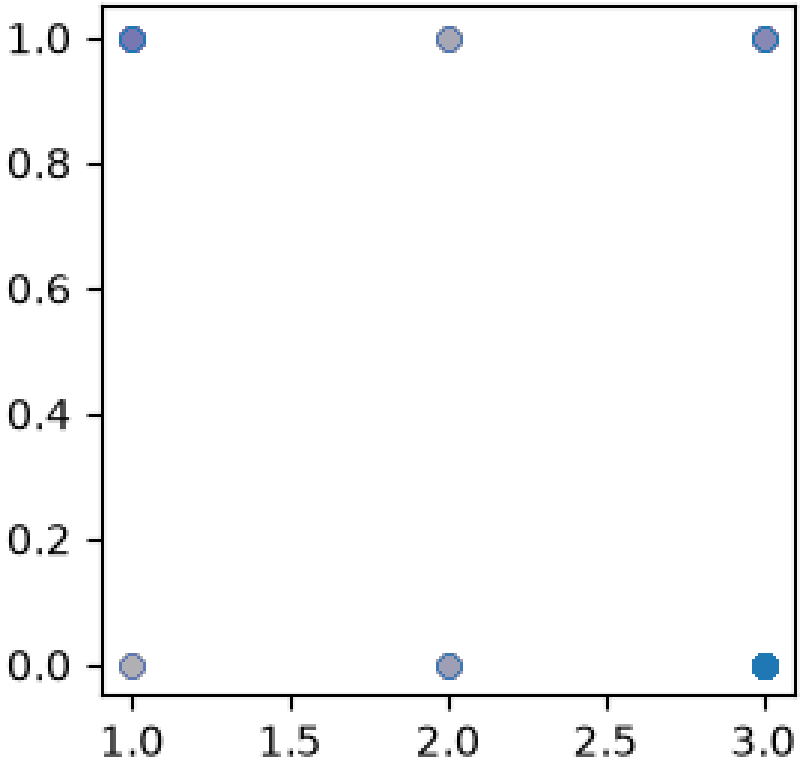

```
fig, ax = plt.subplots()
pclasses = (aged_data[aged_data["Pclass"] == i]
            for i in np.arange(1,4))
for pclass in pclasses:
    ageplot(pclass, ax)

plt.show()
```



It is hardly surprising that in class 3 (the cheapest, in green) the ages tend to be lower than in the other classes. In passing we also notice that class 3 is more frequent than the others (which should perhaps not surprise us either).

To see whether passenger class and survival are related, a scatter plot is enough:

```
fig, ax = plt.subplots(figsize=(3,3))
ax.scatter(data["Pclass"], data["Survived"], alpha=0.005)
plt.show()
```



The scatter plot shows a correlation: passengers in class 3 more often did not survive, and passengers who survived were more often in class 1. It was only visible, however, because we set `alpha = 0.005`, and it suffers from ignoring the absolute frequencies of the classes (a base rate fallacy).

The correct thing to do here is to compute the probability that a passenger in class 3 survived:

$$P(Survived = 1|Pclass = 3) = \frac{P(Survived = 1, Pclass = 3)}{P(Pclass = 3)} \tag{110}$$

```
psp_mask = [(data["Survived"] == 1) & (data["Pclass"] == c)
        for c in np.arange(1,4)]
p_mask = [(data["Pclass"] == c)
          for c in np.arange(1,4)]
total_pclass_counts = [len(data[p_mask[c]])
       for c in np.arange(0,3)]
psp = [len(data[psp_mask[c]]) / total_pclass_counts[c]
       for c in np.arange(0,3)]
for c in range(3):
    print("P( Survived = 1 | Pclass =",c+1,") =", psp[c])

P( Survived = 1 | Pclass = 1 ) = 0.6296296296296297
P( Survived = 1 | Pclass = 2 ) = 0.47282608695652173
P( Survived = 1 | Pclass = 3 ) = 0.24236252545824846
```

We see that a first-class passenger was considerably more likely to survive than to die. For second-class passengers the odds are already worse than 50/50, and fewer than a quarter of third-class passengers survived.

It should not surprise us that these probabilities do not sum to 1: they refer to different populations. For the reverse conditional probability, on the other hand, we do expect a sum of 1:

$$P(Pclass = 3|Survived = 1) = \frac{P(Survived = 1, Pclass = 3)}{P(Survived = 1)} \tag{111}$$

```
total_survivor_count = len(data[data["Survived"] == 1])
pps = [len(data[psp_mask[c]]) / total_survivor_count
       for c in np.arange(0,3)]
```

```
for c in range(3):
    print("P( Pclass =",c+1,"| Survived = 1 ) =", pps[c])

assert np.isclose(sum(pps), 1)   # never compare floating -point numbers exactly
```

```
P( Pclass = 1 | Survived = 1 ) = 0.39766081871345027
P( Pclass = 2 | Survived = 1 ) = 0.2543859649122807
P( Pclass = 3 | Survived = 1 ) = 0.347953216374269
```

This shows that survival status certainly does not tell us which class someone travelled in (because the passengers are not spread evenly over the classes). The two directions $P(\text{Survived} \mid \text{Pclass})$ and $P(\text{Pclass} \mid \text{Survived})$ are tied together by the class frequencies; exactly how is the business of Bayes' theorem in the chapter Covariance and dependence, and the chapter Titanic EDA works the same thing through for the variable `Sex`.

It is always interesting how representative a sample is, because that tells us how far the relationships we have found might generalise. So we now look at whether the variable `Sex` is correlated with other variables. Admittedly we do not yet know what we are doing, but let that be motivation for learning later what it was we did:

```
try:
    st.pearsonr(data["Sex"], data["Age"])
except ValueError as e:
    print(str(e))
```

```
could not convert string to float: np.bytes_(b'male')
```

```
sexbool = (data["Sex"] == b"female")
r, pvalue = st.pearsonr(sexbool, data["Age"])
print(r, pvalue)
```

```
-0.003542250865523883 0.9159102368544852
```

The second return value, `pvalue`, is a *p-value*: the probability, for two variables that really are uncorrelated, of seeing a correlation coefficient at least as far from 0 as ours. Here it is 0.92, so an r like this would be perfectly ordinary for uncorrelated data: the sample cannot tell the tiny deviation from 0 apart from sampling error. That is all the p-value says – in particular, it does not say how likely it is that sex and age *are* uncorrelated. What a p-value is and is not is settled in the chapter Statistical tests.

```
print(st.pearsonr(sexbool, data["Survived"]))
```

```
PearsonRResult(statistic=np.float64(0.5433513806577556), pvalue=np.float64(1.4060661308798955e
    -69))
```

Here we clearly see $r > 0.5$, and a vanishingly small probability p of observing such an r for data that are in fact uncorrelated.

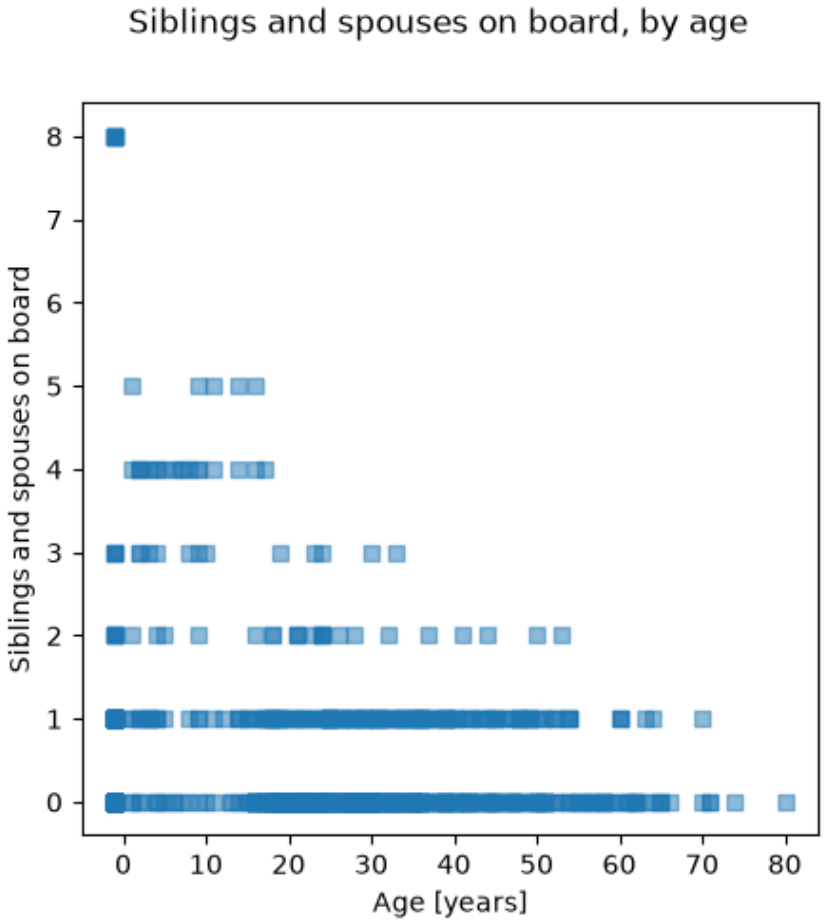
```
print(st.pearsonr(sexbool, data["Pclass"]))
print(st.pearsonr(sexbool, data["Age"]))
print(st.pearsonr(sexbool, data["SibSp"]))
print(st.pearsonr(sexbool, data["Parch"]))
```

```
PearsonRResult(statistic=np.float64( -0.1319004906047393), pvalue=np.float64(7.850524283810112e
    -05))
PearsonRResult(statistic=np.float64( -0.003542250865523883), pvalue=np.float64
    (0.9159102368544852))
PearsonRResult(statistic=np.float64(0.1146308103894224), pvalue=np.float64
    (0.0006076214735498901))
PearsonRResult(statistic=np.float64(0.24548896009069918), pvalue=np.float64(1.0739326016045421e
    -13))
```

So, naively (following the Scipy Stats documentation), we expect the variable `SibSp` to be correlated with sex. According to the metadata, `SibSp` is the number of siblings and spouses travelling with the passenger. Age, too, shows a correlation with `SibSp`:

```
print(st.pearsonr(data["Age"], data["SibSp"]))
fig, ax = plt.subplots(figsize=(5,5))
fig.suptitle("Siblings and spouses on board, by age")
ax.scatter(data["Age"], data["SibSp"],
           alpha=0.5, marker="s")
ax.set_xlabel("Age [years]")
ax.set_ylabel("Siblings and spouses on board")

plt.show()
```

```
PearsonRResult(statistic=np.float64( -0.16434928321340586), pvalue=np.float64(8.120291402857697e
    -07))
```



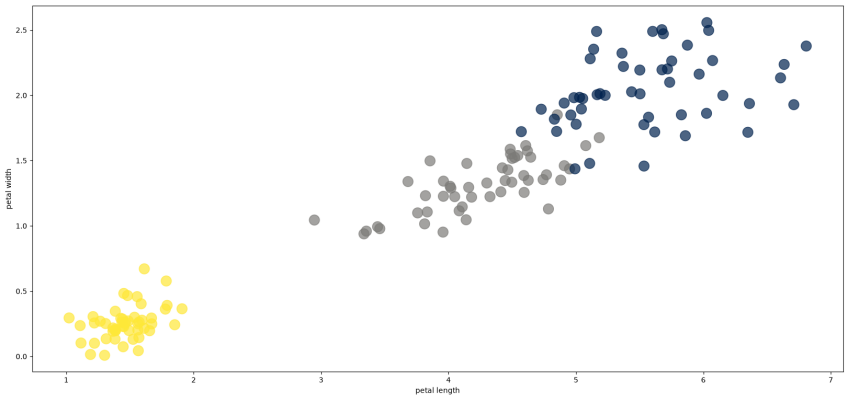
4.3 The Iris dataset: scatter plots and jittering

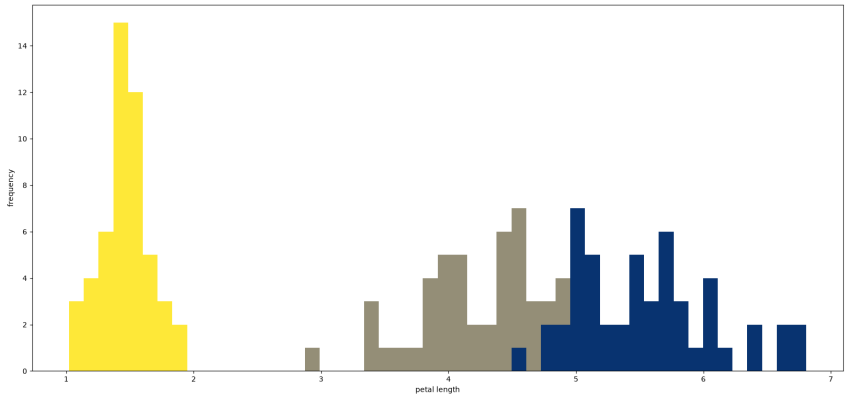
We will come back to the Iris dataset many times. For now we look at a plotting technique called "jittering", which makes it possible to show data in a scatter plot even when many points take the same value. It is an alternative to histogram techniques and to adjusting the transparency (the alpha value).

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

iris = datasets.load_iris()
x = iris.data
classes = iris.target

print(x[:5])
print(classes[:5])
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
[0 0 0 0 0]
```

```
def jitter(xs):
    """Add a little noise to the values so that equal ones do not cover each other up in the
    scatter plot"""
    b = np.diff(np.sort(xs))
    distance = b[b>0].min()
    return xs + distance * (2*np.random.random(len(xs)) - 1)

fig, (ax1,ax2) = plt.subplots(2, figsize=(20,20))

petlen, petwid = jitter(x[:, 2]), jitter(x[:, 3]) # jitter!
#petlen, petwid = x[:, 2], x[:, 3] # no jitter here, it looks like a grid <- - - - try the
    difference!
plmin, plmax = np.min(petlen), np.max(petlen)
cmap = plt.cm.cividis_r

ax1.scatter(petlen, petwid, c=classes, cmap=cmap, alpha=0.7, s=200)
for c in np.unique(classes):
    # spread classes 0, 1, 2 over the colormap; -0.1 keeps class 0 off the very light end
    ax2.hist(petlen[classes == c], range=(plmin,plmax), bins=50, color=cmap(c/2 -0.1))

ax2.set_xlabel("petal length")
ax2.set_ylabel("frequency")
ax1.set_xlabel("petal length")
ax1.set_ylabel("petal width")

plt.show()
```


petal width
petal length
0.0
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7


frequency
petal length
0
2
4
6
8
10
12
14
1
2
3
4
5
6
7

4.4 Pandas: DataFrames, Series and the way over from SQL

4.4.1 DataFrame and Series

The **DataFrame** is the data structure everything in Pandas revolves around: a table with named columns, each column with its own data type. If you know R, you know it as `data.frame`; if you have only used Numpy so far, think of it as a record array (`np.genfromtxt(..., names=True)` in the chapter Exploratory analysis produces one of those), only more convenient. The nearest thing to it is an Excel sheet in Python.

Unlike a Numpy array, a DataFrame is always two-dimensional – there is no such thing as a DataFrame with three or four axes. That is not a restriction but a commitment to exactly the shape data usually arrive in: rows are observations, columns are features.

```
import numpy as np
import pandas as pd

columns = {
    "species": ["setosa", "setosa", "versicolor"],
    "sepal_width": [3.5, 3.0, 3.2],
}
df = pd.DataFrame(columns)
print(df)
```

```
species  sepal_width
0      setosa          3.5
1      setosa          3.0
2  versicolor          3.2
```

There are two ways to get at a column, and they do the same thing: with square brackets, as for a dictionary, or as an attribute. The brackets always work; the attribute notation only works if the column name is a valid Python identifier and does not happen to shadow a method – so when in doubt, brackets.

```
print(df["sepal_width"])
print(df.sepal_width)  # the same thing

print(df.dtypes)  # here you can see that Numpy is underneath
print(df["sepal_width"].to_numpy())
```

```
0    3.5
1    3.0
2    3.2
Name: sepal_width, dtype: float64
0    3.5
1    3.0
2    3.2
Name: sepal_width, dtype: float64
species            str
sepal_width    float64
dtype: object
[3.5 3.  3.2]
```

A single column is not a DataFrame but the second of the two data structures Pandas has: a **Series**, that is, a one-dimensional array with an index. Most methods exist for both, some only for one of the two – the error message `'Series' object has no attribute ...` usually means you have accidentally got hold of a column instead of the table.

To disk and back is `to_csv` and `read_csv`; the latter also accepts a URL. The `index=False` when writing matters: without it the row index lands in the file as a column and comes back as `Unnamed: 0` when the file is read in again.

```
print(type(df["sepal_width"]))

df.to_csv("example.csv", index=False, encoding="utf -8")
back_again = pd.read_csv("example.csv")
print(back_again)
```

```
import os
os.remove("example.csv")

<class 'pandas.Series'>
      species  sepal_width
0      setosa          3.5
1      setosa          3.0
2  versicolor          3.2
```

Pandas has some of the best documentation in the Python ecosystem, and it is worth using instead of guessing: the tutorials for getting started, the API reference with a page per method (`pandas.DataFrame.groupby`, say) and the overview of the data types.

That last one is worth a look because the Pandas type system goes beyond the Numpy types. Besides `int64`, `float64` and `bool` there are, among others:

Kind of data	pandas data type	Scalar
point in time with time zone	`DatetimeTZDtype`	`Timestamp`
duration	–	`Timedelta`
period	`PeriodDtype`	`Period`
interval	`IntervalDtype`	`Interval`
integer with missing values	`Int64Dtype`, …	–
categorical values	`CategoricalDtype`	–
sparse column	`SparseDtype`	–
strings	`StringDtype`	`str`
truth values with missing values	`BooleanDtype`	–

Two of them will turn up again: `CategoricalDtype` in one-hot encoding in Regression analysis and the timestamps in Data cleaning. That the column `species` above has type `str` rather than `object` is new – up to pandas 2, strings lived in an `object` column, which you will still see often in older code and older tutorials.

4.4.2 SQL and relational databases

Anything that can be expressed in SQL can be expressed with DataFrames, and the rest of this chapter puts the two side by side. A few words on where that language fits are in order first.

SQL is the standard query language for relational database systems (RDBMS) and has been an ISO standard since 1987. The two most widespread free systems are MySQL and PostgreSQL; they sit behind a large share of the smaller websites – every WordPress installation comes with a database, and WordPress powers something like 40% of all websites. Anyone pulling data out of a live system almost always writes SQL first and only then carries on in Python.

The most important commands, grouped by what they touch:

Command	Effect
`SELECT`	select data and output it
`INSERT INTO`	add rows
`UPDATE`	change existing values
`DELETE`	delete rows
`CREATE` / `ALTER DATABASE`	create, change a database
`CREATE` / `ALTER` / `DROP TABLE`	create, change, delete a table
`CREATE` / `DROP INDEX`	create, delete a search key

A query then looks like this, for example:

```
SELECT "Petal.Width" FROM iris;
```

The vocabulary of the relational world is worth knowing, because you meet it constantly in documentation and error messages:

Definition
A **database schema** is a finite, non-empty set of **relation schemas**; a relation schema is the formal description of the data types of a **relation**. A relation is what you see as a **table**: its rows are called **tuples** or *entities*, its columns are called **attributes**, and the values permitted for an attribute form its **domain**.

Translated into the Pandas world:

relational	Pandas
relation / table	`DataFrame`
attribute / column	`Series`
tuple / row	a row, addressed via the index
domain	the `dtype` of the column

One difference remains: mathematically, a relation is a *set* of tuples and so has no order, whereas a DataFrame keeps its rows in order and addresses them through an index. That is why SQL needs an `ORDER BY` to get any order at all – and why Pandas does not guarantee the row order after a `merge`.

4.4.3 From SQL to Pandas

The comparison below follows the Pandas documentation, section "Comparison with SQL". In each code cell, the SQL statement appears as a comment above the Pandas expression that does the same job.

```
import numpy as np
import pandas as pd

url = (
    "https://raw.github.com/pandas -dev"
    "/pandas/main/pandas/tests/io/data/csv/tips.csv"
)

tips = pd.read_csv(url)
print(tips)
```

```
total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Operations on Pandas DataFrames produce copies; the original DataFrame stays unchanged.

```
tips.sort_values("total_bill")
print(tips)
print("\nThat is not sorted at all! But the next one is:\n")
sorted_tips = tips.sort_values("total_bill")
print(sorted_tips)
```

```
total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
```

```
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]

That is not sorted at all! But the next one is:

     total_bill    tip     sex smoker   day    time  size
67         3.07   1.00  Female    Yes   Sat  Dinner     1
92         5.75   1.00  Female    Yes   Fri  Dinner     2
111        7.25   1.00  Female     No   Sat  Dinner     1
172        7.25   5.15    Male    Yes   Sun  Dinner     2
149        7.51   2.00    Male     No  Thur   Lunch     2
..          ...    ...     ...    ...   ...     ...   ...
182       45.35   3.50    Male    Yes   Sun  Dinner     3
156       48.17   5.00    Male     No   Sun  Dinner     6
59        48.27   6.73    Male     No   Sat  Dinner     4
212       48.33   9.00    Male     No   Sat  Dinner     4
170       50.81  10.00    Male    Yes   Sat  Dinner     3

[244 rows x 7 columns]
```

Selecting columns is SELECT:

```
# SELECT total_bill, tip, smoker, time FROM tips;
tips[["total_bill", "tip", "smoker", "time"]]
```

	total_bill	tip	smoker	time
0	16.99	1.01	No	Dinner
1	10.34	1.66	No	Dinner
2	21.01	3.50	No	Dinner
3	23.68	3.31	No	Dinner
4	24.59	3.61	No	Dinner
...	...	...	...	...
239	29.03	5.92	No	Dinner
240	27.18	2.00	Yes	Dinner
241	22.67	2.00	Yes	Dinner
242	17.82	1.75	No	Dinner
243	18.78	3.00	No	Dinner

244 rows × 4 columns

You can add new columns computed from old ones:

```
# SELECT *, tip/total_bill as tip_rate FROM tips;
tips.assign(tip_rate=tips["tip"] / tips["total_bill"])
```

	total_bill	tip	sex	smoker	day	time	size	tip_rate
0	16.99	1.01	Female	No	Sun	Dinner	2	0.059447
1	10.34	1.66	Male	No	Sun	Dinner	3	0.160542
2	21.01	3.50	Male	No	Sun	Dinner	3	0.166587
3	23.68	3.31	Male	No	Sun	Dinner	2	0.139780
4	24.59	3.61	Female	No	Sun	Dinner	4	0.146808
...	...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3	0.203927
240	27.18	2.00	Female	Yes	Sat	Dinner	2	0.073584
241	22.67	2.00	Male	Yes	Sat	Dinner	2	0.088222
242	17.82	1.75	Male	No	Sat	Dinner	2	0.098204
243	18.78	3.00	Female	No	Thur	Dinner	2	0.159744

244 rows × 8 columns

Filtering data (selecting rows by conditions) with `WHERE`

```
# SELECT * FROM tips WHERE time = 'Dinner';
is_dinner = tips["time"] == "Dinner" # bitmask
tips[is_dinner]
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

176 rows × 7 columns

Combining several conditions logically with `OR |` or `AND &`:

```
# SELECT * FROM tips WHERE time = 'Dinner' AND tip > 6.00;
tips[(tips["time"] == "Dinner") & (tips["tip"] > 6.00)]
```

	total_bill	tip	sex	smoker	day	time	size
23	39.42	7.58	Male	No	Sat	Dinner	4
59	48.27	6.73	Male	No	Sat	Dinner	4
170	50.81	10.00	Male	Yes	Sat	Dinner	3
183	23.17	6.50	Male	Yes	Sun	Dinner	4
212	48.33	9.00	Male	No	Sat	Dinner	4
214	28.17	6.50	Female	Yes	Sat	Dinner	3

```
# SELECT * FROM tips WHERE size >= 6 OR total_bill > 45;
tips[(tips["size"] >= 6) | (tips["total_bill"] > 45)]
```

	total_bill	tip	sex	smoker	day	time	size
59	48.27	6.73	Male	No	Sat	Dinner	4
125	29.80	4.20	Female	No	Thur	Lunch	6
141	34.30	6.70	Male	No	Thur	Lunch	6
143	27.05	5.00	Female	No	Thur	Lunch	6
156	48.17	5.00	Male	No	Sun	Dinner	6
170	50.81	10.00	Male	Yes	Sat	Dinner	3
182	45.35	3.50	Male	Yes	Sun	Dinner	3
212	48.33	9.00	Male	No	Sat	Dinner	4

Split the data into groups, compute aggregate statistics, put the groups back together – that is `GROUP BY` in SQL and `groupby` in Pandas:

```
# SELECT sex, count(*) FROM tips GROUP BY sex;
tips.groupby("sex").size()
# tips.groupby("sex").count() would be wrong, that counts non -NA entries!

sex
Female     87
Male      157
dtype: int64

# SELECT day, AVG(tip), COUNT(*) FROM tips GROUP BY day;
tips.groupby("day").agg({"tip": "mean", "day": "size"})
```

	tip	day
day		
Fri	2.734737	19
Sat	2.993103	87
Sun	3.255132	76
Thur	2.771452	62

A trap that snaps shut again and again when you aggregate: **the mean of the group means is not the overall mean**, unless all the groups are the same size. The overall mean is the *weighted* mean of the group means, with the group sizes as weights. With the values $10, 20, 30$ in group A and $5, 15$ in group B, the group means are 20 and 10 and their mean is 15 – but the overall mean of all five values is 16. So if all you have is a table of group means and you want the overall mean from it, you need the `size` column as well. The same phenomenon in a sharper form – group means whose order reverses when the groups are pooled – is called *Simpson's paradox*.

```
small = pd.DataFrame({"Group": ["A", "A", "A", "B", "B"], "Value": [10, 20, 30, 5, 15]})
group_means = small.groupby("Group")["Value"].mean()
print("mean of the group means:", group_means.mean())
print("overall mean:           ", small["Value"].mean())
# correct: weight by the group sizes
sizes = small.groupby("Group")["Value"].size()
print("weighted:               ", (group_means * sizes).sum() / sizes.sum())

mean of the group means: 15.0
overall mean:            16.0
weighted:                16.0

# SELECT smoker, day, COUNT(*), AVG(tip) FROM tips GROUP BY smoker, day;
tips.groupby(["smoker", "day"]).agg({"tip": ["size", "mean"]})
```

NaNtip

		size
mean		
smoker	day	
No	Fri	4
2.812500		
	Sat	45
3.102889		
	Sun	57
3.167895		
	Thur	45
2.673778		
Yes	Fri	15
2.714000		
	Sat	42
2.875476		
	Sun	19
3.516842		
	Thur	17
3.030000		

Data rarely arrive in a single table. Suppose the Iris measurements sit in two files: one holds the species of each plant, the other its sepal width. Both have the same number of rows – but may we simply lay them side by side? Only if the rows are in the same order, and you cannot know that once anyone has sorted one of the files in between.

So every row gets a column that identifies it uniquely: a **key**. That can be a plain number, or some unique piece of information that already exists (a student number, an ISBN). When data come together from several sources and nobody can hand out numbers centrally, you use a **UUID** (*universally unique identifier*), a 128-bit random number for which a collision is so unlikely that it can be ignored. The column that uniquely indexes a table is called its **primary key**; when the same key turns up in another table to point at those rows, it is a foreign key there. That is the basic idea of relational databases. The chapter Relational data of *R for Data Science* has a fuller account.

```
import uuid

species = pd.DataFrame({"No": [1, 51], "species": ["setosa", "versicolor"]})
width = pd.DataFrame({"No": [51, 1], "sepal_width": [3.2, 3.5]})
print(species, "\n")
print(width, "\n")

# merged on the key No, so the order does not matter:
print(pd.merge(species, width, on="No"), "\n")

print(uuid.uuid4())  # a fresh UUID
```

```
No     species
0   1      setosa
1  51  versicolor

   No  sepal_width
0  51          3.2
1   1          3.5

   No     species  sepal_width
0   1      setosa          3.5
1  51  versicolor          3.2

2c741528 -2614 -4361 -a4e1 -b190d6217e53
```

It gets interesting when the two tables do not have the same keys. Then you have to decide what happens to the rows that find no partner – and that is exactly what separates the four kinds of join. The two tables below are built so that every case occurs: `A` exists only on the left, `E` only on the right, and `D` appears twice on the right.

```
df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})
df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})
```

```
print(df1, "\n", df2)
```

```
key     value
0   A  0.370613
1   B  1.987784
2   C -0.885252
3   D  0.673109
   key     value
0   B -1.594238
1   D  1.483229
2   D -0.353315
3   E  0.321249
```

```
# SELECT * FROM df1 INNER JOIN df2 ON df1.key = df2.key;
pd.merge(df1, df2, on="key") # inner join is the default
```

	key	value_x	value_y
0	B	1.987784	-1.594238
1	D	0.673109	1.483229
2	D	0.673109	-0.353315

The **inner join** keeps only the keys that occur in both tables: `B` and `D`. That `D` yields two rows is the most important point in the whole business – the key is not unique in `df2`, and a join then multiplies the rows on the other side. If a dataset is suddenly longer after a join than before, this is almost always where the problem lies.

```
# SELECT * FROM df1 LEFT OUTER JOIN df2 ON df1.key = df2.key;
pd.merge(df1, df2, on="key", how="left")
```

	key	value_x	value_y
0	A	0.370613	NaN
1	B	1.987784	-1.594238
2	C	-0.885252	NaN
3	D	0.673109	1.483229
4	D	0.673109	-0.353315

The **left outer join** keeps every row of the left table, partner or not: `A` and `C` come along too, with `NaN` in the columns from `df2`. This is the commonest case in practice – you have a dataset and want to enrich it with extra information without losing any rows along the way.

```
# SELECT * FROM df1 RIGHT OUTER JOIN df2 ON df1.key = df2.key;
pd.merge(df1, df2, on="key", how="right")
```

	key	value_x	value_y
0	B	1.987784	-1.594238
1	D	0.673109	1.483229
2	D	0.673109	-0.353315
3	E	NaN	0.321249

The **right outer join** is the same with the roles swapped: every row of `df2`, `E` included, and `NaN` where the left-hand side is missing.

```
# SELECT * FROM df1 FULL OUTER JOIN df2 ON df1.key = df2.key;
pd.merge(df1, df2, on="key", how="outer")
```

	key	value_x	value_y
0	A	0.370613	NaN
1	B	1.987784	-1.594238
2	C	-0.885252	NaN
3	D	0.673109	1.483229
4	D	0.673109	-0.353315
5	E	NaN	0.321249

The **full outer join** keeps everything, `A` to `E`, and pads both sides with `NaN` wherever nothing matches. Afterwards it is nearly always worth a look at `isna().sum()` to see how many gaps you have let yourself in for.

Changing data is `UPDATE` in SQL; in Pandas we simply run Python code, as if we were changing a single variable:

```
# state before the change:
tips.sort_values("tip").head()
```

	total_bill	tip	sex	smoker	day	time	size
92	5.75	1.00	Female	Yes	Fri	Dinner	2
111	7.25	1.00	Female	No	Sat	Dinner	1
67	3.07	1.00	Female	Yes	Sat	Dinner	1
236	12.60	1.00	Male	Yes	Sat	Dinner	2
0	16.99	1.01	Female	No	Sun	Dinner	2

```
# UPDATE tips SET tip = tip*2 WHERE tip < 2;
tips.loc[tips["tip"] < 2, "tip"] *= 2
tips.sort_values("tip").head()
```

	total_bill	tip	sex	smoker	day	time	size
6	8.77	2.0	Male	No	Sun	Dinner	2
26	13.37	2.0	Male	No	Sat	Dinner	2
27	12.69	2.0	Male	No	Sat	Dinner	2
36	16.31	2.0	Male	No	Sat	Dinner	3
61	13.81	2.0	Male	Yes	Sat	Dinner	2

```
# state before the next change:
tips.sort_values("tip", ascending=False).head()
```

	total_bill	tip	sex	smoker	day	time	size
170	50.81	10.00	Male	Yes	Sat	Dinner	3
212	48.33	9.00	Male	No	Sat	Dinner	4
23	39.42	7.58	Male	No	Sat	Dinner	4
59	48.27	6.73	Male	No	Sat	Dinner	4
141	34.30	6.70	Male	No	Thur	Lunch	6

```
# DELETE FROM tips WHERE tip > 9;
tips = tips.loc[tips["tip"] <= 9]
tips.sort_values("tip", ascending=False).head()
```

	total_bill	tip	sex	smoker	day	time	size
212	48.33	9.00	Male	No	Sat	Dinner	4
23	39.42	7.58	Male	No	Sat	Dinner	4
59	48.27	6.73	Male	No	Sat	Dinner	4
141	34.30	6.70	Male	No	Thur	Lunch	6
183	23.17	6.50	Male	Yes	Sun	Dinner	4

4.4.4 A first look at a dataset

Pandas comes with the tools we discussed in the chapter on descriptive statistics – the ones that were still handwork with Numpy in the Titanic notebook. The 'broken' CSV from there is read directly by `read_csv`, which understands quotation marks. `info` shows the columns, the data types and how many values are missing: compare this with the Titanic notebook from the chapter on Numpy, where we converted the CSV into a TSV by hand, marked missing values with masks and counted groups with loops.

```
titanic = pd.read_csv("titanic.csv")
titanic.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non -Null Count  Dtype
 - - -  - - - - - -       - - - - - - - - - - - - - - -  - - - - -
 0   PassengerId  891 non -null    int64
 1   Survived     891 non -null    int64
 2   Pclass       891 non -null    int64
 3   Name         891 non -null    str
 4   Sex          891 non -null    str
 5   Age          714 non -null    float64
 6   SibSp        891 non -null    int64
 7   Parch        891 non -null    int64
 8   Ticket       891 non -null    str
 9   Fare         891 non -null    float64
 10  Cabin        204 non -null    str
 11  Embarked     889 non -null    str
dtypes: float64(2), int64(5), str(5)
memory usage: 83.7 KB
```

For every numerical column, `describe` gives the count, the mean, the standard deviation (with $n-1$ in the denominator!), the minimum, the quartiles and the maximum:

```
titanic.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

For categorical columns, `value_counts` counts the levels:

```
print(titanic["Sex"].value_counts())
print(titanic["Embarked"].value_counts(dropna=False))   # count the missing values too
```

```
Sex
male      577
female    314
Name: count, dtype: int64
Embarked
S      644
C      168
Q       77
NaN      2
Name: count, dtype: int64
```

`isna` spots missing values. You can discard them (`dropna`) or replace them (`fillna`), with the median for instance. How to deal systematically with missing values, outliers, text fields and dates is the subject of the chapter Data cleaning.

Skewness and kurtosis from the chapter on expected values are built in as well; `kurt` gives the excess kurtosis, that is, the kurtosis minus 3:

```
print(titanic.isna().sum())

age = titanic["Age"].fillna(titanic["Age"].median())
print("skewness:", age.skew(), " excess kurtosis:", age.kurt())
```

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
skewness: 0.5102446555756495  excess kurtosis: 0.9938710163801736
```

Grouped summary measures come from `groupby`, as above:

```
titanic.groupby("Pclass")["Age"].describe()
```

	count	mean	std	min	25%	50%	75%	max
Pclass								
1	186.0	38.233441	14.802856	0.92	27.0	37.0	49.0	80.0
2	173.0	29.877630	14.001077	0.67	23.0	29.0	36.0	70.0
3	355.0	25.140620	12.495398	0.42	18.0	24.0	32.0	74.0

4.5 Data cleaning: outliers, missing values, data types

The first chapter said that most of a data scientist's working day goes not on modelling but on obtaining data, cleaning it and communicating results. This chapter is about the cleaning – "data scrubbing": spotting outliers, handling missing values, repairing data types, taking text apart with regular expressions, reading dates and times correctly. None of it is mathematically deep, and all of it decides whether the models of the chapters that follow are any good.

Principle

Every act of cleaning distorts the data. Removing an outlier, filling in a missing value or normalising a string changes the sample. That is acceptable as long as three rules hold:

1. **Log it.** Every change goes into code, where it can be reproduced – ideally with a column that marks which values were changed. Never edit by hand, in the CSV file or in Excel.
2. **Look again afterwards.** After every step, plot the distribution again and check that it still looks plausible.
3. **Measure against the target metric.** Whether you fill in missing values or throw the rows away is decided not by gut feeling but by the metric you will later use to measure the success of the whole pipeline. If you cannot measure it, choose whichever option changes the data least.

The example data are daily values from the Düsseldorf weather station, taken from the open data portal of the German weather service (DWD), and, at the end, the Titanic dataset for the text processing.

4.5.1 Loading the data and a first look

For each station the DWD supplies a ZIP archive containing one data file and several metadata files. Missing values are coded as `-999` – a fact stated in the file's description, not in the file itself. Without that piece of information Pandas would read `-999` as a valid temperature, and the mean would be nonsense.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import zipfile, urllib.request, pathlib

base = "https://opendata.dwd.de/climate_environment/CDC/observations_germany/climate/daily/kl/
    historical/"
station = "01078"                                    # Düsseldorf
folder = pathlib.Path(base.replace("https://", ""))  # local copy, same path as on the web

# The file name contains the end date and changes every year, as soon as the DWD appends
# another year. So we search for it: locally first, otherwise in the server's directory listing.
local = sorted(folder.glob(f"tageswerte_KL_{station}_*_hist.zip"))
if local:
    path = local[ -1]
else:
    import re
    listing = urllib.request.urlopen(base).read().decode("utf -8")
    filename = re.search(rf"tageswerte_KL_{station}_\d{{8}}_\d{{8}}_hist\.zip", listing).group
    (0)
    path = folder / filename
    folder.mkdir(parents=True, exist_ok=True)
    urllib.request.urlretrieve(base + path.name, path)

with zipfile.ZipFile(path) as archive:
    data_file = [name for name in archive.namelist() if name.startswith("produkt_klima_tag")][0]
    raw = pd.read_csv(archive.open(data_file), sep=";", skipinitialspace=True, na_values= -999)

raw.columns = raw.columns.str.strip()                # column names have leading spaces
raw = raw.drop(columns="eor")                        # "end of record", no content
```

```
print(raw.shape)
raw.tail(3)
```

```
(27029, 18)
```

	STA	MO	ESS	QN_3	DAT	TXK	TMK	FM	QN_4	RSK	RSKF	SDK	SHK_TAG	NM	VPM	PM	TMK	UPM	TXK	TNK	TGK
27026	1078	202512	29	5.0	1.4	9.0	0.5	6.0	NaN	0.0	NaN	6.1	1021.1	0.0	97.0	3.8	-5.8	-8.8
27027	1078	202512	30	6.0	1.6	9.0	0.0	0.0	NaN	0.0	1.5	5.5	1023.6	0.2	89.0	4.8	-5.1	-8.1
27028	1078	202512	31	10.5	3.5	9.0	0.2	6.0	NaN	0.0	6.4	5.9	1019.4	0.2	93.0	3.6	-4.9	-8.6

The column names are shorthand codes explained in the metadata file: `TMK` is the daily mean temperature, `TXK` and `TNK` the maximum and the minimum, `RSK` the amount of precipitation in mm, `SDK` the duration of sunshine in hours, `PM` the air pressure, `UPM` the relative humidity. We keep the columns we need, give them telling names and restrict ourselves to 2025. The date stays an integer of the form `YYYYMMDD` for now; the section on date and time deals with it.

```
weather = (raw.rename(columns={"MESS_DATUM": "date_raw", "TMK": "temp_mean", "TXK": "temp_max",
                              "TNK": "temp_min", "RSK": "precipitation", "SDK": "sunshine", "PM
    ": "pressure", "UPM": "humidity"})
              .loc[:, ["date_raw", "temp_mean", "temp_max", "temp_min", "precipitation", "
    sunshine", "pressure", "humidity"]])
weather = weather[(weather["date_raw"] >= 20250101) & (weather["date_raw"] <= 20251231)].
    reset_index(drop=True)
weather.info()
print("\nMissing values per column:")
print(weather.isna().sum())
```

```
<class 'pandas.DataFrame'>
RangeIndex: 365 entries, 0 to 364
Data columns (total 8 columns):
 #   Column         Non -Null Count  Dtype
 - - -  - - - - - -         - - - - - - - - - - - - - - - -  - - - - -
 0   date_raw       365 non -null    int64
 1   temp_mean      365 non -null    float64
 2   temp_max       365 non -null    float64
 3   temp_min       365 non -null    float64
 4   precipitation  364 non -null    float64
 5   sunshine       21 non -null     float64
 6   pressure       355 non -null    float64
 7   humidity       346 non -null    float64
dtypes: float64(7), int64(1)
memory usage: 22.9 KB

Missing values per column:
date_raw           0
temp_mean          0
temp_max           0
temp_min           0
precipitation      1
sunshine         344
pressure          10
humidity          19
dtype: int64
```

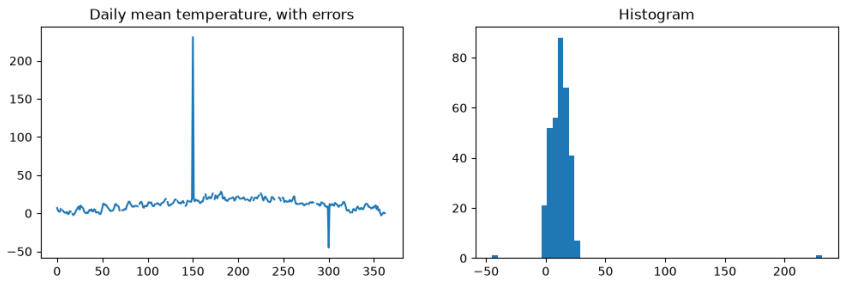
The duration of sunshine is missing on almost every day of 2025; air pressure and humidity have the odd gap. The data file does not say why; the answer is in the archive's metadata files, which is exactly why those have to be read too. These are genuine gaps, of the kind every dataset has. To *assess* how gaps are handled, though, we need gaps whose true value we know. So we make a copy and deliberately delete a random 10% of its daily mean temperatures. That is rule 3 in action: only with a known truth can we measure which repair is best.

```
rng = np.random.default_rng(2025)
truth = weather["temp_mean"].copy()
deleted = rng.random(len(weather)) < 0.10              # mask: which values we "lose"
broken = weather.copy()
broken.loc[deleted, "temp_mean"] = np.nan
broken["temp_mean_missing"] = deleted                 # rule 1: the change is logged as a
    column
print(f"{deleted.sum()} of {len(weather)} daily means removed")

30 of 365 daily means removed
```

4.5.2 Outliers

An outlier is a value so far from the others that it is either a measurement error or a rare event that skews the picture. The number itself will not tell you which; that has to be decided from the context. The first step is always to look. To have something to look at, we build in two errors typical of transcribed measurements: a temperature without its decimal point, and a sign error.

```
faulty = broken.copy()
faulty.loc[150, "temp_mean"] = 231.0      # 23.1 typed without the decimal point
faulty.loc[300, "temp_mean"] = -45.0      # sign and digit transposed

fig, axs = plt.subplots(1, 2, figsize=(12, 3.5))
axs[0].plot(faulty["temp_mean"].values); axs[0].set_title("Daily mean temperature, with errors")
axs[1].hist(faulty["temp_mean"].dropna(), bins=60); axs[1].set_title("Histogram")
plt.show()
print("Mean with errors:", faulty["temp_mean"].mean().round(2),
      " without errors:", broken["temp_mean"].mean().round(2))
```




```
Mean with errors: 12.21  without errors: 11.75
```

In the plot both errors are obvious at once; in the mean they hide behind half a degree. The systematic way to find such values is the **z-score**, the deviation from the mean in units of the standard deviation (see Expected value and variance). The usual rule of thumb: values with $|z| > 3$ count as outliers. Under a normal distribution only one value in 370 lies that far out.

```
column = faulty["temp_mean"]
z = (column - column.mean()) / column.std(ddof=0)      # z -score by hand; NaN stay NaN, scipy.
    stats.zscore would do too
outliers = z.abs() > 3                                 # careful: "< 3" would be the mask of the
     *non* -outliers; NaN gives False
print(faulty.loc[outliers, ["date_raw", "temp_mean"]])

date_raw  temp_mean
150  20250531      231.0
300  20251028      -45.0
```

Both errors we built in are found, and nothing else is. That is not a given: the z-score relies on the mean and the standard deviation, and the outliers themselves distort both. A single very large error can inflate the standard deviation so much that smaller errors stay below the threshold. The variant built on the median and the interquartile range, the one the boxplot uses, is more robust.

What do you do with an outlier once you have found it? Not delete the row: the other columns for that day are fine. Set the value to `NaN` instead, so that the next section treats it like any other missing value, and keep the mask.

```
cleaned = faulty.copy()
cleaned["temp_mean_outlier"] = outliers
cleaned.loc[outliers, "temp_mean"] = np.nan
print("Mean after cleaning:", cleaned["temp_mean"].mean().round(2))
```

```
Mean after cleaning: 11.73
```

4.5.3 Missing values

Pandas has three answers to a missing value: ignore it (most summary measures, `mean` among them, simply leave `NaN` out), discard the row (`dropna`), or insert a substitute value (`fillna`, **imputation**). Discarding is honest but costs data, and not at random: if the duration of sunshine is missing for precisely the last few months, `dropna` throws away precisely those months. Imputation keeps the rows but invents values, and the question is which substitute values do the least damage. We compare the usual strategies on the daily mean temperatures we deleted on purpose, where we know the truth, with the mean absolute error as the target metric.

```
column = cleaned["temp_mean"]
month = cleaned["date_raw"] // 100 % 100                    # month number out of YYYYMMDD

strategies = {
    "constant 0":            column.fillna(0),
    "global mean":            column.fillna(column.mean()),
    "global median":          column.fillna(column.median()),
    "monthly mean":           column.fillna(column.groupby(month).transform("mean")),
    "last value (ffill)":     column.ffill(),
    "linear interpolation":   column.interpolate(),
}
result = {name: (replacement[deleted] - truth[deleted]).abs().mean() for name, replacement in
    strategies.items()}
pd.Series(result, name="MAE in °C").round(2).sort_values()
```

```
linear interpolation     1.28
last value (ffill)       1.60
monthly mean             2.90
global median            6.11
global mean              6.23
constant 0              13.92
Name: MAE in °C, dtype: float64
```

The ranking is typical of time series: whatever knows the neighbours (interpolation, last value) beats whatever knows only the group (monthly mean), which in turn beats whatever knows only the whole (global mean or median). The constant 0 trails far behind and is nevertheless common in practice, because some tools use it as their default. For other data the ranking can look different: for a column with no temporal order `ffill` is pointless, and for skewed distributions with outliers the median is preferable to the mean.

Two details that come up again and again:

- `groupby(...).transform("mean")` computes the mean of each group and writes it back into every row of that group; the result is as long as the column and can go straight into `fillna`. If a group is entirely empty, its mean is `NaN` and the gap stays a gap – as it should.
- Older guides write `fillna(method="ffill")`. That no longer exists in Pandas 3; the methods are now `ffill()` and `bfill()`.

And rule 1 once more: after imputation the column itself no longer knows which of its values are real. The mask `temp_mean_missing` from the start is the only place where that information survives.

```
cleaned["temp_mean"] = cleaned["temp_mean"].interpolate()
cleaned["temp_mean_imputed"] = cleaned["temp_mean_missing"] | cleaned["temp_mean_outlier"]
print(cleaned["temp_mean_imputed"].sum(), "values in the column are not measured but inserted")
```

```
32 values in the column are not measured but inserted
```

4.5.4 Data types, encoding and categorical values

The first check after loading is `info()`: if a column of numbers has type `object`, some text has crept in – a comma as decimal separator, say, a `k.A.` ("no information") or a thousands separator. Such a column is repaired with `pd.to_numeric(..., errors="coerce")`, which turns everything unreadable into `NaN`; afterwards you count how many values that affected.

Text also has an **encoding**. A file is a sequence of bytes, and only the encoding says which bytes stand for which character. The standard is UTF-8; older files from Windows programs are often in Latin-1 or Windows-1252. Assume the wrong encoding and you get either an error message or, worse, silently wrong characters.

```
word = "Düsseldorf"
utf8 = word.encode("utf -8")
latin1 = word.encode("latin -1")
print("UTF -8:  ", utf8, len(utf8), "bytes")
print("Latin -1:", latin1, len(latin1), "bytes")
print("Latin -1 bytes read as UTF -8:", latin1.decode("utf -8", errors="replace"))
print("UTF -8 bytes read as Latin -1:", utf8.decode("latin -1"))
```

```
UTF -8:   b'D\xc3\xbcsseldorf' 11 bytes
Latin -1: b'D\xfcsseldorf' 10 bytes
Latin -1 bytes read as UTF -8:  Dsseldorf
UTF -8 bytes read as Latin -1: DÃ \frac{1}{4}sseldorf
```

The second result is the notorious `DÃ` $\frac{1}{4}$`sseldorf` that haunts so many databases: somebody read UTF-8 as Latin-1 and saved the result. `pd.read_csv` has an `encoding` parameter for this; when in doubt, look at the first few bytes of the file with `open(..., "rb")`.

Categorical values from free-text fields arrive in many spellings. The route is always the same: normalise first (strip spaces, lower-case), then merge the remaining variants with a mapping table, and catch typos by string similarity. The standard library ships `difflib` for that; if you need something more exact, use the edit distance (Levenshtein distance), the number of insertions, deletions and substitutions between two strings.

```
import difflib

city = pd.Series(["Düsseldorf", "Duesseldorf", "düsseldorf ", "Dusseldorf", "Köln", "Koeln", "
    Düsseldorf", "Kln"])
normalised = city.str.strip().str.lower().str.replace("ue", "ü").str.replace("oe", "ö")
table = {"dusseldorf": "düsseldorf"}                          # known variants spelled out
normalised = normalised.replace(table)
known = ["düsseldorf", "köln"]
def closest(s):
    matches = difflib.get_close_matches(s, known, n=1, cutoff=0.6)
    return matches[0] if matches else s
result = normalised.map(closest)
print(pd.DataFrame({"raw": city, "normalised": normalised, "assigned": result}))
print("\nDistinct values before:", city.nunique(), " after:", result.nunique())
```

```
raw  normalised    assigned
0   Düsseldorf  düsseldorf  düsseldorf
1  Duesseldorf  düsseldorf  düsseldorf
2  düsseldorf   düsseldorf  düsseldorf
3   Dusseldorf  düsseldorf  düsseldorf
4         Köln        köln        köln
5        Koeln        köln        köln
6   Düsseldorf  düsseldorf  düsseldorf
7          Kln         kln        köln

Distinct values before: 7  after: 2
```

Seven spellings become two cities. This too is a distortion: assigning `Kln` to `köln` is a guess, and if a city called "Kln" really existed it would be wrong. That is why the table of assignments belongs in the log.

How categorical values are then turned into numbers (ordinal and one-hot encoding) is covered by the table of levels of measurement in the chapter Descriptive statistics.

4.5.5 Regular expressions

As soon as a column contains text you want to pull something out of (the title from a name, the domain from a URL, the postcode from an address), you need **regular expressions**. Their reputation as cryptic strings of symbols comes from the notation, not from the thing itself. A regular expression is a compact description of a finite automaton, of the kind you know from the course on formal languages: it runs over the string from left to right and decides at each position whether the character fits. Two rules make working with them bearable:

1. Write regular expressions **across several lines, with comments**. Python allows this with the flag `re.VERBOSE`: spaces and line breaks in the pattern are then ignored, and anything after a `#` is a comment.
2. Put whatever you want to pull out in **groups**. `re.search` returns the groups, and Pandas' `str.extract` turns them into columns.

```
import re

urls = pd.Series(["https://www.hhu.de/studium", "http://opendata.dwd.de/climate_environment/CDC/
    ",
                  "https://voelkel.pages.cs.uni -duesseldorf.de/ds -skript/", "mailto:
    info@example.org"])
pattern = re.compile(r"""
    ^https?://          # scheme: http or https, then ://
    ([^/]+)             # group 1: everything up to the next slash, that is the host
    (/.*)?$             # group 2 (optional): the path from the slash onwards
""", re.VERBOSE)
print(urls.str.extract(pattern))
```

```
0                          1
0                           www.hhu.de                    /studium
1                     opendata.dwd.de  /climate_environment/CDC/
2  voelkel.pages.cs.uni -duesseldorf.de                /ds -skript/
3                                  NaN                         NaN
```

The last row is `NaN`, because `mailto:` does not fit the pattern. That is intended: a regular expression tells you not only *what* it finds but also *that* something fails to conform to the expected format. The number of `NaN`s after an `extract` is therefore a measure of data quality.

The commonest trap is the **greed** of the repetition operators. `.*` grabs as many characters as it can and gives back only as many as the rest of the pattern needs in order to fit. For the automaton that means running to the end and then backing up. The "lazy" variant `.*?` takes as little as possible.

```
html = '<a href="one.html">One</a> and <a href="two.html">Two</a>'
print("greedy:", re.findall(r'<a href="(.*)">', html))     # takes up to the last ">
print("lazy:  ", re.findall(r'<a href="(.*?)">', html))    # takes up to the next ">

greedy: ['one.html">One</a> and <a href="two.html']
lazy:   ['one.html', 'two.html']
```

Two constructions are worth learning, because they solve problems you could otherwise tinker with for a long time.

The **negative lookahead** `(?!…)` checks that something is *not* at this position, without consuming any characters in the process. So `(?!23)\d\d` matches every two-digit number except `23`. Its counterpart `(?=…)` demands that something does follow.

A **backreference** `\1` stands for whatever group 1 matched, and demands that exactly the same thing appear once more. That is how you find doubled words – the classic typo that the spell checkers of early word processors never caught.

```
print(re.findall(r"(?!23)\d\d", "22 23 24"))   # 23 is missing, 22 and 24 are there

doubled = re.compile(r"\b(\w+)\s+\1\b")
print(doubled.search("She is the the best")[1])        # 'the'
print(doubled.sub(r"\1", "She is the the best"))       # once is enough
```

```
['22', '24']
the
She is the best
```

Groups are also why a pattern can return more than "matches / does not match". The pattern

```
r"^<([a -z]+?)>(.*)</([a -z]+?)>$"
```

takes `"<p>html code</p>"` apart into the three groups `"p"`, `"html code"` and `"p"`. But it also shows the limit straight away: it does not match `'<a href="url">text</a>'`, because the opening tag has an attribute in it. You can extend the pattern, and then comes the next counterexample. The lesson from this is well known and still holds unchanged: **you do not take HTML apart with regular expressions** but with a parser – HTML is not a regular language, and no finite automaton can count nested tags. For a single, known field in a text column, on the other hand, regular expressions are exactly the right tool.

You rarely need more than these building blocks. The following table is the short version, read from the top (binding weakly) to the bottom (binding strongly):

Building block	Notation	Example	Matches
alternation	`A|B`	`a|b`	`a` or `b`
concatenation	`AB`	`ab`	`ab`
repetition, any number of times	`A*`	`a*`	nothing, `a`, `aa`, …
repetition, at least once	`A+`	`a+`	`a`, `aa`, …
repetition, optional	`A?`	`a?`	nothing or `a`
repetition, counted	`A{n,m}`	`a{2,4}`	`aa`, `aaa`, `aaaa`
group	`(A)`	`(ab|c)`	`ab` or `c`, and remembered as a group
any character	`.`	`ab.d`	`abcd`, `ab7d`, …
character class	`[…]`	`[abc]`, `[a-z]`	one of the characters named
negated character class	`[^…]`	`[^def]`	anything except `d`, `e`, `f`
anchor	`^`, `$`	`abc$`	`abc` at the end of the line

Characters with a special meaning (`.`, `*`, `+`, `?`, `(`, `)`, `[`, `]`, `|`,`^`, `$`, `\`) get a backslash in front of them when they are meant literally: `\.` matches a full stop. That is why patterns in Python are always written as a raw string `r"…"` – otherwise every backslash would have to be doubled.

If you work with them regularly, read Friedl's book at some point Friedl [2006]; it explains how the automata really run and why some patterns are slow.

Example exercise: titles in the Titanic dataset

The `Name` column of the Titanic passenger list has the form `Surname, Title. Forenames`, for instance `Braund, Mr. Owen Harris`. Extract the title into a new column, count the values, merge rare and synonymous titles, and compute the survival rate for each title. What does the result say about the order in which people were rescued?

Solution

The two hidden cells above contain the code. The pattern `,\s*([^.]+)\.` captures everything after the comma and any spaces, up to the first full stop, into a group. There are seventeen titles, most of them rare; `Mlle` and `Ms` are variants of `Miss`, `Mme` of `Mrs`, and the professional and noble titles are merged into two groups. The survival rates then come out at around 70 to 80 per cent for `Mrs` and `Miss`, above 50 per cent for `Master` (boys), and below 20 per cent for `Mr`. That is "women and children first", read off a text field. The title is therefore a better feature for prediction than sex alone, because it also encodes the age of the boys; this sort of thing is called *feature engineering*.

Regular expressions are not learned by reading about them. Two sites help with the practice: regexone.com walks through all the building blocks in fifteen small exercises, and the regex visualizer draws

the finite automaton for a pattern – for `a(ffe|de)+` you can see at once where the loop created by the `+` sits.

4.5.6 Date, time and time zones

Dates arrive as text in one of dozens of formats: `20250115`, `15.01.2025`, `01/15/2025`, `2025-01-15T14:30:00+01:00`, or as a count of seconds since 1 January 1970 (the *Unix timestamp*). Pandas reads most of them with `to_datetime`. State the format, though, rather than letting it be guessed: with a date like `01/02/2025` guessing is a matter of luck, and it is slow besides. And keep the raw date as a column until you are sure the conversion is right.

```
cleaned["date"] = pd.to_datetime(cleaned["date_raw"], format="%Y%m%d")
print(cleaned[["date_raw", "date"]].head(3))
print("\nType of the new column:", cleaned["date"].dtype)
```

```
date_raw       date
0  20250101 2025 -01 -01
1  20250102 2025 -01 -02
2  20250103 2025 -01 -03

Type of the new column: datetime64[us]
```

With a proper date type the useful questions become one-liners: day of the week, month, the interval between two days, or aggregating up to months with `resample`. For that we make the date the index.

```
w = cleaned.set_index("date")
print("Day of the week of the first entry:", w.index[0].day_name())
print("Days between the first and the last entry:", (w.index[ -1] - w.index[0]).days)
monthly = w[["temp_mean", "precipitation"]].resample("MS").agg({"temp_mean": "mean", "
    precipitation": "sum"})
monthly.round(1).head(6)
```

```
Day of the week of the first entry: Wednesday
Days between the first and the last entry: 364
```

	temp_mean	precipitation
date		
2025-01-01	3.2	103.6
2025-02-01	4.3	18.6
2025-03-01	8.0	6.2
2025-04-01	12.1	71.0
2025-05-01	14.5	23.8
2025-06-01	19.6	52.4

Time zones are where most of the mistakes happen. A timestamp without a time zone is ambiguous: `2025-03-30 02:30` never existed in Germany, because that night the clocks jumped from 2:00 to 3:00, and `2025-10-26 02:30` existed twice. Pandas therefore distinguishes *naive* timestamps (without a time zone) from *aware* ones (with one). `tz_localize` declares which time zone a naive timestamp was meant in; `tz_convert` converts an aware one into another time zone. Confuse the two and every time shifts by an hour or two – which nobody notices until the data have sunrise at 6 o'clock instead of 7.

```
times = pd.to_datetime(["2025 -01 -15 12:00", "2025 -07 -15 12:00"])    # naive: no time zone
local = times.tz_localize("Europe/Berlin")                              # this is how they were
    meant
print("local:", list(local))
print("UTC:  ", list(local.tz_convert("UTC")))                          # one hour's difference in
     winter, two in summer
print("Unix timestamp:", list((local - pd.Timestamp("1970 -01 -01", tz="UTC")) // pd.Timedelta("
    1s")))  # seconds since 1970, time -zone -free
try:
    pd.Timestamp("2025 -03 -30 02:30").tz_localize("Europe/Berlin")
except Exception as error:
    print("Error:", type(error).__name__, " -", error)
```

```
local: [Timestamp('2025 -01 -15 12:00:00+0100', tz='Europe/Berlin'), Timestamp('2025 -07 -15
    12:00:00+0200', tz='Europe/Berlin')]
UTC:   [Timestamp('2025 -01 -15 11:00:00+0000', tz='UTC'), Timestamp('2025 -07 -15
    10:00:00+0000', tz='UTC')]
Unix timestamp: [1736938800, 1752573600]
Error: ValueError - 2025 -03 -30 02:30:00 is a nonexistent time due to daylight savings time.
    Try using the 'nonexistent' argument.
```

The time zone rules, including every daylight-saving change of the past, live in the *tz database* (also called the IANA time zone database), which Python ships in the module `zoneinfo`; names such as `Europe/Berlin` come from there. Whoever stores timestamps stores them in UTC or as Unix timestamps, and converts to local time only for display.

4.5.7 Summary

Besides the measurements, the cleaned DataFrame now carries three columns that are nothing but the log: which values were missing, which were identified as outliers, and which were therefore inserted. Whoever later trains a model on it can exclude or weight those rows, and whoever reads the analysis can see what is measurement and what is guesswork.

```
cleaned[["date", "temp_mean", "temp_mean_missing", "temp_mean_outlier", "temp_mean_imputed"]].
    iloc[148:153]
```

	date	temp_mean	temp_mean_missing	temp_mean_outlier	temp_mean_imputed
148	2025-05-29	14.7	False	False	False
149	2025-05-30	19.6	False	False	False
150	2025-05-31	19.2	False	True	True
151	2025-06-01	18.8	False	False	False
152	2025-06-02	15.7	False	False	False

Take-home message

Cleaning always distorts. It is permitted when every change is in the code, a mask records which values are affected, and the choice of method was measured against a target metric. Before any calculation, `info()` and a plot; after cleaning, another plot.

What comes next: with clean tables we can investigate the relationships between columns. The next chapter introduces covariance and correlation for exactly that.

5 Multivariate stochastics

5.1 Multivariate statistics: covariance and dependence

5.1.1 Joint distribution and Bayes

Definition
If $X\colon \Omega_1 \to \mathbb{R}$ and $Y\colon \Omega_2 \to \mathbb{R}$ are two random variables, we can form the **product**:

$$X \times Y\colon \Omega_1 \times \Omega_2 \to \mathbb{R}^2,\ (a,b) \mapsto (X(a), Y(b)). \tag{112}$$

Definition
If $X, Y\colon \Omega \to \mathbb{R}$ are two random variables with the same domain, we can form the **joint distribution**:

$$(X,Y) := (X \times Y) \circ \Delta\colon \Omega \to \mathbb{R}^2,\ a \mapsto (X(a), Y(a)). \tag{113}$$

Here $\Delta\colon \Omega \to \Omega \times \Omega$ is the *diagonal map* $a \mapsto (a,a)$.

Definition
For a random variable $X\colon \Omega \to \mathbb{R}^n$ we can write $X = (X_1, \ldots, X_n)$ with $X_i\colon \Omega \to \mathbb{R}$ given by $\pi_i \circ X$, where $\pi_i\colon \mathbb{R}^n \to \mathbb{R}$ is the i-th coordinate projection. The distributions of the X_i are called the **marginal distributions**, or **margins**, of X.

More generally, (X_1, X_3, X_9) is also called a marginal distribution of X, as is any composition of X with a linear projection $\pi\colon \mathbb{R}^n \to \mathbb{R}^m$ – not necessarily onto a coordinate (hyper)plane of the standard basis of $\mathbb{R}^n$.

Remark
The distribution of X_i is $P(X_i \in A) = P(X \in \pi_i^{-1}(A))$.

For a random variable $(X,Y)\colon \Omega \to \mathbb{R}^2$ we can compute the distribution of Y in the discrete case as $P(Y \in A) = \sum_{x \in X(\Omega)} P((X,Y) \in \{x\} \times A)$ (with the corresponding integral in the continuous case), having decomposed $\pi_2^{-1}(A) = X(\Omega) \times A$ into disjoint subsets.

Remark
For a random variable $(X,Y)\colon \Omega \to \mathbb{R}^2$,

$$P((X,Y) = (x,y)) = P(X = x, Y = y) = P(Y = y | X = x) P(X = x) \tag{114}$$

by the definition of conditional probability.

In the pmf notation we also write

$$p(x,y) = p(y|x)p(x) \tag{115}$$

where this is to be read as a definition:

$$p(y|x) = \frac{p(x,y)}{p(x)} \tag{116}$$

More precisely, one writes the distributions with their random variables as subscripts when several are in play:

$$p_{X,Y}(x,y) = p_{Y|X}(y \mid x)\, p_X(x), \tag{117}$$

where, for each fixed x with $p_X(x) > 0$, the function $p_{Y|X}(\cdot \mid x)$ is the pmf of the conditional distribution of Y (likewise in the continuous case, with densities instead of pmfs). This formula is called the **product rule**; the **sum rule** is the preceding remark – summing over the variable we are not interested in.

Proposition

Let $X, Y : \Omega \to \mathbb{R}$ be discrete random variables. Then the following are equivalent:

1. X and Y are independent (in the sense of the chapter Expected value and variance, section *Independence*);
2. the joint pmf factorises: $p_{X,Y}(x, y) = p_X(x)\, p_Y(y)$ for all x, y;
3. conditioning changes nothing: $p_{Y|X}(y \mid x) = p_Y(y)$ for all x with $p_X(x) > 0$ and all y.

Proof

The independence of X and Y means $P(X = x, Y = y) = P(X = x)\, P(Y = y)$ for all x, y, and that is literally statement 2. For the equivalence of 2 and 3 we substitute the definition $p_{Y|X}(y \mid x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$: if 2 holds, the factor $p_X(x) > 0$ cancels, leaving $p_{Y|X}(y \mid x) = p_Y(y)$. If conversely 3 holds, then $p_{X,Y}(x, y) = p_{Y|X}(y \mid x)\, p_X(x) = p_Y(y)\, p_X(x)$ for all x with $p_X(x) > 0$; for x with $p_X(x) = 0$ both sides are 0.

Statement 3 is the generalisation of $P(A \mid B) = P(A)$ from the chapter Expected value and variance: knowing X does not shift the distribution of Y. In the continuous case the same equivalence holds with densities, but only "almost everywhere" – densities, after all, are determined only up to null sets (see Continuous distributions).

Theorem

Bayes' theorem. Given two events $A, B \subseteq \Omega$,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{118}$$

Example

Why the base rate matters. A test detects a rare disease reliably: 99 % of those who have it are detected, and only 1 % of the healthy get a false positive. Even so, after a positive test the probability of actually having the disease is small. The reason is the *base rate* $P(A)$: if only one person in a thousand is ill, then for every ill person correctly detected there are about ten healthy ones falsely flagged – the small error rate of 1 % is acting on a group a thousand times larger. In the language of the confusion matrix from the chapter Titanic EDA: the test is a classifier, the 99 % is its recall (its *sensitivity*), the 1 % its false positive rate, and what we are after is its precision – the proportion of ill people among those who test positive. With $A =$ "ill" and $B =$ "test positive", Bayes' theorem performs exactly this weighting:

$$P(A|B) = \frac{0.99 \cdot 0.001}{0.99 \cdot 0.001 + 0.01 \cdot 0.999} = \frac{0.00099}{0.00099 + 0.00999} \approx 0.09. \tag{119}$$

$P(A|B)$ is emphatically not $P(B|A)$: it is $P(B|A)$ *times* the ratio $P(A)/P(B)$, and $P(B)$ is dominated by the large group of the healthy. Confusing the two conditional probabilities is the *base rate fallacy* – common enough to have a name of its own. For the Titanic rule "women survive", recall 0.68 and precision 0.74 were close together, because survivors and women are about equally frequent; here a base rate of one in a thousand stands between 0.99 and 0.09. (Where the denominator comes from is settled by the law of total probability below.)

Proof

The definition of conditional probability is $P(A|B) = \dfrac{P(A \cap B)}{P(B)}$, and we apply it once more with the roles of A and B swapped.

To work out the denominator $P(B)$, the following theorem often helps:

Theorem
The law of total probability. If $\Omega = A_1 \cup \cdots \cup A_n$ is a decomposition into pairwise disjoint events with $P(A_i) > 0$, then for every event B

$$P(B) = \sum_{i=1}^{n} P(B|A_i)\, P(A_i). \tag{120}$$

Proof
The sets $B \cap A_i$ are pairwise disjoint and their union is B, so $P(B) = \sum_i P(B \cap A_i)$ (finite additivity), and by the definition of conditional probability $P(B \cap A_i) = P(B|A_i)\, P(A_i)$.

In Bayes' theorem you can therefore replace the denominator by $P(B) = P(B|A)P(A) + P(B|A^c)P(A^c)$ when A and its complement are all you have to go on.

Theorem
Bayes' theorem, version 2. Given two random variables $X, Y : \Omega \to \mathbb{R}^n$,

$$P(X = x|Y = y) = \frac{P(Y = y|X = x)P(X = x)}{P(Y = y)}. \tag{121}$$

We often apply this as follows: Y is "known" and X is the provisional model. We then call $Y = y$ the *data* D and $X = x$ the *hypothesis* H.

For data and hypothesis this gives

Theorem
Bayes' theorem, version 3.

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)} \tag{122}$$

Here $P(H)$ is called the **prior** (the a priori probability of the hypothesis), $P(D)$ the **evidence**, $P(D|H)$ the **likelihood** and $P(H|D)$ the **posterior** (the a posteriori probability of the hypothesis). A way to remember it: the likelihood answers the question "how probable are the data, supposing the hypothesis is true?". Unlike $P(H)$, the likelihood $P(D|H)$ can at least in principle be computed.

Example
How to work out a likelihood. So far H and D have been events with finitely many outcomes. In practice the hypothesis is usually a whole distribution. Suppose we have n measurements $D = (x_1, \ldots, x_n)$ and work with normal distributions: the hypothesis H says that the data come from an $\mathcal{N}(\mu, \sigma^2)$ with $\mu = 0$ and $\sigma = 2$. The likelihood $P(D \mid H)$ is then the probability (or probability density) of seeing exactly these n values if the hypothesis is right – and since the measurements are taken to be independent, the joint density factorises into a product by the proposition above:

$$P(D \mid H) = \prod_{i=1}^{n} p_{\mu,\sigma}(x_i) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{(x_i - \mu)^2}{2\sigma^2}\right) \quad \text{with } \mu = 0,\ \sigma = 2. \tag{123}$$

Three roles have to be kept cleanly apart here: the *data-generating distribution*, which exists out in the world and which we do not know; the *observed data* D, which we have; and the *model distribution*, which we declare to be our hypothesis. The likelihood compares the second with the third. If we let μ and σ vary instead of holding them fixed, the number becomes a function $L(\mu, \sigma) := \prod_{i=1}^{n} p_{\mu,\sigma}(x_i)$,

and searching for the parameters that maximise it is the maximum likelihood principle – see Statistical models and estimators, where taking logarithms turns the product into a sum.

Theorem
Bayes' theorem, version 4.

$$\text{Posterior} = \frac{\text{Likelihood} \cdot \text{Prior}}{\text{Evidence}}. \tag{124}$$

Example
Bayesian updating. Today's posterior is tomorrow's prior. In the disease-test example the probability of being ill after a positive test was about 0.09. If the test is repeated, independently of the first run, then 0.09 is now the prior $P(H)$, the likelihood of a positive result is still 0.99 for the ill and 0.01 for the healthy, and the evidence is again worked out by total probability:

$$P(H|D) = \frac{0.99 \cdot 0.09}{0.99 \cdot 0.09 + 0.01 \cdot 0.91} \approx 0.91. \tag{125}$$

Two positive tests in a row turn one in a thousand first into nine per cent and then into ninety-one. Every new observation pushes the posterior further along, and the outcome depends on the initial prior: start at $P(H) = 0.001$ and you need two tests; start at 0.5 (because there are symptoms already, say) and a single test puts you at 0.99. This process is called *Bayesian updating*; the chapter Statistical models turns the likelihood into a general estimation procedure.

5.1.2 Covariance

First a warning familiar from the chapter Descriptive statistics: covariance and correlation measure a relationship across *all* the data points. If the data split into groups, the relationship within each group can have the opposite sign – see Simpson's paradox. So the numbers in this chapter tell you something only once you know that the sample comes from a single population.

Definition
Given two random variables $X, Y \colon \Omega \to \mathbb{R}$, the **covariance** is defined as

$$\operatorname{Cov}(X,Y) := \mathbb{E}\left((X - \mathbb{E}X)\,(Y - \mathbb{E}Y)\right). \tag{126}$$

In particular, $\mathbb{V}(X) = \operatorname{Cov}(X,X)$.

The covariance suffers from the same complaint as the variance: it depends on the scale. The cure is to normalise by the standard deviations:

Definition
The (Pearson) **correlation coefficient** is

$$\rho(X,Y) := \frac{\operatorname{Cov}(X,Y)}{\sigma(X)\sigma(Y)}. \tag{127}$$

Clearly $\rho(X,Y) = 0$ exactly when $\operatorname{Cov}(X,Y) = 0$.

One says that X and Y are **positively correlated** if $\rho(X,Y) > 0$, **negatively correlated** (or anticorrelated) if $\rho(X,Y) < 0$, and **uncorrelated** if $\rho(X,Y) = 0$.

Remark

For data. If $x, y \in \mathbb{R}^n$ are given as two columns of a table, you replace the expected value in the definitions by the mean and obtain the *empirical covariance* and the *empirical correlation coefficient*

$$\operatorname{Cov}(x,y) := \frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y}) = \frac{1}{n}\,\tilde{x}\cdot\tilde{y}, \qquad r := \frac{\operatorname{Cov}(x,y)}{\sigma(x)\,\sigma(y)} = \frac{\tilde{x}\cdot\tilde{y}}{\|\tilde{x}\|\,\|\tilde{y}\|}, \tag{128}$$

with the centred vectors $\tilde{x}, \tilde{y}$ from the chapter Expected value and variance. The correlation coefficient of data is thus, quite literally, the cosine of the angle between the centred columns. It is the number `np.corrcoef(x, y)[0, 1]` prints in the chapter Titanic EDA; `np.cov(x, y, ddof=0)` gives the covariance matrix in the n-version (without `ddof=0`, Numpy divides by $n-1$ here, unlike `np.var`). For the Iris exercise from the chapter Descriptive statistics you compute it once across all the rows and once per species:

```
x = datasets.load_iris().data; y = datasets.load_iris().target
print(np.corrcoef(x[:, 0], x[:, 1])[0, 1])                    # -0.118
for s in range(3):
    print(s, np.corrcoef(x[y == s, 0], x[y == s, 1])[0, 1])   # 0.743, 0.526, 0.457
```

Proposition

The covariance is symmetric: $\operatorname{Cov}(X,Y) = \operatorname{Cov}(Y,X)$

The covariance is bilinear: $\operatorname{Cov}(aX+bY, Z) = a\operatorname{Cov}(X,Z) + b\operatorname{Cov}(Y,Z)$ for $a, b \in \mathbb{R}$.

The covariance is positive (semi)definite: $\operatorname{Cov}(X,X) = \mathbb{V}(X) \geq 0$, and $\operatorname{Cov}(X,X) = 0$ holds exactly when X is almost surely constant.

Proof

Symmetry and semidefiniteness can be read straight off the definition ($\operatorname{Cov}(X,X) = \mathbb{E}((X-\mathbb{E}X)^2) \geq 0$). For bilinearity, $aX+bY-\mathbb{E}(aX+bY) = a(X-\mathbb{E}X) + b(Y-\mathbb{E}Y)$ by linearity of the expected value; multiplying out with $(Z-\mathbb{E}Z)$ and using linearity once more gives the claim. For the case of equality: $\mathbb{E}((X-\mathbb{E}X)^2) = 0$ says that the non-negative random variable $(X-\mathbb{E}X)^2$ has expected value 0, and hence is almost surely 0.

Symmetry and bilinearity give us the rules of calculation that come up constantly in practice:

Proposition

Rules of calculation for the covariance. For random variables X, Y and a constant $\beta \in \mathbb{R}$,

1. $\mathbb{V}(X+Y) = \mathbb{V}(X) + \mathbb{V}(Y) + 2\operatorname{Cov}(X,Y)$,
2. $\operatorname{Cov}(X,\beta) = 0$,
3. $\operatorname{Cov}(X+\beta, Y) = \operatorname{Cov}(X,Y)$,
4. $\operatorname{Cov}(X,Y) = \mathbb{E}(XY) - \mathbb{E}(X)\,\mathbb{E}(Y)$.

Proof

(1) $\mathbb{V}(X+Y) = \operatorname{Cov}(X+Y, X+Y) = \operatorname{Cov}(X,X) + \operatorname{Cov}(X,Y) + \operatorname{Cov}(Y,X) + \operatorname{Cov}(Y,Y)$ by bilinearity, and the two middle terms are equal by symmetry. (2) The constant random variable β has expected value β, so $\beta - \mathbb{E}\beta = 0$ and $\operatorname{Cov}(X,\beta) = \mathbb{E}((X-\mathbb{E}X)\cdot 0) = 0$. (3) Bilinearity and (2): $\operatorname{Cov}(X+\beta, Y) = \operatorname{Cov}(X,Y) + \operatorname{Cov}(\beta, Y) = \operatorname{Cov}(X,Y)$. (4) Multiply out the definition and use linearity of the expected value:

$$\mathbb{E}\big(XY - X\,\mathbb{E}Y - Y\,\mathbb{E}X + \mathbb{E}X\,\mathbb{E}Y\big) = \mathbb{E}(XY) - 2\,\mathbb{E}X\,\mathbb{E}Y + \mathbb{E}X\,\mathbb{E}Y = \mathbb{E}(XY) - \mathbb{E}X\,\mathbb{E}Y. \tag{129}$$

Rule (4) is the counterpart of the formula $\mathbb{V}(X) = \mathbb{E}(X^2) - (\mathbb{E}X)^2$ from the chapter Expected value and variance; for $X = Y$ the two coincide. From (1) it also follows that $\mathbb{V}(X_1 + \cdots + X_n) = \sum_i \mathbb{V}(X_i)$ whenever the X_i are pairwise uncorrelated.

Theorem

Independent random variables are uncorrelated. If X and Y are independent (in the sense of the chapter Expected value and variance, section Independence) and have finite variance, then $\mathbb{E}(XY) = \mathbb{E}(X)\,\mathbb{E}(Y)$ and hence $\mathrm{Cov}(X, Y) = 0$.

Proof

In the discrete case, independence makes the joint pmf a product, $p(x, y) = p_X(x)\,p_Y(y)$, and the double sum splits:

$$\mathbb{E}(XY) = \sum_x \sum_y x\,y\,p_X(x)\,p_Y(y) = \Big(\sum_x x\,p_X(x)\Big)\Big(\sum_y y\,p_Y(y)\Big) = \mathbb{E}(X)\,\mathbb{E}(Y). \tag{130}$$

In the continuous case the sums become integrals over the joint density $p(x, y) = p_X(x)\,p_Y(y)$ (joint densities are introduced in the next chapter), and the calculation is the same. Rule (4) then gives $\mathrm{Cov}(X, Y) = \mathbb{E}(XY) - \mathbb{E}X\,\mathbb{E}Y = 0$.

Remark

The converse fails: uncorrelated random variables can be dependent. Take X uniformly distributed on $[-1, 1]$ and $Y = X^2$: then $\mathbb{E}(XY) = \mathbb{E}(X^3) = 0 = \mathbb{E}X\,\mathbb{E}Y$, so $\mathrm{Cov}(X, Y) = 0$, and yet Y is a function of X. The covariance measures *linear* relationships only. For jointly normally distributed (X, Y), on the other hand, the converse does hold, as we shall see in the next chapter.

The covariance therefore behaves like an inner product – but only "modulo constants", since constant random variables have covariance 0 with everything, themselves included. This can be made rigorous by reading the covariance as an inner product on an $\mathcal{L}^2$ space of random variables in which two random variables differing only by a constant are identified. The correlation coefficient then plays the role of the cosine of the angle between two random variables. For data it literally is that (see the remark *For data* above); the "cosine similarity" so widespread in machine learning is the same cosine, just without the centring.

You can get some practice at "seeing" the correlation coefficient in a scatter plot of two variables, for instance with the game "Guess the Correlation" by Omar Wagih.

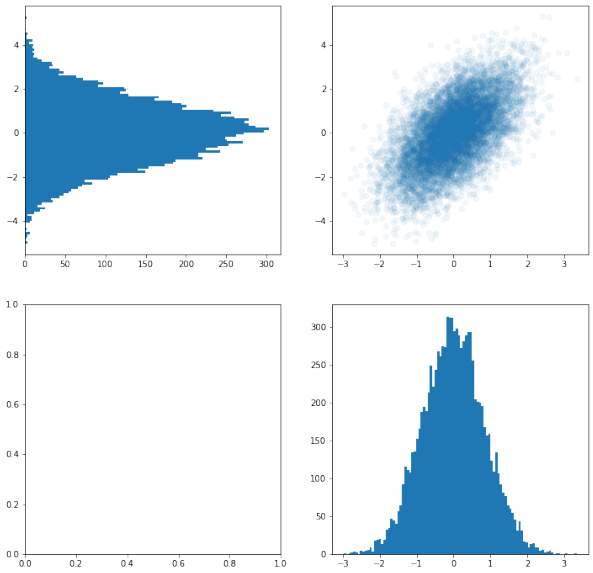


Figure 16: A sample from a bivariate normal distribution, with the two marginal histograms.

The picture shows why the correlation cannot be read off the individual variables. Each of the two marginal histograms is an unremarkable bell curve; they would look exactly the same if the correlation had the opposite sign, or were 0, because the marginal distributions do not depend on it. The relationship shows only in the joint scatter plot, in the tilt of the cloud of points. This is the observation made above about the joint distribution: it cannot be reconstructed from the individual ones.

What covariance and correlation mean geometrically we will see in the next chapter, on the multivariate normal distribution: there the covariance matrix describes the shape and position of the ellipses that are the contour lines of the density.

5.2 The multivariate normal distribution

5.2.1 Densities of multivariate distributions

Recall: if $f\colon \mathbb{R} \to [0,\infty)$ is a density function belonging to the distribution P of a real random variable X, then $P(X < c) = \int_{-\infty}^{c} f(x)dx$.

Generalising this to two dimensions, that is, to a random variable (X, Y) with values in $\mathbb{R}^2$, gives $P(X < c, Y < d) = \int_{-\infty}^{c}\int_{-\infty}^{d} f(x,y)\,dy\,dx$.

So the density function must have the signature $f\colon \mathbb{R}^2 \to [0,\infty)$: it encodes a probability density for every possible value of (X, Y).

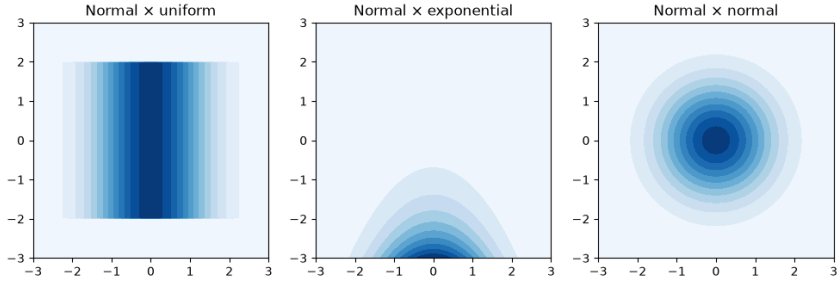
More generally, a function $f\colon \mathbb{R}^n \to [0,\infty)$ with $\int_{\mathbb{R}^n} f(x)dx = 1$ defines a probability measure P by $P(A) = \int_A f(x)dx$, and f is called a density for the distribution P.

5.2.2 Derivation: from the symmetry to the formula

Before we define the multidimensional normal distribution, we want to find it. The question is: which distribution on the plane $\mathbb{R}^2$ plays the role that $\mathcal{N}(0,1)$ plays on the line? The one-dimensional normal distribution has two properties that single it out: it is symmetric about its expected value, and among all distributions with a given variance it is the one with the greatest entropy, that is, the one that assumes nothing beyond its expected value and variance (see The normal distribution; exactly what entropy means for continuous distributions is settled in the chapter Relative entropy).

We already know one way of building a two-dimensional distribution out of two one-dimensional ones: the **product distribution** of two independent random variables, whose density is the product of the two densities. Let us try out a few candidates and look down on the density from above:

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt

xs = np.linspace( -3, 3, 200)
X, Y = np.meshgrid(xs, xs)
candidates = {
    "Normal \times uniform":     st.norm.pdf(X) * st.uniform.pdf(Y, loc= -2, scale=4),
    "Normal \times exponential": st.norm.pdf(X) * st.expon.pdf(Y + 3),
    "Normal \times normal":      st.norm.pdf(X) * st.norm.pdf(Y),
}
fig, axs = plt.subplots(1, 3, figsize=(12, 4))
for ax, (name, density) in zip(axs, candidates.items()):
    ax.contourf(X, Y, density, levels=12, cmap="Blues")
    ax.set_title(name)
    ax.set_aspect("equal")
plt.show()
```



Only the third picture is **rotationally symmetric**: you can turn it about the origin without anything changing. That is no accident; it is a calculation:

$$\phi(x)\,\phi(y) = \frac{1}{2\pi}\,e^{-x^2/2}\,e^{-y^2/2} = \frac{1}{2\pi}\,e^{-(x^2+y^2)/2} = \frac{1}{2\pi}\,e^{-r^2/2}, \tag{131}$$

the density depends only on the distance r from the origin. The product of two standard normal distributions is therefore the "most symmetric" distribution on the plane, and it inherits the maximum entropy property. This is our **standard normal distribution in two dimensions**, and in n dimensions it works the same way: $\phi(z) = (2\pi)^{-n/2}e^{-\|z\|^2/2}$ for $z \in \mathbb{R}^n$.

Every other normal distribution arises from it by three operations, which we add one at a time. Let Z be standard normally distributed in $\mathbb{R}^n$.

1. **Shifting:** $X = Z + \mu$ has expected value μ. The density is then evaluated at $z = x - \mu$.
2. **Scaling:** $X = DZ$ with a diagonal matrix $D = \operatorname{diag}(d_1, \ldots, d_n)$ stretches the i-th axis by d_i. The variance in that direction becomes d_i^2, since $\mathbb{V}(d_i Z_i) = d_i^2 \mathbb{V}(Z_i)$: double the data and you quadruple the variance.
3. **Rotating:** $X = RZ$ with an orthogonal matrix R (that is, $R^T R = I$) turns the cloud of points without changing any distances.

Put together: $X = AZ + \mu$ with $A = RD$. What density does X have? We undo the transformation, $Z = A^{-1}(X - \mu)$, and substitute into $e^{-\|z\|^2/2}$. For the squared length, every matrix M satisfies

$$\|Mz\|^2 = (Mz)^T (Mz) = z^T M^T M z, \tag{132}$$

because transposing a product reverses the order of the factors. With $M = A^{-1}$ and $z = x - \mu$ the exponent therefore reads $(x-\mu)^T (A^{-1})^T A^{-1}(x-\mu) = (x-\mu)^T (AA^T)^{-1}(x-\mu)$. The matrix

$$\Sigma := AA^T = RD^2 R^T \tag{133}$$

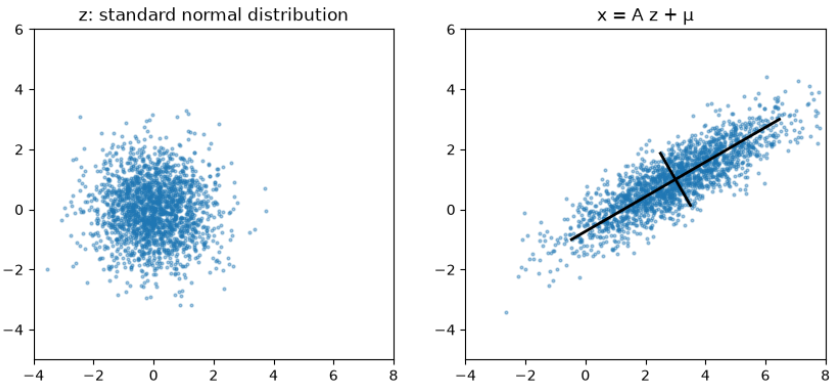
is exactly the covariance matrix of X (check this with the rules of calculation from the chapter Covariance), and Σ^{-1} in the exponent **undoes the rotation and the stretching**: it measures the distance from x to μ not with an ordinary ruler but in units of the spread in each direction. The normalising factor also picks up $\det(\Sigma)^{-1/2} = |\det A|^{-1}$, because the rotation and the stretching enlarge areas (volumes, in general) by $|\det A|$ and thin the density out accordingly.

We can follow this through on the computer: draw a standard normal sample, apply A to it, and read Σ and the stretch factors back off the sample:

```
rng = np.random.default_rng(1)
z = rng.standard_normal((2000, 2))          # standard normal in the plane, one row per point
D = np.diag([2.0, 0.5])                     # scaling: axis 1 by 2, axis 2 by 1/2
phi = np.pi / 6                             # rotating by 30 degrees
R = np.array([[np.cos(phi), -np.sin(phi)],
              [np.sin(phi),  np.cos(phi)]])
mu = np.array([3.0, 1.0])                   # shifting
A = R @ D
x = z @ A.T + mu                            # x = A z + mu, row by row for all points

Sigma = A @ A.T
print("Sigma = A A^T =\n", Sigma.round(3))
print("empirical covariance matrix of the sample:\n", np.cov(x.T).round(3))
values, vectors = np.linalg.eigh(Sigma)     # Sigma is symmetric, hence eigh
print("square roots of the eigenvalues:", np.sqrt(values).round(3), " - - these are the stretch
    factors 1/2 and 2")

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].scatter(z[:, 0], z[:, 1], s=4, alpha=0.4)
axs[0].set_title("z: standard normal distribution")
axs[1].scatter(x[:, 0], x[:, 1], s=4, alpha=0.4)
axs[1].set_title("x = A z + μ")
for i in range(2):                          # eigenvectors as the axes of the ellipse, length 2
    standard deviations
    v = vectors[:, i] * np.sqrt(values[i]) * 2
    axs[1].plot([mu[0] - v[0], mu[0] + v[0]], [mu[1] - v[1], mu[1] + v[1]], color="black", lw=2)
for ax in axs:
    ax.set_aspect("equal"); ax.set_xlim( -4, 8); ax.set_ylim( -5, 6)
plt.show()
```

```
Sigma = A A^T =
 [[3.063 1.624]
 [1.624 1.187]]
empirical covariance matrix of the sample:
```

```
 [[3.094 1.61 ]
 [1.61  1.164]]
square roots of the eigenvalues: [0.5 2. ] - - these are the stretch factors 1/2 and 2
```



The black axes are the eigenvectors of Σ, scaled by the square roots of the eigenvalues: the rotation R sits in the eigenvectors, the stretching D in the eigenvalues. In practice this works the other way round: if your data have correlated features, you do not first have to "rotate them straight" before modelling them with a normal distribution. You substitute the empirical mean and the empirical covariance matrix directly into the formula, which we now record as a definition.

5.2.3 The multivariate normal distribution

Definition

Let $n \in \mathbb{N}$, let $\mu \in \mathbb{R}^n$ and let $\Sigma \in Mat(n \times n; \mathbb{R})$ be a positive definite symmetric matrix.

Then the distribution belonging to the density function

$$\phi(x) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{(x-\mu)^T \Sigma^{-1} (x-\mu)}{2}\right) \tag{134}$$

is called the *n-dimensional (multivariate) normal distribution with expected value μ and covariance matrix Σ.*

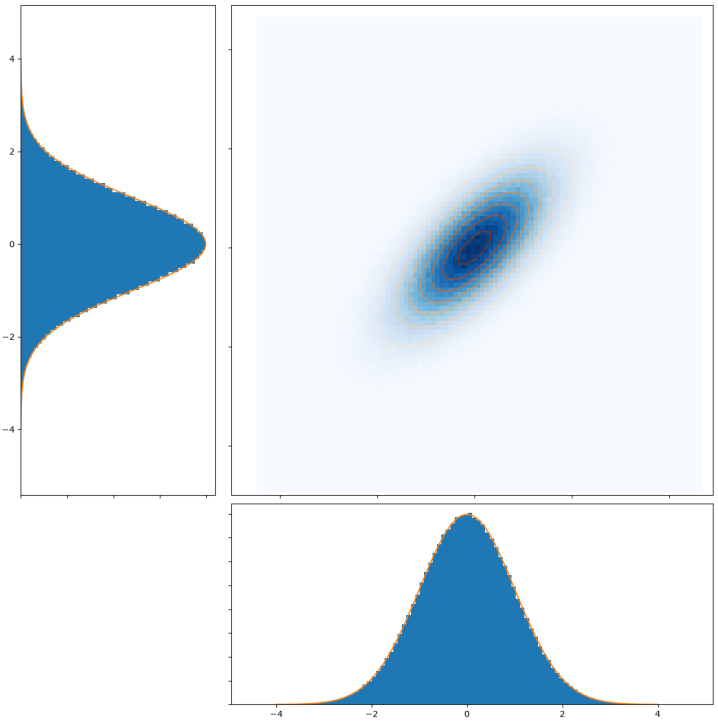
This distribution really does have the expected value and the covariance matrix its name promises; it is also written $\mathcal{N}(\mu, \Sigma)$.

In deep learning, a *variational autoencoder* models every data point as a normal distribution $\mathcal{N}(\mu, \mathrm{diag}(\sigma^2))$ in a latent space, where μ and σ are produced by the encoder network (the course *Introduction to Deep Learning*).

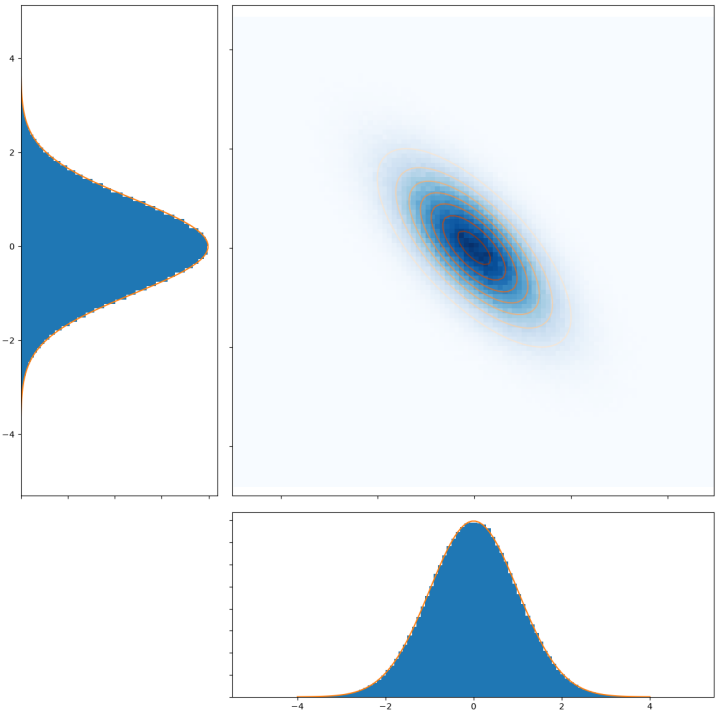
```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt

# instantiate a bivariate normal distribution:
def getmvn_by_rho(rho=0.7):
    origin = [0, 0]
    Sigma = np.array([[1,   rho],
                      [rho,   1]])
    Z = st.multivariate_normal(mean=origin, cov=Sigma)
    return Sigma, Z

# draw a sample:
def samplexy(Sigma, Z, size=1000000):
    sample = Z.rvs(size=size)
    x, y = sample[:,0], sample[:,1]
    # sample covariance == Sigma
    assert np.all(np.isclose(Sigma,
                             np.round(np.cov(x,y),
                                      decimals=2)))
    return x, y
```

Tip: read up on how *meshgrids* work in Numpy (and, with them, the function mgrid). A hint: a meshgrid is a kind of explicit broadcasting done in advance, so that the arithmetic afterwards runs vectorised and faster. Try it out yourself on small meshgrids by evaluating a function of two variables, $x^2 + y^2$ say, on a meshgrid in vectorised form.

```
# 2D plotting:
def plotalot(x, y, Z):
    fig, axs = plt.subplots(2,2, figsize=(9,9))
    fig.tight_layout(pad= -18)   # negative padding: the marginal histograms should sit right up
     against the main plot
    axs[0,1].hist2d(x, y, bins=100, cmap=plt.cm.Blues)
    xs, ys = np.mgrid[ -5:5:.1, -5:5:.1] # meshgrid
    axs[0,1].contour(xs, ys,
                     Z.pdf(np.dstack((xs, ys))),
                     cmap=plt.cm.Oranges, alpha=.7)
    axs[0,1].set_xticklabels([])
    axs[0,1].set_yticklabels([])
    axs[1,1].hist(x, bins=100, density=True)
    axs[1,1].plot(xl := np.linspace( -4,4,100),
                  st.norm.pdf(xl, loc=0, scale=1))
    axs[1,1].set_yticklabels([])
    axs[1,1].set_aspect(10)
    axs[0,0].hist(y, bins=100, density=True,
                  orientation="horizontal")
    axs[0,0].plot(st.norm.pdf(yl := np.linspace( -4,4,100), loc=0, scale=1),
                  yl)
    axs[0,0].set_xticklabels([])
    axs[0,0].set_aspect(0.1)
    axs[1,0].set_axis_off()
    plt.show()

def plot_by_rho(rho=0.7):
    Sigma, Z = getmvn_by_rho(rho)
    x, y = samplexy(Sigma, Z)
    plotalot(x, y, Z)

plot_by_rho()
```



```
# negatively correlated looks like this:
plot_by_rho( -0.7)
```



So the marginal distributions look exactly the same – the correlation shows only in the joint distribution, where it is plain enough to see: in the two-dimensional histogram (also called a heatmap) and in

the contour plot (the ellipses).

The covariance matrix has a direct geometric interpretation: every positive definite symmetric matrix A defines an inner product $\langle v, w\rangle_A := \langle Av, w\rangle = (Av)^T w = v^T A^T w = v^T Aw = \langle v, Aw\rangle$ and with it a so-called quadratic form $v \mapsto \langle v, v\rangle_A$. The contour lines (level sets) of the quadratic form are ellipses, with the same axes as the contour lines of the corresponding normal distribution.

The formula defining the density of the multivariate normal distribution contains the expression $v^T\Sigma^{-1}v$ for the vector $v = x - \mu$, and that is exactly the inner product defined by the matrix Σ^{-1}. The density is therefore $c \cdot e^{-Q(v)/2}$ with $Q(v) := v^T\Sigma^{-1}v$ and a constant c, and the level sets of the density are precisely the level sets of Q. For $\Sigma = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$ we have $\Sigma^{-1} = \frac{1}{1-\rho^2}\begin{pmatrix} 1 & -\rho \\ -\rho & 1 \end{pmatrix}$.

Remark

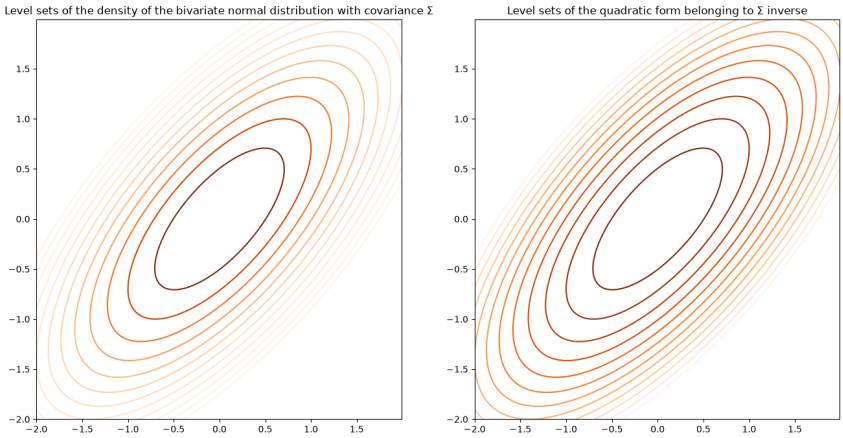
The multivariate normal distribution is always unimodal. Because Σ^{-1} is positive definite, $Q(v) \geq 0$ with equality only for $v = 0$, and along every ray leading away from μ the form Q is strictly increasing. The density $c \cdot e^{-Q(v)/2}$ therefore has exactly one maximum, at $x = \mu$, and falls off from it in every direction – whatever Σ looks like, including a negative covariance or very different variances. The covariance matrix deforms the peak into an ellipse; it cannot create a second one. So if a two-dimensional histogram shows two peaks, that is a sure sign that the data do *not* come from a single normal distribution – but rather, say, from a mixture of several, as in the chapter Expectation Maximization.

The covariance matrix encodes exactly how the ellipse arises from a perfect circle. The circle corresponds to the identity matrix as covariance matrix, and $\Sigma = RD^2R^\top$ from the derivation above says in which directions (the columns of R) and how strongly (the squares of the entries of D) the stretching happens. Σ itself, being symmetric, rotates nothing; Σ^{-1} has the same eigenvectors and compresses where Σ stretches. That is precisely why Σ^{-1} in the density undoes the stretching and measures the distance to μ in units of the spread.

To get a better grip on the geometry, start with bivariate normal distributions and make the matrix Σ more complicated step by step (multiples of the identity matrix, diagonal matrices with different entries on the diagonal, arbitrary matrices). Any covariance matrix Σ whatsoever can be brought into diagonal form by an orthogonal change of basis, which is to say that rotating the space makes the ellipse (the contour line of the normal distribution) parallel to the axes. Scaling the axes then turns the ellipse into a circle. Seen that way, every normal distribution is a standard normal distribution, just in different coordinates.

```
rho = 0.7
Sigma, Z = getmvn_by_rho(rho)
Sigma_inv = np.linalg.inv(Sigma)   # = 1/(1 -rho^2) * [[1, -rho], [ -rho, 1]]

def Q(x, y):
    """Quadratic form v^T \Sigma^{ -1} v for v = (x, y)"""
    return Sigma_inv[0,0]*x**2 + 2*Sigma_inv[0,1]*x*y + Sigma_inv[1,1]*y**2

xs, ys = np.mgrid[ -2:2:.01, -2:2:.01]

# The density is c * exp( -Q/2) with c = 1/(2 pi sqrt(det \Sigma)). On the left and on the right
# we draw the same levels, so that you can see they are the same ellipses:
levels_Q = np.arange(0.5, 6.5, 0.5)
c = 1 / (2 * np.pi * np.sqrt(np.linalg.det(Sigma)))
levels_pdf = np.sort(c * np.exp( -levels_Q / 2))

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16,8))
ax1.contour(xs, ys,
            Z.pdf(np.dstack((xs, ys))),
            levels=levels_pdf,
            cmap=plt.cm.Oranges)
ax1.set_title("Level sets of the density of the bivariate normal distribution with covariance \
    Sigma")
ax2.contour(xs, ys, Q(xs, ys),
            levels=levels_Q,
```

```
            cmap=plt.cm.Oranges_r)
ax2.set_title("Level sets of the quadratic form belonging to \Sigma inverse")

plt.show()
```



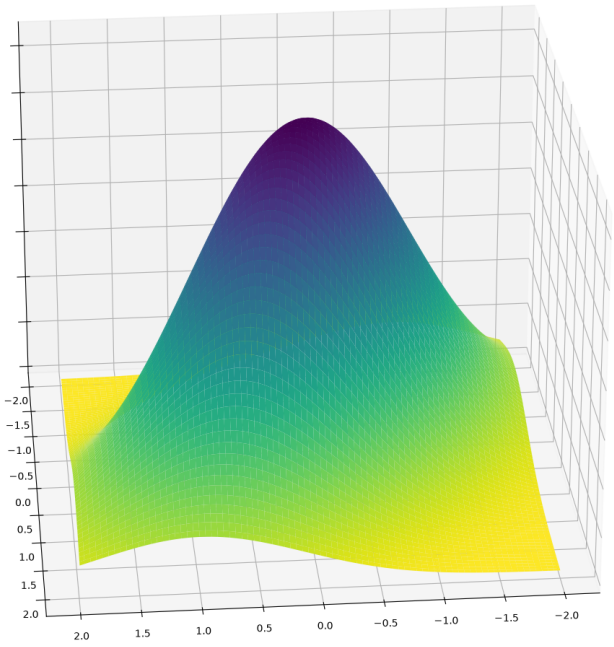
The axes of the ellipses can also be computed: they are the eigenvectors of Σ, and the square roots of the eigenvalues give (up to a common factor) the lengths of the semi-axes. For $\Sigma = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$ these are the two diagonals, with eigenvalues $1+\rho$ and $1-\rho$. The chapter on Dimensionality reduction builds precisely on this eigendecomposition:

```
values, vectors = np.linalg.eigh(Sigma)    # Sigma is symmetric, hence eigh (eigenvalues in
    ascending order)
print("Eigenvalues:", values)              # 1 - rho and 1 + rho
print("Eigenvectors (columns):")
print(vectors)                             # (1, -1)√/2 and (1,1)√/2: the axes of the ellipses
```

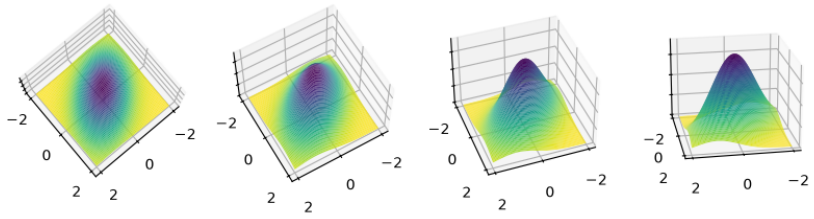
```
Eigenvalues: [0.3 1.7]
Eigenvectors (columns):
[[ -0.70710678  0.70710678]
 [ 0.70710678  0.70710678]]
```

To get to know three-dimensional plotting with matplotlib as well, let us take another look at the density function of the normal distribution:

```
# 3D plotting:
from mpl_toolkits.mplot3d import axes3d

rho = 0.5
Sigma, Z = getmvn_by_rho(rho)

def plot3dangles(a, b, fig, where):
    ax = fig.add_subplot(where, projection='3d')
    ax.view_init(a, b)
    ax.set_zticklabels([])
    ax.plot_surface(xs, ys, Z.pdf(np.dstack((xs, ys))),
                    rstride=5, cstride=5,
                    cmap='viridis_r', edgecolor='none')

fig=plt.figure(figsize=(15,15))
plot3dangles(25, 85, fig, 111)
plt.show()
```




```
fig=plt.figure(figsize=(10,10))
plot3dangles(80, 50, fig, 141)
plot3dangles(60, 60, fig, 142)
plot3dangles(40, 70, fig, 143)
plot3dangles(20, 80, fig, 144)
plt.show()
```



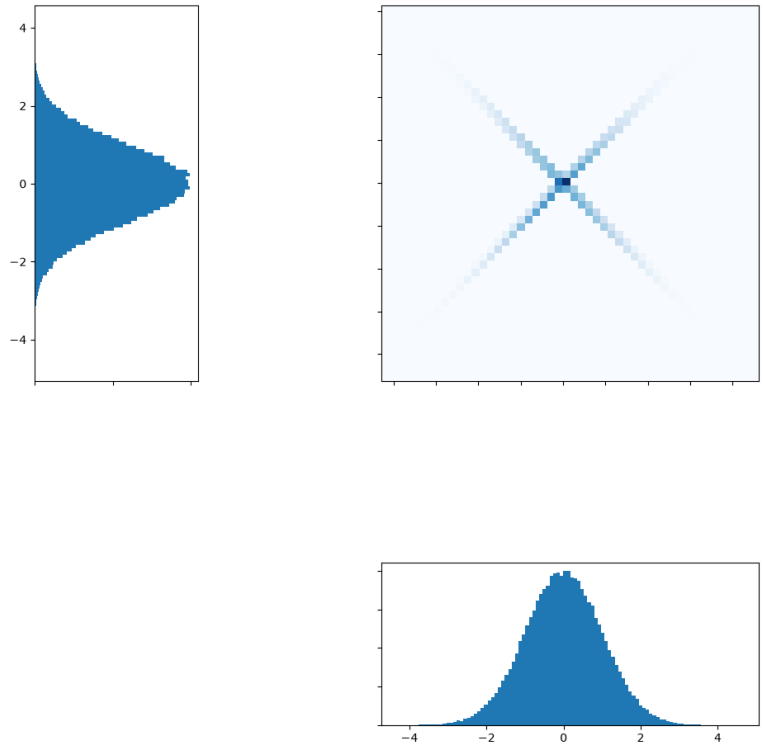
Lemma
The multivariate normal distribution with expected value μ and covariance Σ is the distribution of maximal entropy among all distributions with this expected value and this covariance matrix whose support is all of $\mathbb{R}^n$.

Lemma
A random vector $X = (X_1, \ldots, X_n)$ (that is, a multivariate distribution) is multivariate normally distributed if and only if every linear combination $\sum_{i=1}^{n} \lambda_i X_i$ is univariate normally distributed.

Careful

This does not mean that every linear combination of normally distributed random variables is again normally distributed! Here is an example that could almost be from the Datasaurus: X and Y are each normally distributed on their own but dependent on one another – and bang, $X + Y$ is not normally distributed (because X+Y=0 has probability 50%):

```
# plot the weirdo sample:
def plotalot2(x, y):
    fig, axs = plt.subplots(2,2, figsize=(13,13))
    axs[0,1].hist2d(x, y, bins=50, cmap=plt.cm.Blues)
    axs[0,1].set_xticklabels([])
    axs[0,1].set_yticklabels([])
    axs[1,1].hist(x, bins=100, density=True)
    axs[1,1].set_yticklabels([])
    axs[1,1].set_aspect(10)
    axs[0,0].hist(y, bins=100, density=True,
                  orientation="horizontal")
    axs[0,0].set_xticklabels([])
    axs[0,0].set_aspect(0.1)
    axs[1,0].set_axis_off()
    plt.show()

samplesize = 100000
x = st.norm.rvs(0, 1, size=samplesize)
t = 2*st.bernoulli.rvs(0.5, size=samplesize) - 1
y = t*x
plotalot2(x, y)
```



Wikipedia has a nice article on the fact that normally distributed uncorrelated random variables can perfectly well be dependent on one another.

The random variables X, Y from the previous plot are exactly such an example: their correlation coefficient is 0.

In the other direction there is a statement with strong hypotheses:

Lemma
If X and Y are two real random variables that are jointly bivariate normally distributed, that is $(X, Y) \sim \mathcal{N}(\mu, \Sigma)$, then X and Y are independent as soon as they are uncorrelated.

The converse holds as well: if X and Y are each univariate normally distributed *and* independent, then (X, Y) is bivariate normally distributed.

Exercise:
Convince yourself that you know what the covariance matrix looks like in the case of independence.

5.3 Relative entropy and Kullback–Leibler divergence

5.3.1 Logarithms

The elementary properties of the logarithm – definition, change of base, bit and nat – are covered in the digression in the chapter Expected value and variance. Here we are after a different question: why it is the logarithm, of all functions, that turns up in information theory, and nothing else.

Lemma
Let $l\colon \{1,\dots\} = \mathbb{N}_{>0} \to \mathbb{R}$ be a function with

1. $l(ab) = l(a) + l(b)$ – "completely additive"
2. $l(n+1) \geq l(n)$ – "monotonically increasing" Then there is a constant $c \in \mathbb{R}$ such that $l(n) = c\ln(n)$.

Remark
Because of the formula

$$\ln(x) = \frac{\log_a(x)}{\log_a(e)} \tag{135}$$

we can read the constant c as $c = \log_a(e)$, that is, the function l is a logarithm to base a.

Equally clearly, the same lemma holds for $\log_2$ in place of ln (only the constant is different).

So if we want to count something in such a way that the count of a product is the sum of the individual counts, we have to use a logarithm.

Proof
We want to show that for functions $f, g\colon \mathbb{N}_{>0} \to \mathbb{R}$ satisfying the conditions of the lemma,

$$f(n)g(2) = f(2)g(n) \tag{136}$$

always holds, since with $g = \ln$ this proves the lemma ($c = f(2)$).
Let $n, k, m \in \mathbb{N}$ be such that $2^{m-1} \leq n^k \leq 2^m$. Monotonicity then gives

$$f(2^{m-1}) \leq f(n^k) \leq f(2^m) \tag{137}$$

and additivity gives

$$(m-1)f(2) \leq kf(n) \leq mf(2) \tag{138}$$

The same chain of inequalities holds for g in place of f; this one we multiply by -1 as well:

$$-mg(2) \leq -kg(n) \leq -(m-1)g(2). \tag{139}$$

Adding the two chains of inequalities and simplifying gives

$$-f(2)g(2) \leq k\left(f(n)g(2) - f(2)g(n)\right) \leq f(2)g(2) \tag{140}$$

Dividing by k and letting $k \to \infty$, we see that $f(n)g(2) - f(2)g(n) \to 0$ (as $k \to \infty$, on which the expression does not depend at all).

Lemma
If a continuous function $F\colon \mathbb{R} \to \mathbb{R}$ satisfies the functional equation $F(xy) = xF(y) + F(x)y$ and $F(x)/x$ is monotonically increasing, then there is a constant $c \in \mathbb{R}$ such that $F(x) = -x\log(x)c$.

Proof
The function $g(x) := F(x)/x$ satisfies $g(xy) = g(x) + g(y)$ and is monotonic by assumption, so there is a c' with $g(x) = c' \log(x)$, and $c := -c'$ does the job.

Remark
So the entropy contribution $-p\log(p)$ of an outcome with probability p turns out to be a very natural quantity. (The *self-information* $I(x) = -\log P(\{x\})$ from the chapter Expected value and variance is the factor that multiplies the p.)

Lemma
The logarithm is a *concave* function, that is, for a convex combination $\lambda x + \mu y$ (so $\lambda + \mu = 1$ and $0 \leq \lambda, \mu \leq 1$) we have

$$\lambda \log(x) + \mu \log(y) \leq \log(\lambda x + \mu y) \tag{141}$$

Proof
Idea: The derivative of $\log(x)$ is $\frac{1}{x}$, which is strictly decreasing on $(0, \infty)$. The mean value theorem then shows that $\log(x)$ must be strictly concave (so in fact $<$ rather than $\leq$ in the inequality above).

We have argued this heuristically before; now we can prove it rigorously:

Proposition
If $p = (p_1, \ldots, p_n)$ describes a discrete distribution on $\Omega = \{1, \ldots, n\}$, then the entropy satisfies $H(p) \leq \log(n)$ (the entropy of the uniform distribution).

Proof
We use the fact that the sum in the definition of the entropy is a convex combination, and that the logarithm is concave:
$H(p) = \sum_{i=1, p_i \neq 0}^{n} p_i \log \frac{1}{p_i}$
$\leq \log\left(\sum_{i=1, p_i \neq 0}^{n} p_i \frac{1}{p_i}\right)$
$\leq \log\left(\sum_{i=1}^{n} 1\right) = \log(n)$

5.3.2 Definition and properties

Definition
Let $p, r \in \Delta_n$ (the set of probability vectors in $\mathbb{R}^n$, that is, vectors whose entries lie between 0 and 1 and sum to exactly 1). The *entropy* (also: Kullback–Leibler divergence D_{KL}) of p relative to r is
$H(p\|r) := \sum_{i,\ p_i > 0}^{n} p_i \log\left(\frac{p_i}{r_i}\right) \in \mathbb{R} \cup \{\infty\}$.

Proposition
Write $u_n = (\frac{1}{n}, \ldots, \frac{1}{n}) \in \Delta_n$ for the uniform distribution on $\{1, \ldots, n\}$. Then

$$H(p\|u_n) = H(u_n) - H(p) = \log(n) - H(p) \tag{142}$$

Proposition

$$H(p\|r) = 0 \text{ if and only if } p = r \tag{143}$$

(without proof)

Proposition

$$H(p\|r) \geq 0 \tag{144}$$

Proof
$H(p\|r) = \sum p_i \log\left(\frac{p_i}{r_i}\right) = -\sum p_i \log\left(\frac{r_i}{p_i}\right)$ (since $\log(a/b) = -\log(b/a)$) $\geq -\log\left(\sum p_i \frac{r_i}{p_i}\right) \geq -\log\left(\sum r_i\right) = -\log 1 = 0$

where we have again used the concavity of the logarithm.

Example
Relative entropy is not symmetric, and it can become infinite. Let $p_t := (t,\ 1-t)$ for $0 < t < 1$, a coin that for small t comes up 1 almost surely. In one direction,

$$H\left(u_2 \,\|\, p_t\right) = \frac{1}{2}\log\frac{1}{2t} + \frac{1}{2}\log\frac{1}{2(1-t)} \xrightarrow{t\to 0} \infty, \tag{145}$$

since the first summand grows without bound, like $-\frac{1}{2}\log t$. In the other direction the proposition above gives

$$H\left(p_t \,\|\, u_2\right) = \log(2) - H(p_t), \qquad H(p_t) = -t\log t - (1-t)\log(1-t) \xrightarrow{t\to 0} 0, \tag{146}$$

so $H(p_t\|u_2) \to \log 2$: finite, and to base 2 exactly one bit. Mistake the almost certain coin p_t for a fair one and you pay at most one bit; mistake the fair coin for p_t, and the side you thought impossible will surprise you infinitely.

5.3.3 Joint entropy

Definition
The *joint entropy* of two discrete random variables X, Y is

$$H(X,Y) := \mathbb{E}I_{X,Y} = -\sum_{i=1}^{n}\sum_{j=1}^{n} P(X = z_i, Y = z_j)\log P(X = z_i, Y = z_j) \tag{147}$$

Proposition

$$H(X) + H(Y) \geq H(X,Y) \geq \max\left(H(X), H(Y)\right) \tag{148}$$

Example

$$H(X,X) = H(X) \tag{149}$$

since in the sum on the left every summand off the diagonal is 0.

Proposition
If X, Y are independent random variables, then $H(X,Y) = H(X) + H(Y)$.

Proof
Independence means that for all i, j:

$$P(X = z_i, Y = z_j) = P(X = z_i) \cdot P(Y = z_j) \tag{150}$$

We take logarithms:

$$\log P(X = z_i, Y = z_j) = \log P(X = z_i) + \log P(Y = z_j) \tag{151}$$

and so, all in all,

$$\begin{aligned}
H(X,Y) &:= -\sum_{i=1}^{n}\sum_{j=1}^{n} P(X = z_i, Y = z_j) \log P(X = z_i, Y = z_j) && (152)\\
&= -\sum_{i=1}^{n}\sum_{j=1}^{n} P(X = z_i)P(Y = z_j) \left(\log P(X = z_i) + \log P(Y = z_j)\right) && (153)\\
&= -\sum_{i=1}^{n}\sum_{j=1}^{n} P(X = z_i)P(Y = z_j) \log P(X = z_i) && (154)\\
&\quad -\sum_{i=1}^{n}\sum_{j=1}^{n} P(X = z_i)P(Y = z_j) \log P(Y = z_j) && (155)\\
&= -\sum_{i=1}^{n} P(X = z_i) \log P(X = z_i) && (156)\\
&\quad -\sum_{j=1}^{n} P(Y = z_j) \log P(Y = z_j) && (157)\\
&= H(X) + H(Y) && (158)
\end{aligned}$$

Definition
The *mutual information* (also *transinformation*) of two random variables X, Y is defined as

$$I(X;Y) := H(X) + H(Y) - H(X,Y) = H(X,Y \| X \times Y) \tag{159}$$

Here $H(\cdot\|\cdot)$ stands for the relative entropy of the *distribution* of (X,Y) with respect to the product distribution $P_X \times P_Y$, that is, the distribution (X,Y) would have if X and Y were independent.

A very common application of mutual information is in clustering. Suppose we have clustered a dataset by some method of our choice – that is, arrived at an estimated class membership (say 1, 2 or 3 as a label) – and a subset of the data has been classified by hand (for testing). Then we can compute the mutual information of these two class variables to quantify how well the clustering worked on the test data. The advantage is that the concrete labels no longer matter (that is, the method is invariant under permutation of the class names).

The *conditional entropy* of X given Y is defined as

$$H(X|Y) := H(X) - I(X;Y) \tag{160}$$

Example
The mutual information of X with itself is just the entropy: $I(X;X) = H(X)$.

If X, Y are independent, then $I(X;Y) = H(X) + H(Y) - (H(X) + H(Y)) = 0$.

Proposition

$$H(X|Y) = H(X,Y) - H(Y) \tag{161}$$

and if X, Y are independent, then $H(X|Y) = H(X)$.

Theorem
Entropic Bayes' theorem

$$H(X|Y) = H(Y|X) + H(X) - H(Y) \tag{162}$$

Example
If X depends entirely on Y (for example Y a throw of a die and $X = f(Y)$ with $f =$ mod 2), then $H(X,Y) = H(Y)$ and $H(X|Y) = 0$.

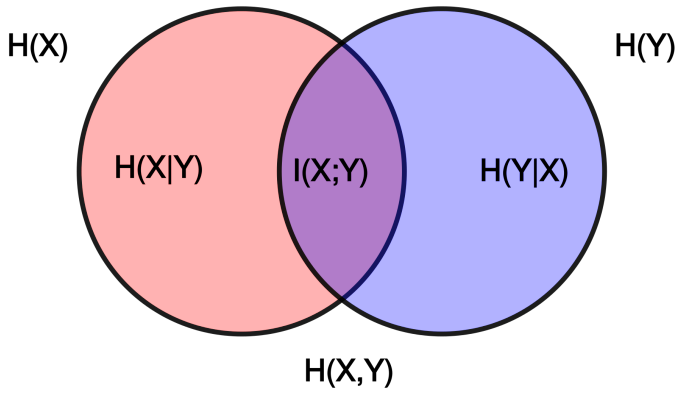


Figure 17: Entropy, conditional entropy, joint entropy and mutual information in a Venn diagram. Figure: Wikimedia Commons, user KonradVoelkel (2010, public domain).

Definition
The *cross-entropy* of X (distributed according to $p \in \Delta_n$) and Y (distributed according to $r \in \Delta_n$) is

$$H(X;Y) = \mathbb{E}_X I(Y) = -\sum_{i,\ p_i>0}^{n} p_i \log r_i \in \mathbb{R} \cup \{\infty\}. \tag{163}$$

A summand with $p_i > 0$ and $r_i = 0$ is ∞ here (with $\log 0 = -\infty$): an event that is impossible under r but occurs under p comes at infinite cost.

Careful: some authors unfortunately write the cross-entropy as $H(X,Y)$ too, which is easily confused with the joint entropy. And the notation $H(X;Y)$ resembles that of the mutual information $I(X;Y)$ above, which is something else entirely.

Proposition

$$H(X;Y) = H(X) + H(X\|Y) \tag{164}$$

If we hold X fixed and vary Y, minimising $H(X;Y)$ and minimising $H(X\|Y)$ come to the same thing, which is why machine learning often substitutes one for the other. The maximum likelihood method can be read as minimising a cross-entropy, and hence as minimising a relative entropy. That is how, and why, entropy matters to machine learning.

Remark
In deep learning you meet this observation under three names: *negative log-likelihood*, *cross-entropy* and *KL divergence to the labels* all refer to the same loss function for classification problems. The last two differ only by the constant $H(X)$ from the proposition above, which does not depend on the model parameters. The course *Introduction to Deep Learning* takes this connection for granted.

Otherwise, the cross-entropy is used considerably less often in the theory than the relative entropy (usually called the Kullback–Leibler divergence D_{KL}).

5.3.4 Continuous entropy

Note
This section is not part of the course proper; it reaches ahead to measure theory (Radon–Nikodym derivative, invariant measures), which this course does not cover. On a first reading it is enough to take away the core statement: in the continuous case, entropy is only meaningfully defined *relative* to a reference measure, so relative entropy is the fundamental notion.

One might think that the formula $H(p) = -\sum p_i \log p_i$ generalises to $H(P) = -\int \log P dP$, but as it stands that makes no sense. What one can do is this:

Definition
Let P be a continuous distribution with density function f defined on a set $\Omega \subseteq \mathbb{R}^n$. Then

$$h(X) := \mathbb{E}\left(-\log(f(X))\right) = -\int_\Omega f(x) \log f(x) dx \tag{165}$$

is called the *differential entropy* of X (likewise defined by Claude Shannon).

It already has some of the pleasant properties we would expect of a continuous entropy, translation invariance for instance: $h(X) = h(X - v)$ for a vector $v \in \mathbb{R}^n$.

Unfortunately the differential entropy is not invariant under arbitrary transformations, not even under isomorphisms.

Jaynes suggested picturing ever finer histograms, that is, a set of N discrete points $\{x_i\}$. We then write $M(a,b) = \lim_{N\to\infty} \frac{1}{N}(\text{number of } x_i : a < x_i < b)$ and call a density m with $M(a,b) = \int_a^b m(x)dx$ an *invariant measure*. With that in hand we write

$$\lim_{N\to\infty} \log(N) - \int p(x) \log \frac{p(x)}{m(x)} dx \tag{166}$$

but since $\lim_{N\to\infty} \log(N) = \infty$, we cheekily drop that part and so obtain a quantity that behaves much like $h(x)$ but is scale invariant:

$$H(X) := -\int p(x) \log \frac{p(x)}{m(x)} dx \tag{167}$$

The first thing to note is that the "discrete entropy" of a continuous distribution is really ∞. The second is that a workable notion of entropy evidently cannot do without reference to a further distribution (here the "invariant measure").

A theoretical analysis of what $h(X)$ and $H(X)$ really are leads to the general relative entropy $H(P, Q)$, with $h(X) = -H(f\lambda \,\|\, \lambda)$, where λ is the Lebesgue measure and f a density of X.

Definition
Let P, R be continuous distributions. The *relative entropy* $H(P\|R)$ (also: Kullback–Leibler divergence $D_{KL}(P, R)$) is

$$H(P\|R) := -\int_\Omega \log\left(\frac{dR}{dP}\right) dP \tag{168}$$

where $\dfrac{dR}{dP}$ is the Radon–Nikodym derivative of one measure with respect to another.

We will not pursue that any further either; instead we record that for P and R with density functions $p, r\colon \mathbb{R}^n \to [0, \infty)$ it looks simpler:

$$H(P\|R) = \int_\Omega p(x) \log \frac{p(x)}{r(x)} dx \tag{169}$$

This expression, in turn, we may evaluate by Riemann sums (if the density functions are friendly enough). If for instance $n = 1$ and $\Omega = \mathbb{R}$, we can look at

$H(P\|R) = \int_{\mathbb{R}} p(x) \log \frac{p(x)}{r(x)} dx = \sum_{k=-\infty}^{\infty} \int_k^{k+1} p(x) \log \frac{p(x)}{r(x)} dx$

and bound each of the integrals on the right by a value, so as to estimate $H(P\|R)$.

$$H(P\|R) \approx \sum_{k=-\infty}^{\infty} p(k) \log \frac{p(k)}{r(k)} \tag{170}$$

The subdivision into intervals of length 1 can of course be made as fine as we like, making the approximation as accurate as we like. Replacing a continuous distribution by such a discrete one is the analogue of forming a histogram.

If you still doubt that abstract information theory has any relevance to the practical work of data scientists and machine learning engineers, a trip into the Wikipedia page on applications of mutual information and conditional entropy is recommended.

6 Data protection

6.1 Data protection and the GDPR: a glossary

In computer science, **handling personal data responsibly** is a central concern – legally as much as ethically. This chapter gives an overview of the most important terms and principles of data protection in the context of the European General Data Protection Regulation (GDPR; in German *Datenschutz-Grundverordnung*, DSGVO).

The chapter sits here because from the next chapter on we train models on real datasets, and before personal data serve as training material the terms that govern how such data may be handled ought to be clear. It is deliberately a glossary, with no mathematics and no code. Each term comes with examples and with real cases; the fine cases are all taken from the GDPR Enforcement Tracker kept by the law firm CMS and linked there by number, so that every one of them can be checked for date, amount and source.

6.1.1 Personal data

Personal data are any information relating to an identified or identifiable natural person. A person counts as identifiable if they can be identified directly or indirectly – by a name, by location data or by an online identifier, for instance.

Pseudonymised and **encrypted** data enjoy the protection of the GDPR as well, and so do supposedly **anonymised** data, as soon as the link to a person can be restored with reasonable effort (recital 26).

Examples:

- Your favourite colour of gummy bear is not a personal datum. The same answer in a table beside a student number is one.
- A longitudinal survey in schools wants to link the answers of the same children across years without storing any names. A code built from the initial letters of the parents' names, or a hash of the name with a fixed prefix, is a **pseudonym**: anyone who knows the rule can work backwards, so the data remain personal. A number handed to the children is a number nobody brings back the following year, and a link with the number encoded in it destroys the very trust in anonymity that it was meant to protect.
- Whether data are anonymous is a question of information, not of names: anyone who has collected enough bits about a person (postcode, date of birth and sex are enough for most people) holds personal data, names or no names. That is the entropy from the stochastics part of this book in legal dress.

6.1.2 How the GDPR came about

The GDPR did not appear out of nowhere in 2018. Four stages explain why it looks the way it does:

- **The right to informational self-determination** (*informationelle Selbstbestimmung*): the Federal Constitutional Court (Bundesverfassungsgericht) derived it from the general right of personality in its census judgment (*Volkszählungsurteil*) of 15 December 1983 (1 BvR 209/83 and others). Its central sentence is still the best short account of what the protection is for: a social order "in which citizens can no longer know who knows what about them, when, and on what occasion" would be incompatible with the right of self-determination. The occasion was a planned census – which is to say, a data protection problem does not begin with computers but with bringing data together.
- **The Federal Data Protection Act** (*Bundesdatenschutzgesetz*, BDSG): first enacted in 1977, fundamentally recast in 1990 in order to implement the census judgment. It is still in force, but since 2018 only as a supplement to the GDPR, wherever the GDPR leaves the member states room of their own (employee data protection, for example).
- **Directive 95/46/EC** of 24 October 1995: the GDPR's immediate predecessor. It already contained the principles, the legal bases and the rights of the data subject, but as a directive it had to be transposed into national law – with correspondingly different results from one member state to the next. The GDPR repealed it and covered the same ground as a directly applicable regulation.

- **The ePrivacy Directive 2002/58/EC**: special rules for electronic communications, successor to Directive 97/66/EC and amended by 2009/136/EC. From it comes the duty to tell people when information is stored on their devices and to let them refuse – which is what every website today puts on screen as a *cookie banner*. A replacement ePrivacy Regulation has been under negotiation since 2017.

6.1.3 The central terms

- **Scope** (Art. 2 Abs. 1 GDPR – article citations keep the German form the reader will need in order to look them up: *Abs.* is a paragraph, *lit.* a point): the Regulation applies to the processing of personal data wholly or partly by automated means, and to the non-automated processing of such data where they form part of a filing system or are intended to. A card index therefore falls under it; a single sheet of notes does not.
- **Processing**: any operation performed on personal data – from collection through storage and use to erasure.
- **Controller** (*Verantwortlicher*): the body that decides on the purposes and means of the processing.
- **Processor** (*Auftragsverarbeiter*): a third party that processes data on the controller's behalf – an external service provider, say. The processor is bound by the controller's instructions, and the relationship has to be set down in writing in a **data processing agreement** (*Auftragsverarbeitungsvertrag*, AVV) under Art. 28 GDPR: the subject matter and duration of the processing, its nature and purpose, the categories of data subject, and the obligations and rights of both sides.
- **Consent**: a freely given, informed and unambiguous agreement by the data subject to the processing. Freely given also means that there must be no relationship of dependence. A school in Skellefteå had pupils' attendance recorded by facial recognition, with their parents' consent; the Swedish supervisory authority held the consent ineffective because of the imbalance of power between a school and the families (ETid-67, 2019, around €18,600). A nursery's telephone list, by contrast, is perfectly in order once the parents have consented.

An example of controller and processor: an online shop is the controller for its customer data; the parcel service that receives the delivery address and the cloud provider whose servers hold the database are processors. Both need a contract with the shop, and the shop remains the party under the obligation.

6.1.4 The GDPR's basic principles (Art. 5)

1. **Lawfulness, fairness and transparency** (Art. 5 Abs. 1 lit. a) – "lawfully, fairly and in a transparent manner in relation to the data subject"; the German text says "nach Treu und Glauben", in good faith, where the English says "fairly"
2. **Purpose limitation** (lit. b) – use only for "specified, explicit and legitimate purposes"
3. **Data minimisation** (lit. c) – "adequate, relevant and limited to what is necessary"
4. **Accuracy** (lit. d) – "accurate and, where necessary, kept up to date"; inaccurate data are to be rectified or erased
5. **Storage limitation** (lit. e) – in identifiable form only for as long "as is necessary for the purposes …"
6. **Integrity and confidentiality** (lit. f) – protection "against unauthorised or unlawful processing and against accidental loss"

Art. 5 Abs. 2 adds **accountability** to these: the controller must not only ensure that the principles are complied with but also be able to demonstrate it. That is why data protection in practice means so much documentation.

Examples of the principles:

- *Purpose limitation:* asking for a surname in order to generate an AI portrait serves no purpose. Sending someone a discount code on their birthday does need the date of birth – but not as a mandatory field on every order.

- *Data minimisation:* a recommender algorithm needs purchase histories, not addresses; those belong in a separate system, in the extreme case on a machine with no network connection (*air gap*). Less data stored also means less data to leak when something goes wrong.
- *Accuracy:* an incorrect entry at Schufa, the German credit agency, has to be rectified on request; the right to that is Art. 16.
- *Integrity and confidentiality:* the chat platform Knuddels stored passwords in clear text, was hacked, and received a fine of €20,000 – kept deliberately low because the breach was reported and dealt with in exemplary fashion (ETid-25, 2018). AOK Baden-Württemberg, a statutory health insurer, used the data of prize-draw entrants for advertising without a clean check of consent in its systems: €1.24 million (ETid-306, 2020).

6.1.5 Legal bases (Art. 6) and special categories (Art. 9)

- **Legal basis**: processing personal data is lawful only if at least one of the six conditions in Art. 6 Abs. 1 GDPR is met:

  1. consent of the data subject,
  2. performance of a contract with the data subject (or pre-contractual steps at their request),
  3. compliance with a legal obligation,
  4. protection of vital interests,
  5. performance of a task carried out in the public interest or in the exercise of official authority,
  6. legitimate interest (*berechtigtes Interesse*) of the controller or of a third party, unless the interests or fundamental rights of the data subject override it.

  Consent, then, is only *one* of six possible bases. An online shop needs no separate consent for the delivery address, because processing it is necessary to perform the contract – and so is passing it on to the parcel service. Keeping invoices for ten years is a legal obligation under tax law, and on that ground a request for erasure may be refused.

  The most expensive German cases arose from a missing legal basis for data on a company's own staff: at a service centre in Nuremberg, H&M kept and evaluated notes on employees' private circumstances and illnesses – €35.3 million (ETid-405, 2020). notebooksbilliger.de kept its staff under permanent video surveillance; the supervisory authority imposed €10.4 million, which the court cut to €900,000 (ETid-519, 2021). Individuals are held to account too: a police officer who took a witness's telephone number from a case file and used it for private messages paid €1,400 (ETid-50, 2019).

- **Special categories of personal data** (Art. 9 GDPR, colloquially "sensitive data"): data revealing racial or ethnic origin, political opinions, religious or philosophical beliefs or trade union membership, together with genetic data, biometric data for the purpose of uniquely identifying a person, data concerning health, and data concerning sex life or sexual orientation. Processing them is **prohibited in principle** (Art. 9 Abs. 1) – but not absolutely: Art. 9 Abs. 2 sets out an exhaustive list of exceptions, among them explicit consent, obligations arising from employment and social security law, data the data subject has manifestly made public themselves, health care, and scientific research under safeguards. Religious affiliation may therefore be processed if one of these exceptions applies; without such an exception, none of the six legal bases in Art. 6 is enough either.

6.1.6 Privacy by Design and by Default

- **Privacy by Design** (Art. 25 Abs. 1): data protection has to be thought about when a system is conceived, not bolted on afterwards. Three maxims sum up what that means in practice:

  1. What is made technically impossible needs no supervision. A field that does not exist cannot be misused.
  2. Separate business processes functionally and anonymise as early as you can – the later in a process a link to a person survives, the smaller the circle of people who get to see it.

3. Lay out the architecture for privacy, physically as much as electronically: separate systems, separate access rights, separate rooms.

- **Privacy by Default** (Art. 25 Abs. 2): the default settings must be the data-sparing ones. The reason is empirical rather than legal: people often fail, in everyday life, to act on the wish for privacy they profess – out of convenience, out of ignorance, or because in the particular case they weigh things differently. Clicking "accept all" on every cookie banner is not an expression of preference but the path of least resistance. So in practice the default setting decides the outcome.

Examples: defaults must not permit unnecessary collection of data. A cookie banner that arrives with every box already ticked breaches privacy by default. TikTok set children's accounts to public by default and was fined €345 million for it (ETid-2032, 2023).

6.1.7 Data minimisation in practice

Data minimisation (Art. 5 Abs. 1 lit. c; in the old BDSG the same principle went by the name "Datensparsamkeit", data parsimony, § 3a BDSG a.F., in the version then in force) means in practice:

- Collect no irrelevant data
- Do without optional entries (age on an order, say, where it is not needed)
- Say that an entry is voluntary, and default it to "do not collect"

The practical rule for data science projects: for every field holding personal data, ask why you have to store it. If there is no answer, leave it out. If there is one, write it down – that is your legal basis. For everything else, obtain consent.

6.1.8 Further terms and concepts

- **Pseudonymisation**: processing in such a way that the data can no longer be attributed to a person without additional information.
- **Self-generated code** (longitudinal studies): in repeated surveys you want to bring together the answers of the same person across several waves without being able to attribute them to that person. So the respondents build a code themselves out of stable personal attributes – the first and last letters of their parents' first names and the day of their own date of birth, say – and only its hash is stored. Because the same person reproduces the same code, the waves can be linked; because those collecting the data do not know the code, they cannot work backwards from it. The methods and their error rates are well studied Pöge [2011]. It remains a pseudonym all the same: anyone who knows the construction rule and enough about the respondents can run through candidate codes – so hashed data are not automatically anonymous.
- **Rights of the data subject**: Chapter III of the GDPR gives the data subject seven rights that the controller must satisfy on request – as a rule within one month and free of charge:
  - **Access** (Art. 15): which data are being processed, for what purpose, where they came from, who they go to, how long they are stored – along with a copy of the data. That covers video recordings too: Deichmann's Hungarian subsidiary refused a customer the footage from its shop and paid €54,800 (ETid-430, 2020).
  - **Rectification** (Art. 16): correct inaccurate data, complete incomplete data.
  - **Erasure** (Art. 17), also the "right to be forgotten": when the purpose no longer exists, for instance, or consent has been withdrawn.
  - **Restriction of processing** (Art. 18): the middle way, while the accuracy of the data is being checked, say – the data stay stored but may not be used further.
  - **Notification obligation** (Art. 19): the controller must communicate any rectification, erasure or restriction to every recipient the data were disclosed to.
  - **Data portability** (Art. 20): receive your own data in a commonly used machine-readable format and take them with you to another provider.

  - **Objection** (Art. 21): against processing that rests on legitimate interest; against direct marketing at any time and with no balancing of interests at all.

  Added to these is **liability** (Art. 82): anyone who suffers material or non-material damage through an infringement has a claim to compensation. The controller escapes it only by proving that it is not in any way responsible for the damage.

- **Notification of data breaches** (Art. 33): breaches of the protection of personal data are to be reported to the supervisory authority within 72 hours. Twitter reported a breach late and incompletely: €450,000 (ETid-485, 2020); Knuddels (above) reported at once and was treated more leniently for it.

- **Transfers of data to third countries**: permitted only where the level of data protection is adequate. The simplest route is an **adequacy decision** of the Commission under Art. 45: if it finds that a country offers an adequate level of protection, data may be transferred there as if within the EU. For the United States this has been a farce in several acts:

  - **Safe Harbor** (Commission decision of 26 July 2000) – declared invalid by the Court of Justice of the European Union in its judgment in *Schrems I* on 6 October 2015.
  - **EU-US Privacy Shield** (Implementing Decision (EU) 2016/1250) – declared invalid by the Court of Justice in its judgment in *Schrems II* on 16 July 2020, in both cases because US authorities can gain access to the data and data subjects in the EU have no effective remedy against that.
  - **EU-U.S. Data Privacy Framework** (Implementing Decision (EU) 2023/1795, in force since July 2023) – the third attempt, legally contested from the outset. It stands or falls with the independence of the US oversight bodies; as of September 2026 exactly that is in doubt again, and a fresh challenge has been announced. Anyone planning a US cloud today should therefore expect the basis for it to fall away once more. Meta transferred Facebook data to the United States regardless and received the highest GDPR fine so far, €1.2 billion (ETid-1844, 2023). noyb, the organisation Schrems founded, also got the Austrian supervisory authority to find the use of Google Analytics unlawful in 2022 – no fine, but consequences for almost every website. Anyone who puts data into a US cloud needs to know this history.

6.1.9 Fines and enforcement

What the GDPR added to the directive of 1995 is less the principles than the sanctions. It is enforced by independent **supervisory authorities** (Art. 51 ff.) – in Germany one per federal state plus the Federal Commissioner. Their catalogue of **corrective powers** (Art. 58 Abs. 2) reaches well beyond money: warnings about intended processing, reprimands after an infringement, orders to comply with a data subject's requests, orders to rectify or erase, withdrawal of a certification, suspension of transfers to third countries – and, as the sharpest instrument, a temporary or definitive **limitation or ban** on processing. For a data-driven business model that is the real threat; a fine can be paid, a ban on processing cannot.

Administrative fines (Art. 83) are to be "effective, proportionate and dissuasive" and come in two tiers: up to €10 million or 2% of worldwide annual turnover in the preceding year for breaches of the controller's own obligations (Art. 25 privacy by design or Art. 33 notification, say), and up to €20 million or 4% for breaches of the principles, of the rights of data subjects or of the rules on transfers to third countries – whichever amount is the higher. What counts in fixing the amount includes the nature, gravity and duration of the infringement, whether it was intentional or negligent, the remedial measures taken, earlier infringements, and the degree of cooperation with the supervisory authority.

Alongside supervision there is self-regulation: **codes of conduct** (Art. 40) are sector-specific rule-books that associations draw up and have approved by the supervisory authority. Keeping to one makes it easier to discharge the accountability obligation of Art. 5 Abs. 2.

That disposes of two misunderstandings: data protection is neither too complicated to engage with nor free of consequences if you ignore it. At the same time, the small fines do not cost a startup its life. Mercedes paid €3,000 in Spain for a lack of cooperation with the supervisory authority (ETid-885, 2021); a corporation cares about that only because earlier infringements raise the next fine.

Case	Year	Amount	What it was about	Tracker
Meta Platforms Ireland	2023	€1.2 billion	data transferred to the USA with no valid basis	ETid-1844
TikTok	2023	€345 million	children's accounts public by default	ETid-2032
H&M, Nuremberg	2020	€35.3 million	notes on employees' private lives and illnesses	ETid-405
AOK Baden-Württemberg	2020	€1.24 million	prize-draw data used for advertising without clean consent	ETid-306
notebooksbilliger.de	2021	€900,000 (after court)	permanent video surveillance of staff	ETid-519
Twitter	2020	€450,000	data breach reported too late	ETid-485
Deichmann, Hungary	2020	€54,800	access to video footage refused	ETid-430
Knuddels	2018	€20,000	passwords in clear text, exemplary reporting	ETid-25
School in Skellefteå	2019	€18,600	facial recognition with ineffective consent	ETid-67
Police officer, Baden-Württemberg	2019	€1,400	a witness's telephone number used privately	ETid-50
Mercedes, Spain	2021	€3,000	lack of cooperation with the supervisory authority	ETid-885

Some everyday cases with no fine attached that are worth knowing: dashcam footage may be used in evidence, according to the Federal Court of Justice (Bundesgerichtshof, 2018), yet recording the traffic permanently remains unlawful. A circular email to all customers with the recipients visible instead of in BCC is a notifiable data breach. A phone call from "PayPal" asking for login details is not a data protection problem but fraud; data protection and IT security overlap, but they are not the same thing. The Düsseldorf open data portal shows how to publish data without giving people away.

Exercise: the Enforcement Tracker as a dataset

The Enforcement Tracker is itself a table with over 3,000 rows: country, authority, date, amount, company, sector, articles infringed, type of infringement, source. Load the table (the site offers an export; failing that, a small script that reads the data embedded in the page will do) and answer, with the tools from the chapter Pandas: which countries impose the most fines, and which the highest? How has the number of cases per year developed? Which type of infringement is the most frequent, and which the most expensive on average? And is the mean or the median the right measure of location for fine amounts?

Hint

The amounts are extremely skewed to the right (a few cases above €100 million, most of them below €50,000), so the median is the more informative measure of location, and a histogram needs a logarithmic axis. By count Spain leads by a wide margin; by total it is Ireland, because of the

proceedings against the US corporations based there. The most frequent type of infringement is "Insufficient legal basis for data processing" – Art. 6, that is.

6.1.10 The EU's legal framework beyond the GDPR

The GDPR protects personal data. Since 2020 the EU has pursued a second, complementary aim with its **European data strategy** (Commission communication of 19 February 2020): data should be easier to share and use within Europe, under European rules. Out of that has come a series of regulations that apply side by side and are worth keeping apart – roughly along three lines: protection of the data subject (GDPR), rules for markets and platforms (DMA, DSA), access to and sharing of data (DGA, Data Act) – and, cutting across all of them, the AI Act as product regulation for AI systems.

- **AI Act** (*KI-Verordnung*, Regulation (EU) 2024/1689): the first comprehensive legal framework for AI systems, proposed in April 2021 and adopted in 2024. The approach is risk-based – prohibited practices (social scoring by public authorities, for instance), high-risk applications with extensive obligations (medicine, personnel selection, law enforcement), transparency obligations for the rest. Build a model into a product and you yourself become an addressee of this regulation.
- **Digital Markets Act (DMA)** (Regulation (EU) 2022/1925): obligations for very large platforms designated as *gatekeepers* – interoperability of messengers, say, and a ban on favouring one's own services.
- **Digital Services Act (DSA)** (Regulation (EU) 2022/2065): responsibility for content, reporting channels, transparency of recommender systems and of advertising.
- **Data Governance Act (DGA)** (Regulation (EU) 2022/868): a framework for the re-use of protected public sector data, for data intermediation services, and for *data altruism* – voluntarily making your own data available for purposes in the general interest.
- **Data Act** (Regulation (EU) 2023/2854): access to the data that connected devices generate – whoever buys a machine should be able to get at its data too, and to switch supplier.

For everyday work with data the GDPR remains the yardstick; the other regulations decide which data are available at all, and on what conditions a model may be placed on the market.

6.1.11 Further resources

- The GDPR in plain words (in German)
- Full text of the GDPR (EUR-Lex)
- Checklist of technical and organisational measures (BayLDA) (in German)
- Film: *Democracy – Im Rausch der Daten* (bpb) (in German)
- GDPR Enforcement Tracker of the law firm CMS: every fine case named in this chapter, with its date, amount and source (CC BY-NC-SA 4.0)
- Alfred Spector, Peter Norvig, Chris Wiggins, Jeannette M. Wing: *Data Science in Context* Spector et al. [2022] – data science without formulas and without code, but very good on what goes on around the technology; ethics in it is not an appendix but a running theme.

7 Machine learning

7.1 k-means: clustering, implemented from scratch

7.1.1 Where this fits: classification, regression, clustering

Every procedure in this part of the book starts from a table: rows are instances, columns are features. The first question is always whether one column is singled out as the **label**: the quantity we want to predict, and one that is known for the rows we already have (the *ground truth*).

- If there is a label, we speak of **supervised learning**: we learn the relationship between the remaining columns and the label in order to predict it for new rows. If the label is on a nominal scale (species, colour, yes/no), the task is called **classification**; if it is continuous (price, temperature), it is called **regression**. Formally, classification is a special case of regression; in practice it needs one extra idea to turn real-valued predictions into classes (the keyword is *logistic regression*, the central model of the course *Introduction to Deep Learning*).
- If there is no label, we speak of **unsupervised learning**. **Clustering** invents the label: it divides the rows into groups, and the group number is a new column that was not there before. Such an invented, unobserved column is called a **latent variable**. The unsupervised counterpart of regression is *feature engineering*, that is, computing new columns from old ones (the principal components in the chapter Dimensionality reduction, for instance). The most general unsupervised goal is to learn the distribution of the data itself; such a *generative model* can then produce new rows (the chapter Expectation Maximization).

The most important practical difference is the **assessment**: in supervised learning you know the label and can hold back some of the rows and count the predictions on them (*train-test split*, the chapter Model selection). In unsupervised learning there is nothing to count; whether a clustering is "good" is a matter of plausibility, or of a summary measure chosen after the fact.

Task	Label	Example in this book	Chapter
Classification	nominal, known	predicting the iris species from four leaf measurements (example below); recognising items of clothing on Fashion-MNIST	Model selection, Kernel methods
Regression	continuous, known	predicting the price of a house in Melbourne from its location and size	Regression analysis
Clustering	none, it is invented	dividing iris instances into three groups without their species	this chapter, Expectation Maximization

A one-line classifier that also builds the bridge to regression: for the iris data, the rule "petal length under 2.5 cm, then *setosa*, otherwise not" predicts the species *setosa* without a single error. That is a threshold on one feature; logistic regression learns exactly such thresholds, but from several features at once.

Let us check this with the confusion matrix from the chapter Titanic EDA:

```
import numpy as np
from sklearn import datasets

iris = datasets.load_iris()
petal_length = iris.data[:, 2]
truth = (iris.target == 0)            # ground truth: is it setosa?
prediction = (petal_length < 2.5)     # the rule

tp = np.sum( prediction &  truth); fp = np.sum( prediction & ~truth)
fn = np.sum(~prediction &  truth); tn = np.sum(~prediction & ~truth)
print("Confusion matrix [[tn, fp], [fn, tp]] =")
print(np.array([[tn, fp], [fn, tp]]))
print("Accuracy =", (tp + tn) / len(truth))
```

```
Confusion matrix [[tn, fp], [fn, tp]] =
[[100   0]
 [  0  50]]
Accuracy = 1.0
```

Not a single mix-up, accuracy 1; the rule "women survive" on the Titanic only reached 0.79. The iris data are a textbook example, after all.

We begin with clustering, because k-means is the simplest procedure that shows all the building blocks of machine learning: a model with parameters, an optimisation objective and an iterative algorithm.

k-means is an unsupervised learning procedure for dividing a dataset into k clusters. Depending on the shape of the data, k-means may already be enough for a classification or a predictor, or it may be the starting point for more complicated algorithms. Here k is the number of clusters, and it has to be fixed in advance. There are heuristics for finding the 'optimal' k; one of them is to start at $k = 2$ and look at plots of the clusters until you are satisfied.

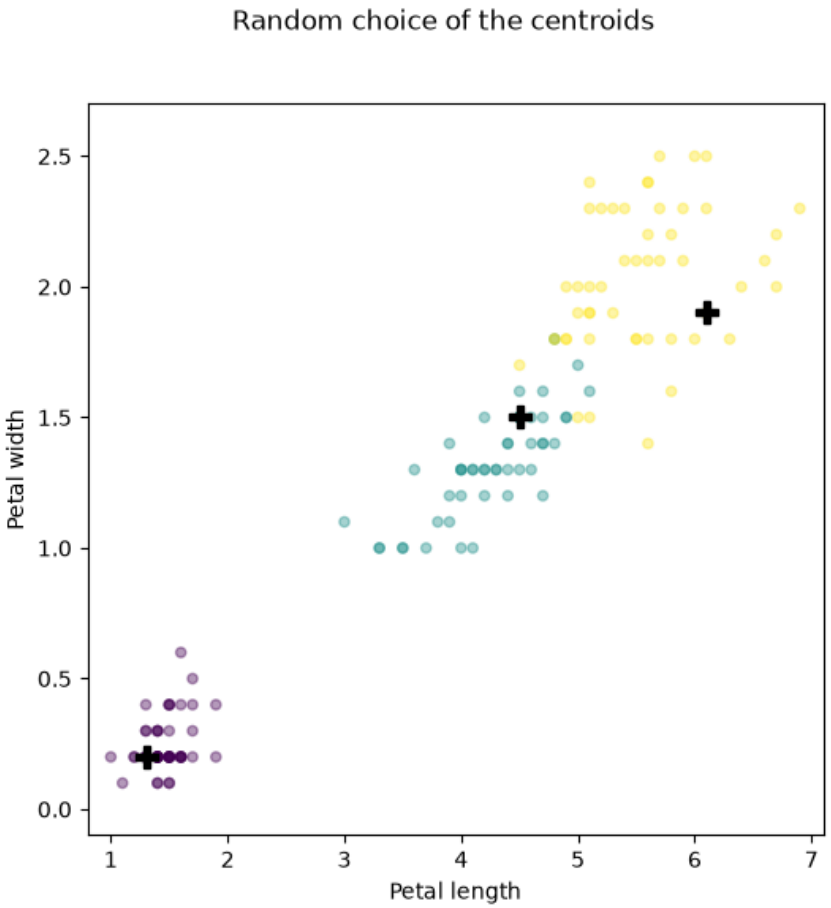
First we go through the idea in pseudocode, taking the petal length and petal width columns of the iris data as the example, that is, in $\mathbb{R}^d$ with $d = 2$:

1. We fix a k, the number of clusters. In the iris example: 3
2. We fix k distinct points $c_1, \ldots, c_k \in \mathbb{R}^d$, the 'centroids'. These can be restricted to points of the dataset, chosen simply by drawing uniformly at random.
3. For every other point $x \in \mathbb{R}^d$ we compute which centroid c_i is nearest and assign x to group i.
4. Then we update the i-th centroid to the mean of all points in group i. If nothing (or little enough) changes in the process, we are done.

7.1.2 One k-means iteration

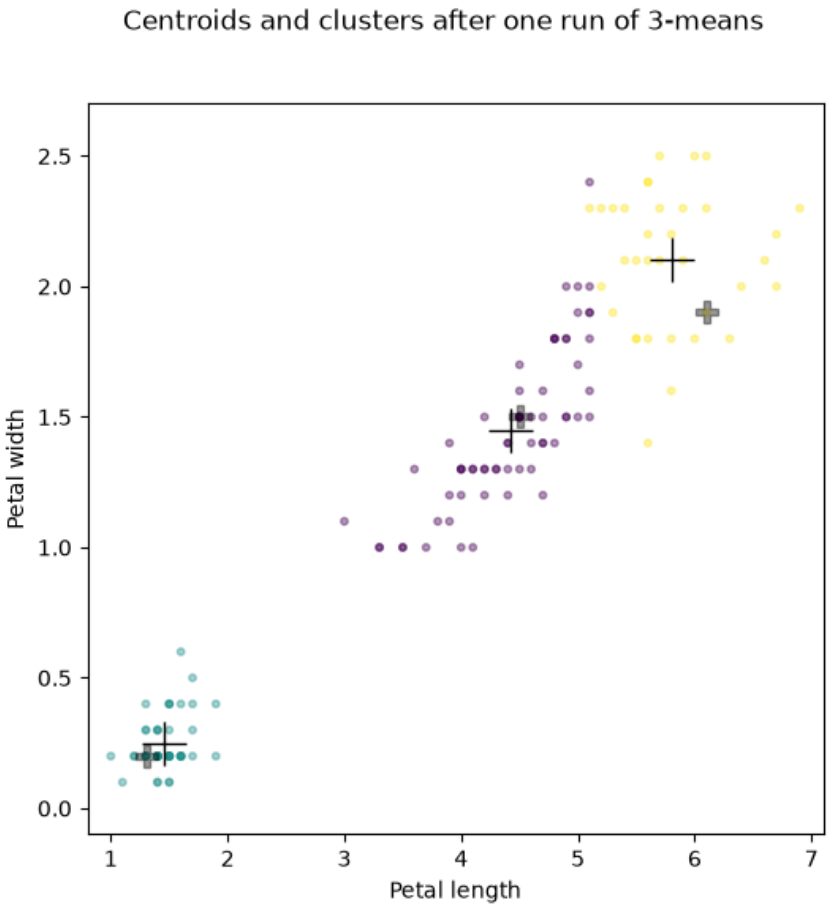
```
import numpy as np
np.random.seed(2222)
import matplotlib.pyplot as plt
from sklearn import datasets

iris = datasets.load_iris()
# permute the data to make it look less sorted by class already:
permutation = np.random.permutation(np.arange(len(iris.data)))
x = iris.data[permutation]
classes = iris.target[permutation]

k = 3
demodata = x[:, 2:4] # petal length, petal width
random_indices = np.random.choice(len(demodata), k)
initial_cs = demodata[random_indices]
cs = np.copy(initial_cs)
print(cs)

[[4.5 1.5]
 [1.3 0.2]
 [6.1 1.9]]

def plotinit(the_data):
    fig, ax = plt.subplots(figsize=(6,6))
    ax.set_xlim(np.min(the_data[:,0]) -0.2, np.max(the_data[:,0])+0.2)
    ax.set_ylim(np.min(the_data[:,1]) -0.2, np.max(the_data[:,1])+0.2)
    ax.set_xlabel("Petal length")
    ax.set_ylabel("Petal width")
    return fig, ax

fig, ax = plotinit(demodata)
ax.scatter(demodata[:,0], demodata[:,1],
           c=classes, vmax=k -1,
           alpha=0.4, s=20)
```

```
ax.plot(cs[:,0], cs[:,1],
        'P', c='black', markersize=10)
fig.suptitle("Random choice of the centroids")
plt.show()
```




```
# We put the class guessed by k -means into a new column:
point_classification = np.full(len(demodata), -1.0)

# We keep a history of the centroids for plotting:
cs_history = np.full((len(demodata),k,len(demodata[0])), -1.0)

# Point by point, everything gets classified:
for i, point in enumerate(demodata):
    # Determine the nearest centroid:
    distances = [np.linalg.norm(centroid - point)
                 for centroid in cs]
    pointclass = np.argmin(distances)
    point_classification[i] = pointclass
    # In this first pass we recompute the
    # centroid immediately, not only
    # once the pass is over:
    cs_history[i] = cs
    cs[pointclass,:] = np.mean(demodata[point_classification == pointclass],
                               axis=0)

# Look at the results:
fig, ax = plotinit(demodata)
ax.scatter(demodata[:,0], demodata[:,1],
           c=point_classification, vmax=k -1,
           alpha=0.4, s=10)
ax.plot(initial_cs[:,0], initial_cs[:,1],
        'P', c='black', markersize=10, alpha=0.4)
```

```
ax.plot(cs[:,0], cs[:,1],
        '+', c='black', markersize=20)
fig.suptitle("Centroids and clusters after one run of 3 -means")
plt.show()
```

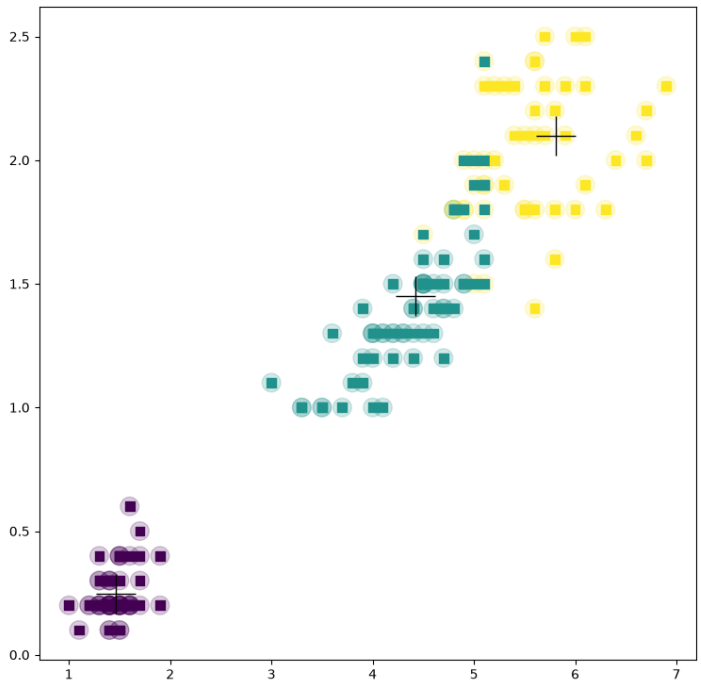



```
# The class names are permuted;
# for a better visualisation we 'repair' that,
# and then the colours match the original classes:
point_classification[point_classification == 0] = 3
point_classification[point_classification == 1] = 0
point_classification[point_classification == 3] = 1
```

You can also watch at leisure how the algorithm clusters the points step by step:

```
import matplotlib.animation as animation
from IPython import display as ipydisplay
import io, base64

fig, ax = plotinit(demodata)
fig.set_size_inches(9, 9)

def init():
    ax.scatter(demodata[:,0], demodata[:,1],
           c=classes, vmax=k -1, alpha=0.2, s=200)

def update(i):
    ax.clear()
    init()
    ax.scatter(demodata[:i+1,0], demodata[:i+1,1],
               vmax=k -1,
               c=point_classification[:i+1],
               marker='s', s=50)
    ax.plot(cs_history[i,:,0], cs_history[i,:,1],
```

```
        '+', c='black', markersize=30)

ani = animation.FuncAnimation(fig, update,
                              frames=len(demodata), interval=100,
                              init_func=init)
video = ani.to_html5_video()
# Also output the last frame as a PNG: the web export shows the video,
# the PDF export (which cannot do HTML) uses the still image.
png = io.BytesIO()
fig.savefig(png, format='png')
ipydisplay.display({'text/html': video,
                    'image/png': base64.b64encode(png.getvalue()).decode('ascii')},
                   raw=True)
plt.close()
```

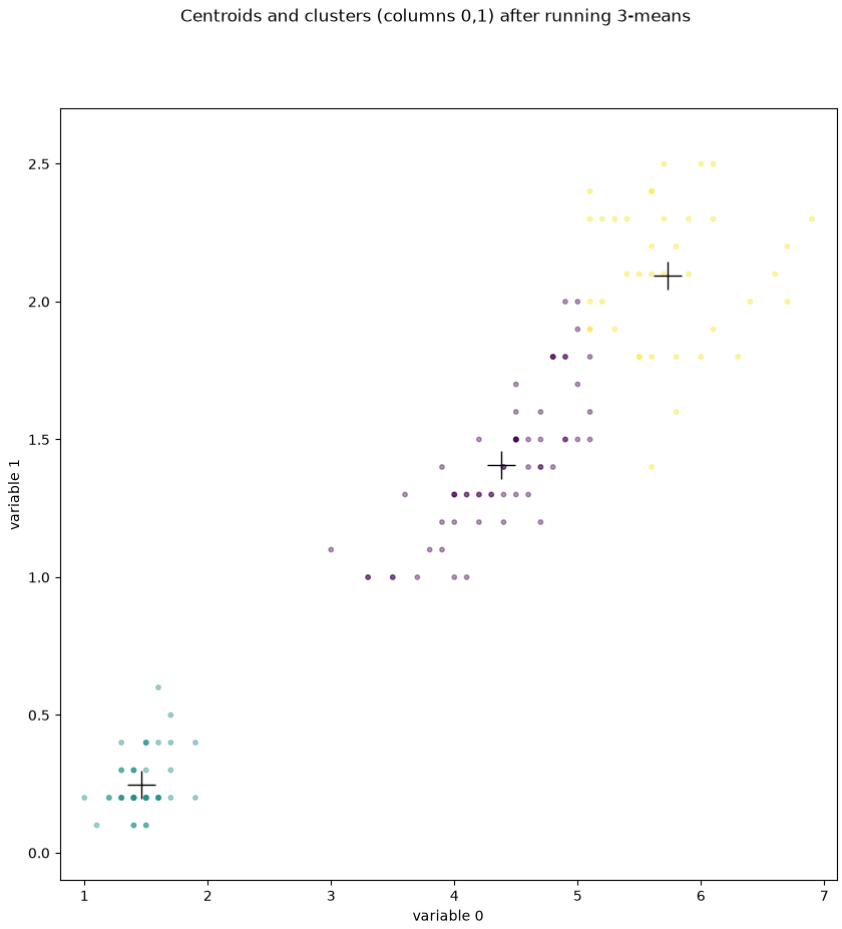


What we see here is that we can run the procedure a second time, with the new centroids as the starting centroids, and get a better clustering. If the clusters (and hence the centroids) stop changing under repeated runs, we speak of *convergence*. Using the full dataset (not just 2 columns) of course gives better clusters. What we want now is a method that performs one k-means pass on an arbitrary dataset with given initial centroids. While we are at it, we can replace the previous implementation by a vectorised one:

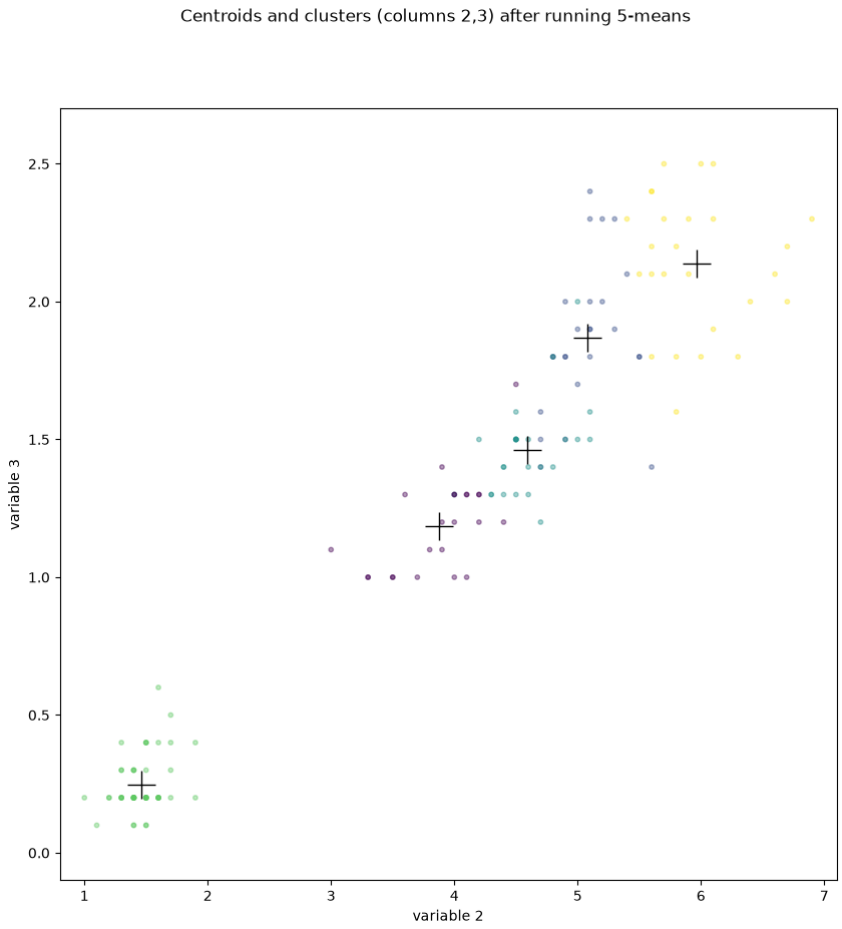
7.1.3 A vectorised k-means implementation

```
def kmeans(data, k, centroids):
    alldistances = np.array([np.linalg.norm(data - centroid, axis=1)
                             for centroid in centroids])
    assert alldistances.shape == (k, len(data))
    point_classification = np.argmin(alldistances, axis=0)
    new_centroids = np.array([np.mean(data[point_classification == pointclass],
                                      axis=0)
                              for pointclass in np.arange(k)])
    return point_classification, new_centroids
```

```
point_classification, new_centroids = kmeans(demodata, 3, cs_history[ -1])

# Look at the results:
def plotinit(data, col1=0, col2=1):
    fig, ax = plt.subplots(figsize=(10,10))
    ax.set_xlim(np.min(data[:,col1]) -0.2, np.max(data[:,col1])+0.2)
    ax.set_ylim(np.min(data[:,col2]) -0.2, np.max(data[:,col2])+0.2)
    ax.set_xlabel("variable %s" % col1)
    ax.set_ylabel("variable %s" % col2)
    return fig, ax

def plot_results(data, k, centroids, classes, col1=0, col2=1):
    fig, ax = plotinit(data, col1=col1, col2=col2)
    ax.scatter(data[:,col1], data[:,col2],
               c=classes, vmax=k -1,
               alpha=0.4, s=10)
    ax.plot(centroids[:,col1], centroids[:,col2],
            '+', c='black', markersize=20)
    fig.suptitle("Centroids and clusters (columns %s,%s) after running %s -means" % (col1, col2,
     k))
    plt.show()

plot_results(demodata, 3, new_centroids, point_classification, col1=0, col2=1)
```



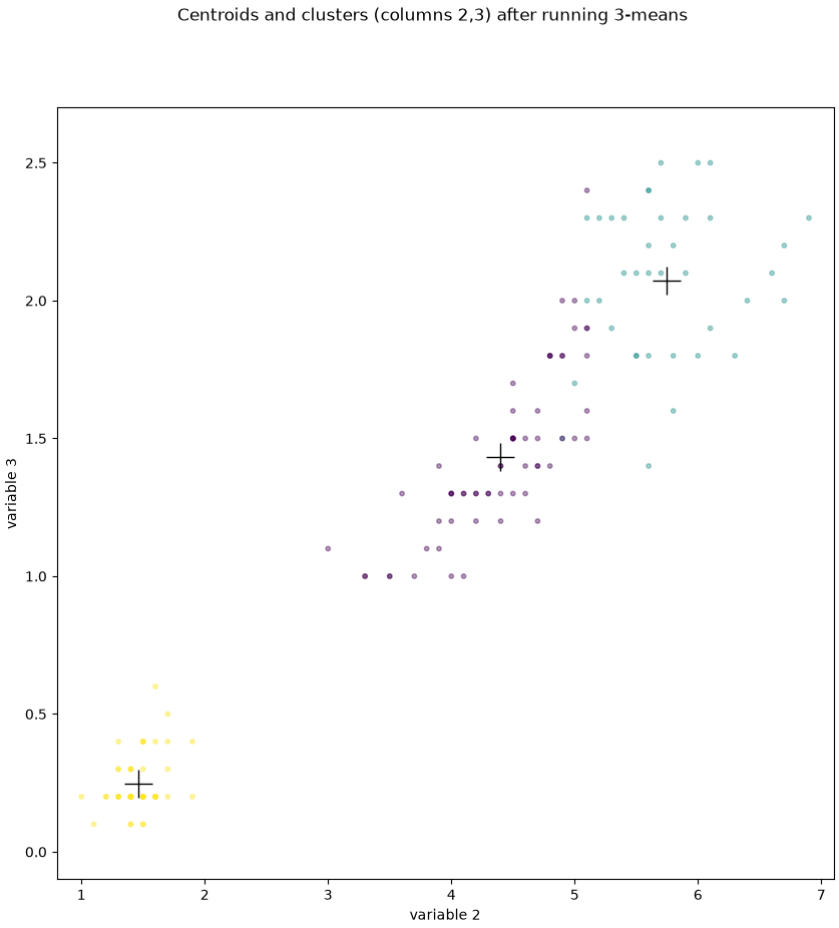

```
# Because it is so easy now:
k = 5
data = x # sepal length, sepal width, petal length, petal width
random_indices = np.random.choice(len(data), k)
initial_centroids = data[random_indices]
point_classification, new_centroids = kmeans(data, k, initial_centroids)
plot_results(data, k, new_centroids, point_classification, col1=2, col2=3)
```



7.1.4 A converging k-means implementation

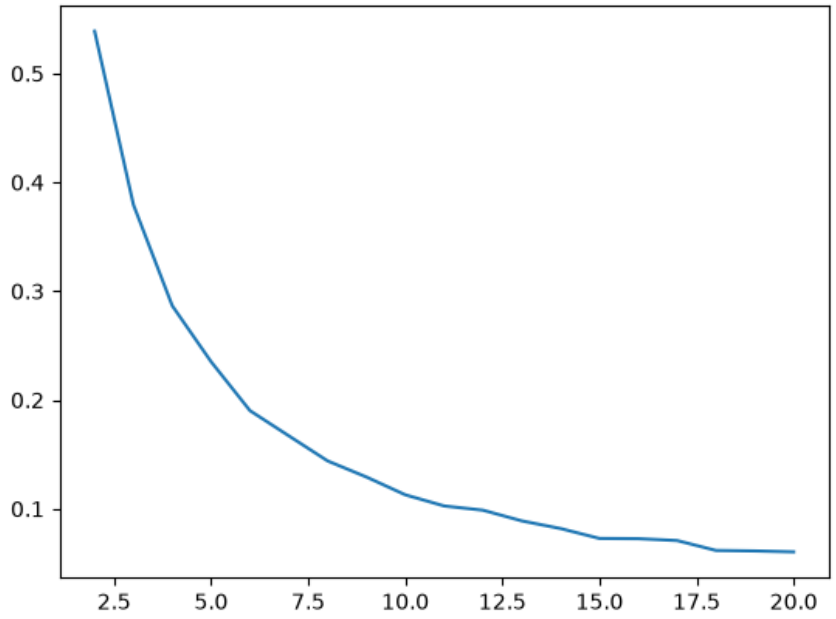
```
np.random.seed(2222)

# 3 -means up to convergence:
def convergent_kmeans(data, k, breakloop=100, noisy=True):
    random_indices = np.random.choice(len(data), k)
    new_centroids = data[random_indices]
    old_centroids = np.full(new_centroids.shape, 0) # will be overwritten anyway
    point_classification = None
    i = 0
    while(not np.allclose(old_centroids, new_centroids, rtol=0.0001) and i < breakloop):
        i += 1
        if(noisy):
            print("iteration step ", i)
        old_centroids = new_centroids
        point_classification, new_centroids = kmeans(data, k, old_centroids)
        # if any of the new centroids is NaN, need to replace it by the old one to continue
        # this can happen if centroid is so bad, the empty cluster forms (=> no mean)
        nan_centroids = np.isnan(new_centroids).any(axis=1)
        new_centroids[nan_centroids] = old_centroids[nan_centroids]
    return point_classification, new_centroids

point_classification, new_centroids = convergent_kmeans(data, 3)
plot_results(data, 3, new_centroids, point_classification, col1=2, col2=3)

iteration step  1
iteration step  2
iteration step  3
iteration step  4
```

```
iteration step  5
iteration step  6
```



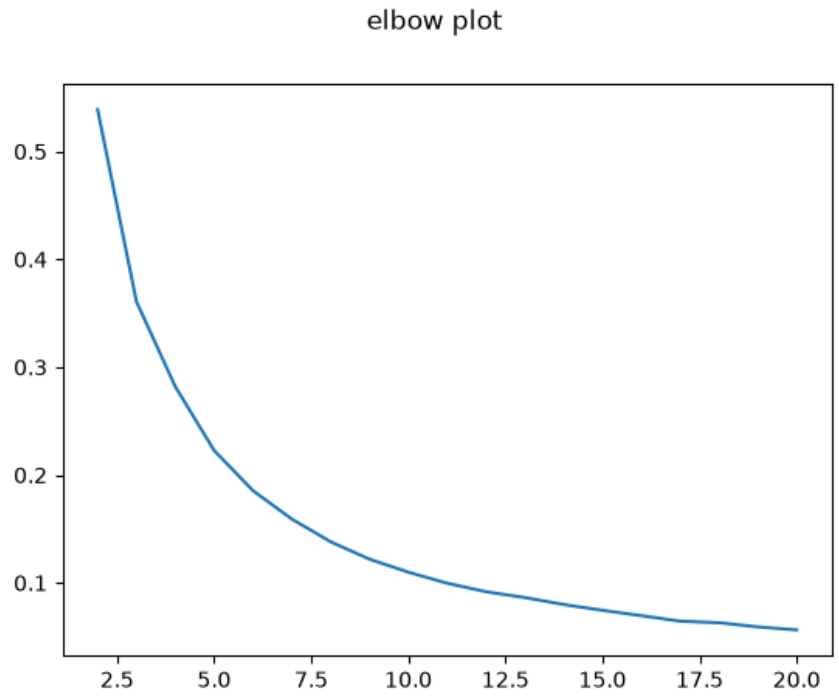
7.1.5 Which k is the right one?

```
np.random.seed(11111)
max_k_to_try = 20
assert max_k_to_try <= len(data)

import warnings

elbow_values = []
with warnings.catch_warnings():        # only for this loop: suppress "Mean of empty slice"
    warnings.simplefilter("ignore")
    for k in range(2, max_k_to_try+1):
        #print("looking at k =", k)
        point_classification, centroids = convergent_kmeans(data, k, noisy=False)
        # Compute the inner variance of every cluster:
        total_inner_variance = np.nansum([np.var(data[point_classification == pointclass])
                                          for pointclass in range(k)])
        group_variance = np.var(centroids)
        elbow_values.append(group_variance / total_inner_variance)

fig, ax = plt.subplots()
ax.plot(np.arange(2, max_k_to_try+1), elbow_values)
plt.show()
```



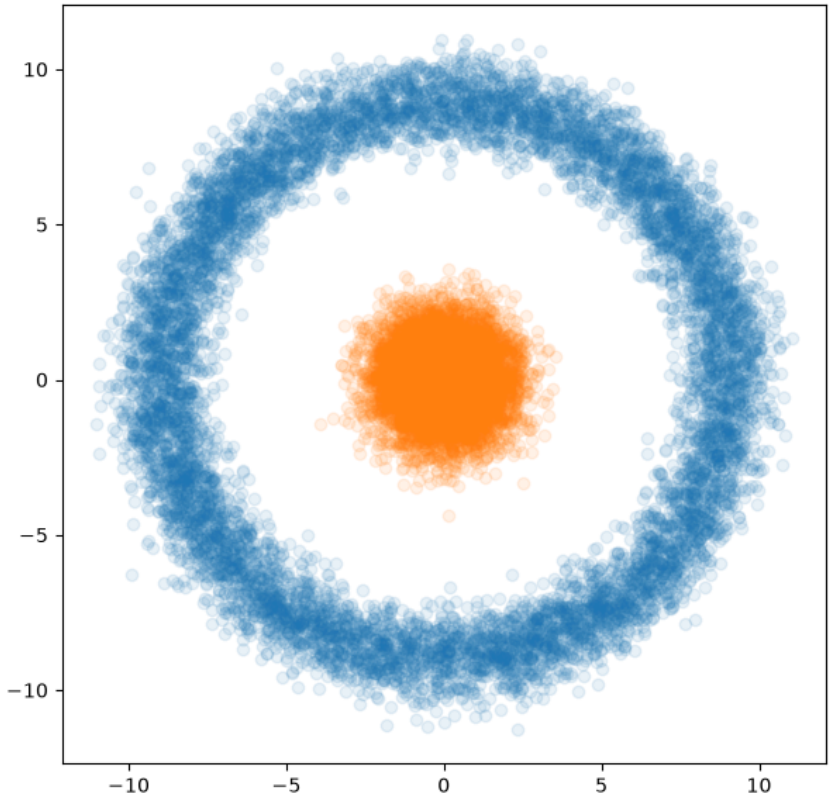
7.1.6 Robust k-means and the elbow plot

```
def robust_kmeans(data, k, howmany=20):
    score = 100000000
    for i in range(howmany):
        point_classification, centroids = convergent_kmeans(data, k, noisy=False)
        total_inner_variance = np.nansum([np.var(data[point_classification == pointclass])
                                          for pointclass in range(k)])
        group_variance = np.var(centroids)
        new_score = group_variance / total_inner_variance
        if(new_score < score):
            final_classification, final_centroids = point_classification, centroids
            score = new_score
    return final_classification, final_centroids, score

with warnings.catch_warnings():        # as above: empty clusters produce "Mean of empty slice"
    warnings.simplefilter("ignore")
    elbow_values = [robust_kmeans(data, k)[2] for k in range(2, max_k_to_try+1)]
fig, ax = plt.subplots()
fig.suptitle("elbow plot")
ax.plot(np.arange(2, max_k_to_try+1), elbow_values)
plt.show()
```

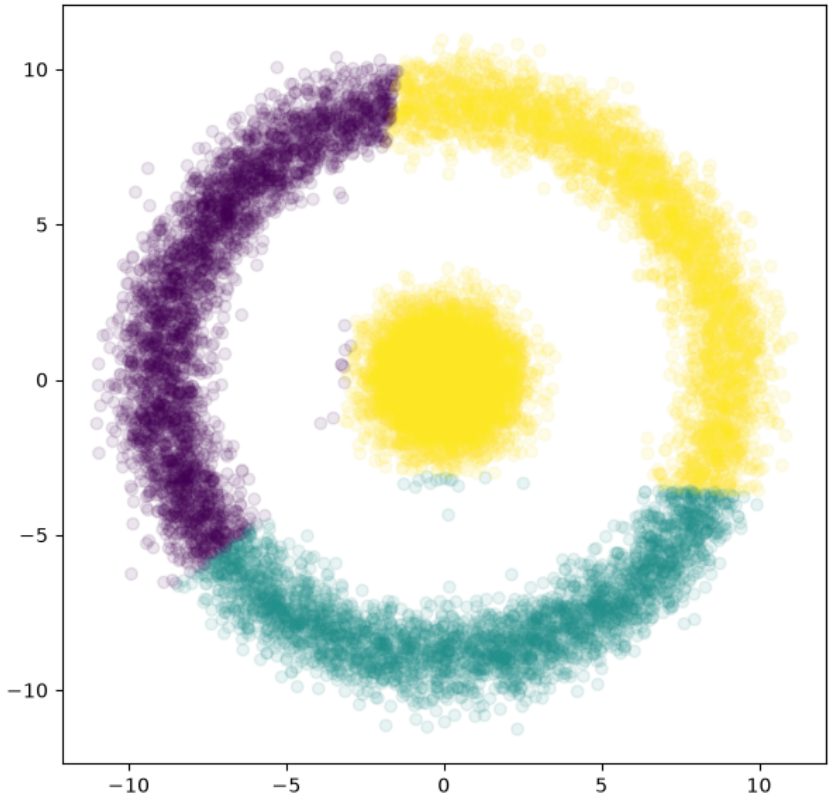


7.1.7 An example with different data

```
import scipy.stats as st

def rings(n, radius=(8, 9)):
    rs = st.multivariate_normal(mean=radius[0]+np.ptp(radius),
                                cov=np.ptp(radius)/2
                               ).rvs(size=n)
    phis = st.uniform.rvs(size=n) * 2 * np.pi
    x, y = np.cos(phis), np.sin(phis)
    return (rs * np.stack([x, y])).T

n = 10000
sample = rings(n)
secondsample = st.multivariate_normal.rvs(mean=[0,0], cov=np.eye(2), size=n)
fig, ax = plt.subplots(figsize=(7,7))
ax.scatter(sample[:,0], sample[:,1], alpha=1000/n)
ax.scatter(secondsample[:,0], secondsample[:,1], alpha=1000/n)
plt.show()
```




```
merged_sample = np.concatenate([sample, secondsample])
final_classification, final_centroids, score = robust_kmeans(merged_sample, 3, howmany=10)
#final_classification, final_centroids = convergent_kmeans(merged_sample, 3, breakloop=100,
    noisy=False)
fig, ax = plt.subplots(figsize=(7,7))
ax.scatter(merged_sample[:,0], merged_sample[:,1], alpha=0.1, c=final_classification)
plt.show()
```



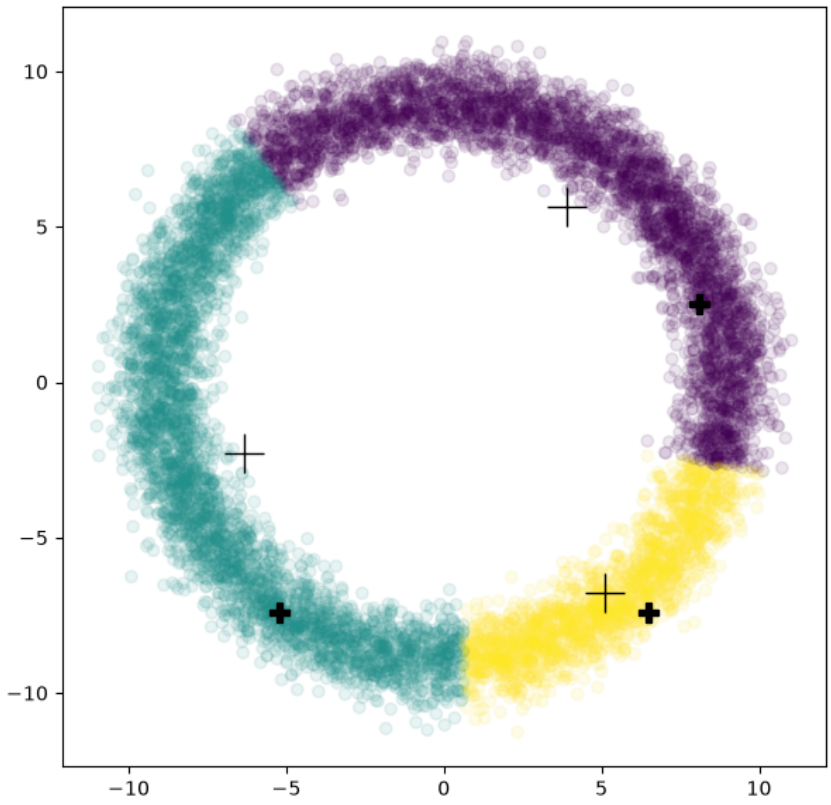
7.1.8 kmeans++

To counteract the problem of bad initial centroids, you can use the **kmeans++** algorithm:

1. Choose a first centroid uniformly at random among all the data points
2. For every point x, compute $D(x)$, the distance to the nearest centroid
3. Choose the next centroid from a weighted distribution that picks each x with weight proportional to $D(x)^2$.
4. Repeat steps 2–3 until k centroids have been found.
5. Ordinary k-means up to convergence

```
def kmeanspp(data, k):
    centroid_indices = [np.random.choice(len(data))] # uniform
    while(len(centroid_indices) < k):
        distances = np.array([np.linalg.norm(data -centroid, axis=1)
                              for centroid in data[centroid_indices]])
        smallest_distances = np.min(distances, axis=0)
        weights = np.power(smallest_distances, 2)
        # must normalize weights to sum up to 1:
        weights = weights / np.sum(weights)
        new_centroid_index = np.random.choice(len(data), p=weights)
        if(not new_centroid_index in centroid_indices):
            centroid_indices += [new_centroid_index]
    return data[centroid_indices]

centroids = kmeanspp(sample, 3)
point_classification, new_centroids = kmeans(sample, 3, centroids)
fig, ax = plt.subplots(figsize=(7,7))
ax.scatter(sample[:,0], sample[:,1], alpha=0.1, c=point_classification)
ax.plot(new_centroids[:,0], new_centroids[:,1],
            '+', c='black', markersize=20)
```

```
ax.plot(centroids[:,0], centroids[:,1],
        'P', c='black', markersize=10)
plt.show()
```



7.1.9 Trivia

A few more things worth knowing about k-means:

- The average time complexity is in $O(k*n*T)$, with n the sample size and T the number of iterations.
- The worst-case complexity is in $O(n^{(k+2/d)})$, where d is the number of variables (the dimension). Arthur, Vassilvitskii: "How slow is the k-means method?", 2006
- What happens in theory? What gets minimised is the so-called **inertia**, the sum of the *squared* distances of all points from their nearest centroids, $J(c_1, \ldots, c_k) = \sum_i \min_j \|x_i - c_j\|^2$. That also points at the biggest problem: when the clusters are not convex and homogeneous, this is a less sensible thing to optimise. The curse of dimensionality bites here too: in high-dimensional spaces distances behave unintuitively – points drawn uniformly from a ball, for example, lie near the boundary more often than near the middle. One more reason to learn about principal component analysis.
- The **bisecting k-means** variant: First 2-means, then one of the largest clusters is split with 2-means, and so on, until there are k clusters altogether.
- The **mini-batch k-means** variant: A compromise between our first variant (the one with the animation) and the last one. It has great advantages for online data processing (streaming) and for large amounts of data in general.
- For k-means procedures to succeed, the data need to fall into clusters of roughly ellipsoidal shape with similar variance. Examples where it can go wrong are plotted in the scikit-learn documentation.
- The **elbow criterion** is not a sensible one. Details and alternatives have been discussed in the literature for years Schubert [2023].

7.1.10 k-means in scikit-learn

There is of course no need to reinvent the wheel: `scikit-learn` comes with k-means implementations of its own:

```
from sklearn.cluster import KMeans
# Much faster, of course, because parallelised in C:
kmeans_result = KMeans(n_clusters=3).fit(iris.data)
print(kmeans_result.cluster_centers_)
print(kmeans_result.labels_)
print(iris.target)
from sklearn import metrics
print("NMI = I(P;C) / mean(H(P), H(C)) between prediction P and class C:",
      metrics.normalized_mutual_info_score(iris.target, kmeans_result.labels_))
```

```
[[6.85384615 3.07692308 5.71538462 2.05384615]
 [5.006      3.428      1.462      0.246     ]
 [5.88360656 2.74098361 4.38852459 1.43442623]]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 0 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 0 0 0 0 2 0 0 0 0
 0 0 2 2 0 0 0 0 2 0 2 0 2 0 0 2 2 0 0 0 0 0 2 0 0 0 0 2 0 0 0 2 0 0 0 2 0
 0 2]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
NMI = I(P;C) / mean(H(P), H(C)) between prediction P and class C: 0.7419116631817836
```

```
# There is also this one:
from sklearn.cluster import MiniBatchKMeans
#from sklearn.cluster import BisectingKMeans # in sklearn v.1.1
```

7.1.11 Where k-means reaches its limits

The assignment step assigns every point to the nearest centroid. That decomposes the whole plane (in general, the whole feature space) into k regions, the **Voronoi cells** of the centroids: the boundary between two cells is the perpendicular bisector of the line segment joining the two centroids, hence a straight line (in higher dimensions, a hyperplane). Every cell is an intersection of half-planes and therefore **convex**. So what k-means finds is always a decomposition into convex pieces with straight boundaries – whatever the data look like.

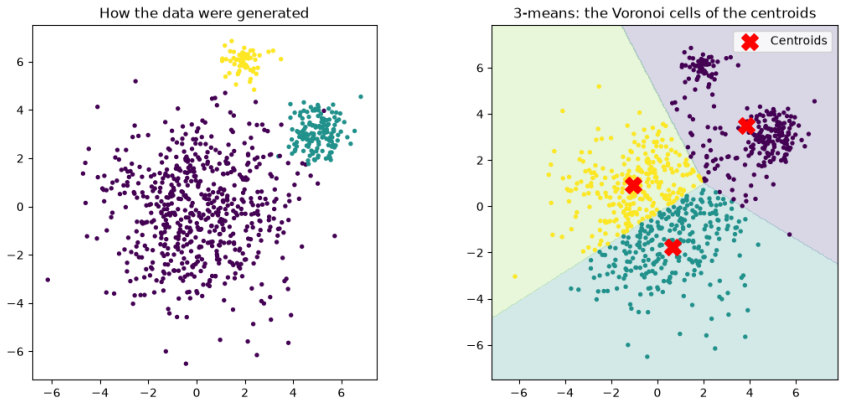
The typical weaknesses that give k-means away all follow from this:

- **Non-convex clusters.** Two rings lying one inside the other, two half-moons, two spirals: every straight line cuts through both, so no choice of centroids can separate them. We saw this above with the rings; the chapter Kernel methods shows the way out: map the data first, so that the clusters become convex.
- **Very different cluster sizes.** A large cluster draws centroids towards itself and gets split up, while small clusters are merged into one – the inertia is counted in points, not in clusters.
- **Very different extent (variance).** The boundary lies halfway between the centroids, not where the broad cluster ends: the narrow cluster steals the outer points of the broad one.
- **Outliers.** A single far-away point shifts a centroid (the mean is not robust), or, with an unlucky initialisation, even gets a centroid of its own.
- **Features on different scales.** The Euclidean distance adds up squares; with one feature in metres and one in millimetres, the second is weighted a thousand times more strongly. Scale first; see Dimensionality reduction.

- **Overlapping clusters.** Where the distributions run into one another there is no sensible hard boundary; a Gaussian mixture model with soft assignments (Expectation-Maximization) is then the better fit.

Add to that what we already know: k has to be chosen in advance, and the result depends on the initialisation. The three blobs below – one large and broad, two small and narrow – show the size problem in a single picture:

```
from sklearn.datasets import make_blobs

# three blobs: different sizes and different widths
X, true_class = make_blobs(n_samples=[600, 150, 60],
                           centers=[[0, 0], [5, 3], [2, 6]],
                           cluster_std=[2.0, 0.6, 0.4],
                           random_state=7)
km = KMeans(n_clusters=3, n_init=10, random_state=7).fit(X)

# Voronoi cells: every point of the plane takes the colour of its nearest centroid
xs, ys = np.meshgrid(np.linspace(X[:, 0].min() - 1, X[:, 0].max() + 1, 300),
                     np.linspace(X[:, 1].min() - 1, X[:, 1].max() + 1, 300))
cells = km.predict(np.c_[xs.ravel(), ys.ravel()]).reshape(xs.shape)

fig, axs = plt.subplots(1, 2, figsize=(13, 5.5))
axs[0].scatter(X[:, 0], X[:, 1], c=true_class, s=8, cmap="viridis")
axs[0].set_title("How the data were generated")
axs[1].contourf(xs, ys, cells, levels=[ -0.5, 0.5, 1.5, 2.5], alpha=0.2, cmap="viridis")
axs[1].scatter(X[:, 0], X[:, 1], c=km.labels_, s=8, cmap="viridis")
axs[1].scatter(km.cluster_centers_[:, 0], km.cluster_centers_[:, 1],
               marker="X", s=200, c="red", label="Centroids")
axs[1].set_title("3 -means: the Voronoi cells of the centroids")
axs[1].legend()
for ax in axs:
    ax.set_aspect("equal")
plt.show()
```



The boundaries are straight lines, the cells convex – and k-means invests two of its three centroids in the large cluster and merges the two small ones into one. That is not a failure of the procedure; it is precisely the decomposition with the smallest inertia: halving 600 points saves more squared distance than separating 210. If your data look like this, you need a procedure that also models the shape and the size of the clusters: a Gaussian mixture model, with its own covariance matrix and its own weight for each component, would have no trouble here.

7.2 Statistical models, estimators and maximum likelihood

In the data science process, the road from the data to a product (be it a report, a prediction or new and interesting data), we model the data. Strictly speaking we model partial aspects, sections of the data (just as the data themselves are only a section of reality). In doing so we may be interested in how the data came about, but we may equally be driven by the wish to be able to say something particular about them, or to do something particular with them. So it helps to be clear about what kind of information one wants to coax out of the data, and then to tune the models to that. Often several approaches are available for computing the same kind of information. The results may well differ, and frequently (in unsupervised learning) we cannot compare the computed results with the "right" result. All that is left then is to compare the information content of the models (to compute the entropy), to examine every model assumption for plausibility, and to develop general criteria for the quality of models. The last of these is what we now set about.

The classical approach – classical in the sense of historically oldest – begins by formalising the models and the aspects one wants to compute from the data with them. One speaks of a **statistical model**, of a **statistic**, and of various **estimators** for a statistic which can then be compared with one another. One criterion is whether an estimator lands near the "right" result at least for very large samples. But hold on – if we do not know the right result, how can we say anything of the sort?

The solution is to restrict ourselves to those statistical models in which we can describe the rough structure of the right result for an estimator well enough that statements such as "in the limit this estimator estimates the expected value of that statistic on the data" become available after all. And then one can talk about how good the values look away from the limit, for somewhat smaller samples. Along the way we meet a notion that matters a great deal to data science and machine learning: the **maximum likelihood** principle.

A very concrete example first. Given data that come as a vector x of length n of real numbers (`float`), of which we know that they arose as a sample of a one-dimensional normally distributed random variable X with expected value $\mu(X)$ and variance $\sigma^2(X)$, we want to estimate the standard deviation $\sigma(X)$. For this the books offer two different formulas, $\sigma(x)$ and $\sigma^*(x)$, which differ only in their denominator:

$$\sigma(x)^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2, \qquad \sigma^*(x)^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\mu)^2 \tag{171}$$

Quite concretely, for a sample of $n = 48$ values that really were drawn from a normal distribution:

```
import numpy as np

x = np.array([6.5281, 3.8003, 4.9575, 7.4818, 6.7351, 1.0454, 4.9002, 2.6973,
              2.7936, 3.8212, 3.2881, 5.9085, 4.5221, 3.2434, 3.8877, 3.6673,
              5.9882, 2.5897, 3.6261, 1.2918, -2.1060, 4.3072, 4.7289, 1.5157,
              7.5395, 0.0913, 3.0915, 2.6256, 6.0656, 5.9387, 3.3099, 3.7563,
              1.2244, -0.9616, 2.3042, 3.3127, 5.4606, 5.4048, 2.2253, 2.3954,
              0.9029, 0.1600, -0.4125, 6.9016, 1.9807, 2.1239, 0.4944, 4.5550])

print(np.mean(x))            # 3.3689...
print(np.var(x))             # 5.0037...  denominator n
print(np.var(x, ddof=1))     # 5.1102...  denominator n -1
print(np.std(x))             # 2.2369...
print(np.std(x, ddof=1))     # 2.2606...
```

So the two formulas deliver noticeably different numbers – a good two per cent apart in the variance, and that at $n = 48$. Which is the right one? And in passing: that we may replace $\mu(X)$ by `mean`(x) at all is something we have so far only taken to be plausible; the justification is the example "the sample mean is unbiased" in the section *Unbiasedness* below.

With these formulas one has to be a little careful when implementing them in practice – because what is μ? In an application I do not know $\mu(X)$ and must therefore use $\mu(x)$. But then it turns out that the information in $\{x_i - \mu(x)\}_{i=1}^{n-1}$ suffices to compute $x_n - \mu(x)$, since by the definition of $\mu(x)$ we have $x_n - \mu(x) = \sum_{i=1}^{n-1}(\mu(x) - x_i)$. In a sense, then, using $\mu(x)$ in place of $\mu(X)$ leaves us one degree of freedom fewer than n. We shall see in a moment how that affects the properties of these two estimators for $\sigma(X)$.

Once statistical models have been defined formally, one can also talk very well about confidence intervals and hypothesis tests, which is what the chapter Statistical tests does. Confidence intervals

extend the bare estimation of a statistic by the information on what margin of error one attaches to that estimate. Hypothesis tests extend it by arriving at decisions in the end. In discussing hypothesis tests it is unavoidable to speak also about the problematic notion of **significance**. But one thing at a time!

As a map for the definitions that follow – on the left what one talks about, on the right what it formally becomes:

in everyday terms	formally
a family of distributions from which we want to pick one	**statistical model** $(\mathcal{X}, P_\theta : \theta \in \Theta)$
the quantity we want to estimate (say μ or σ)	map $\tau\colon \Theta \to \Sigma$
the sample we are holding in our hands	an element $x \in \mathcal{X}$ – or, as long as it has not been drawn yet, a random variable with values in $\mathcal{X}$
the procedure that turns the sample into a number	**statistic**, or **estimator**, $T\colon \mathcal{X} \to \Sigma$

The sample space $\mathcal{X}$ here is not the space of a single measured value but the space of all samples: with n real measurements $\mathcal{X} = \mathbb{R}^n$, so a sample is a single point in a fairly large space.

The crucial move is in the last row. Because the sample is a random variable and the estimation procedure is a map on it, the result of estimating is again a random variable. That lets us speak of its expected value and its variance – and that is exactly what we need in order to be able to say at all what a *good* estimator is: bias, consistency and mean squared error are all formulated in this language.

7.2.1 Statistical models

Definition
A statistical model is a triple $(\mathcal{X}, \mathcal{F}, P_\theta : \theta \in \Theta)$ consisting of a **sample space** $\mathcal{X}$, a σ-algebra $\mathcal{F}$ on $\mathcal{X}$ and a class $\{P_\theta\}_{\theta\in\Theta}$ of probability measures on $(\mathcal{X}, \mathcal{F})$ that contains at least two elements, that is $|\Theta| > 1$. We often simply write $(\mathcal{X}, P_\theta : \theta \in \Theta)$.

Remark
A statistical model is therefore not yet the 'finished model' with which we want to classify or make predictions in the data science sense, but a whole class of models parametrised by the set Θ. This set Θ may be as wild as you like; it should only have at least two elements, so that in the case $\Theta = \{\theta\}$ the notion does not shrink to 'the sample space with one probability measure on it'. When speaking of expected values (and hence, say, of variances) in the context of a statistical model, one has to be careful, and it is best to carry the θ along in the notation: $\mathbb{E}_\theta(Y)$ for the expected value of a random variable $Y : \mathcal{X} \to \mathbb{R}$, because that value depends on θ.

Definition
A statistical model is called a **parametric model** if the parameter set Θ is a subset of some $\mathbb{R}^d$ for a $d \in \mathbb{N}$. For $d = 1$ one also says **one-parameter model**.

If the sample space $\mathcal{X}$ is discrete (at most countable) and $\mathcal{F} = \mathcal{P}(\mathcal{X})$, the model is called **discrete**, and in that case every P_θ has a discrete density p_θ, that is $p_\theta(x) = P_\theta(\{x\})$.

If the sample space $\mathcal{X}$ is a (Borel) measurable subset of $\mathbb{R}^n$ (an open set, say, or the level set of a continuous function), if $\mathcal{F}$ is a restriction of the Borel σ-algebra of $\mathbb{R}^n$, and if moreover every P_θ has a density function f_θ, then the statistical model is called a **continuous model**.

A statistical model that is either discrete or continuous is also called a **standard model**.

Remark
To a first approximation we can picture the sample space as the set of possible outcomes at sample size 1 (the population Ω sends its regards). But: one can also relabel the repeated running of an experiment as really-one-experiment, call a sequence of individual outcomes a single outcome, and so model sample sizes greater than 1. For every statistical model the **product model** can be constructed:

$$(\mathcal{X}^N, \mathcal{F}^{\otimes N}, P_\theta^{\otimes N} : \theta \in \Theta). \tag{172}$$

The projections $X_i \colon \mathcal{X}^N \to \mathcal{X}$ are then independent, identically distributed random variables with respect to every distribution P_θ. The notation $\mathcal{F}^{\otimes N}$ denotes the obvious way of defining a σ-algebra on $\mathcal{X}^N$ (you may ignore it, as usual), and $P_\theta^{\otimes N}$ assigns to an event in $\mathcal{X}^N$ of the form $A_1 \times \cdots \times A_N$ the product $\prod_{i=1}^N P_\theta(A_i)$. With that the N-fold repetition is modelled.

Example
We model the throw of a potentially unfair die with a discrete model: $\mathcal{X} = \{1, \ldots, 6\}$ and $\Theta = \Delta_6$, the set of all possible discrete distributions on a six-element set (written as probability vectors in $\mathbb{R}^6$), with $P_\theta(k) = \theta_k$. Concretely, for $\theta = (\theta_1, \ldots, \theta_6)$ the probability of throwing a 6 in the model with parameter θ is given by $P_\theta(6) = \theta_6$.

Example
The twice-repeated throw of the die we therefore model with $\mathcal{X}^2$ (outcomes are pairs of numbers of pips) and $\Theta = \Delta_6$ (since the second throw is to be modelled as identically distributed to the first). Here $P_\theta^{\otimes 2}(5, 6) = P_\theta(5)P_\theta(6) = \theta_5\theta_6$, as one expects of independent events.

Example
We can also model whether the two throws came up as doubles or not – and so take $\mathcal{X} = \{\text{doubles}, \text{no doubles}\}$ as the sample space. Here we can simply keep the parameter space $\Theta = \Delta_6$; that is one possible statistical model. We do see, though, that $P_\theta^{\otimes 2}(k, k) = \theta_k^2$, so that $P_\theta(\text{doubles}) = \sum_{k=1}^6 \theta_k^2$, and hence the parameter $s = \sum_{k=1}^6 \theta_k^2$ really suffices and we could imagine a smaller parameter space. But then the question arises which distribution P_s we want to assign to the parameter s. One concretely sensible choice is the distribution θ with maximal entropy, that is, a distribution as close to the uniform distribution as possible. This does not become unique, because permutations of $\{1, \ldots, 6\}$ change neither the parameter s nor the entropy $H(\theta)$. If one nevertheless wants to pick a model that is canonical in some way, one can for instance demand an ordering on the θ, that is $\theta_k \geq \theta_{k+1}$. So we see here: a 'too large' parameter space is sometimes the simpler model.

Example
Imagine this. We have found some data x in the wild (read: on the web), now believe that all instances x_i are samples of a random variable X with values in Ω, and so model with $\mathcal{X} = \Omega^n$. If we notice that the values of x are `floats` of some kind, we can look at the histogram. Perhaps we do not get much out of it and cannot really tell whether X is a variable uniformly distributed on $[-1, 1]$, or a normal distribution centred at 0, or even an exponential distribution. In reality that could happen, for instance, when n is very small. The parameter space would then be

$$\Theta = \{\text{uni}\} \times \{0\} \cup \{\text{norm}\} \times \{\sigma > 0\} \cup \{\text{exp}\} \times \{\lambda > 0\}. \tag{173}$$

The first component of Θ is the type of the distribution, {uni, norm, exp}, the second then a possible parameter. Such a model is not a parametric model! We can, though, make one out of it in an unnatural way, by encoding the categorical variable {uni, norm, exp} as the numbers $\{1, 2, 3\}$ and thus representing Θ as a subset of $\mathbb{R}^2$.

Example
$\mathcal{X} = \mathbb{R}^N$ with $\Theta = \mathbb{R} \times \mathbb{R}_{>0}$ and $P_\theta^{\otimes N} = P_{\mu,\sigma}^{\otimes N}$ a univariate normal distribution is the standard model of a sample of size N of a univariate random variable that is normally distributed.

Definition
Let $(\mathcal{X}, P_\theta : \theta \in \Theta)$ be a statistical model and Σ a further event space (that is, a set Σ with a σ-algebra defined on it). Then any random variable $S\colon \mathcal{X} \to \Sigma$ is called a **statistic** of the statistical model.

Remark
As with the notion of a random variable itself, the question arises why we introduce a new name, given that a statistic is nothing but a random variable, which in turn is nothing but a measurable map – in the discrete case, then, statistic = random variable = map. We have seen very plainly by now that we handle random variables differently from maps we call maps. It is exactly the same with a statistic: we use a random variable differently when we use this word for it. It is like an API wrapper around the same data structure, so that one can more easily do different things with it without mixing them up. Just think how `String`s and `float`s are also, in a way, nothing but an API wrapper around bit strings. It works so well that we want to do it again straight away!

7.2.2 Estimators

Definition
Let $(\mathcal{X}, P_\theta : \theta \in \Theta)$ be a statistical model and Σ a further event space. Let $\tau\colon \Theta \to \Sigma$ be an arbitrary map that assigns to every $\theta \in \Theta$ a characteristic quantity $\tau(\theta) \in \Sigma$. Then we call every statistic $T\colon \mathcal{X} \to \Sigma$ an **estimator** (also a **point estimator**) for τ.

Remark
Nobody said that T has to be a sensible, good estimator – in fact T need have nothing to do with τ at first, beyond domain and range having to fit the statistical model and τ respectively. That gives us the flexibility to compare good estimators with miserable ones. The identity is of course admissible as τ, that is $\tau = id\colon \Theta \to \Theta$, $\tau(\theta) = \theta$, and the estimators for τ under consideration are then estimators of the parameter $\theta \in \Theta$.

Example
In the standard model of a univariate normal distribution with sample size N, the projection $\Theta \to \mathbb{R}$ onto the first component, that is onto the expected value μ, is a map τ whose estimators interest us. One possible estimator is the statistic $c\colon \mathbb{R}^N \to \mathbb{R}$ that is constantly 0 (a very conservative estimate). Another possible estimator is the arithmetic mean $\texttt{mean}\colon \mathbb{R}^N \to \mathbb{R}$, for which it really does hold that $\texttt{mean}(x)$ comes arbitrarily close to $\mu(X)$ for large N, where X is the normally distributed random variable that was sampled.

Example
In most statistical models we can consider the map $\sigma\colon \Theta \to \mathbb{R}_{\geq 0}$ that assigns to a parameter θ the standard deviation $\sigma(P_\theta)$ of the distribution P_θ (it only fails when the second moment does not exist, that is when $\mathbb{V}(P_\theta) = \infty$ occurs). For it we know the two estimators σ, $\sigma^*\colon \mathcal{X} \to \mathbb{R}_{\geq 0}$. The two are related by $\sigma^*(x) = \sqrt{\frac{n}{n-1}}\,\sigma(x)$, equivalently $\sigma^*(x)^2 = \frac{n}{n-1}\,\sigma(x)^2$; this factor is called the **Bessel correction** (cf. Descriptive statistics). Which of the two estimators is the better one is settled by the next section.

Definition
An estimator $T\colon \mathcal{X} \to \Sigma$ for $\tau\colon \Theta \to \Sigma$ is called **unbiased** if $\mathbb{E}_\theta(T) = \tau(\theta)$ holds for all $\theta \in \Theta$. If it does not, the estimator is called **biased**, and the difference is the **bias** $\mathbb{B}_\theta(T) := \mathbb{E}_\theta(T) - \tau(\theta)$.

The **mean squared error** of the estimator against the value to be estimated is

$$\mathtt{mse}(T,\theta) := \mathbb{E}_\theta\left((T-\tau(\theta))^2\right) = \mathbb{V}_\theta(T) + (\mathbb{B}_\theta(T))^2 \tag{174}$$

A sequence of estimators $T_n\colon \mathcal{X}^n \to \Sigma$ for τ in n-fold product models is called **consistent** if for $n\to\infty$ the estimator converges to τ in probability: $T_n \xrightarrow{P_\theta} \tau(\theta)$. Convergence in probability means $\forall\epsilon>0 : \lim_{n\to\infty} P(|T_n-\tau(\theta)|>\epsilon) = 0$.

The sequence T_n is called **asymptotically unbiased** if $\lim_{n\to\infty}\mathbb{E}_\theta(T_n) = \tau(\theta)$.

Example
In the standard model of an N-fold repeated univariate normal distribution, the sample mean `mean` is an unbiased estimator for the parameter μ.

Example
For the variance the estimator $\sigma_N^2 = \frac{1}{N}\sum_{i=1}^N (X_i - \mathtt{mean}(X_j))^2$ is not unbiased, since $\mathbb{E}(\sigma_N^2) = \frac{N-1}{N}\sigma^2(X)$. The bias is therefore $-\frac{\sigma^2(X)}{N}$, which vanishes as $N\to\infty$. So the estimator is asymptotically unbiased.

The estimator adjusted by the *Bessel correction*, $\frac{N}{N-1}\sigma_N^2 = \frac{1}{N-1}\sum_{i=1}^N (X_i - \mathtt{mean}(X_j))^2$, is unbiased. So we say: σ_N^2 is asymptotically unbiased, the corrected estimator even unbiased (free of bias).

The bias arises solely from the fact that μ is estimated along with everything else. If the expected value $\mu(X)$ is known, then already

$$\frac{1}{N}\sum_{i=1}^N (X_i - \mu(X))^2 \tag{175}$$

is unbiased, since $\mathbb{E}\left((X_i-\mu(X))^2\right) = \sigma^2(X)$ holds for each individual i and the expected value is linear. Only putting $\mathtt{mean}(x)$ in place of $\mu(X)$ costs the degree of freedom that the introduction to this chapter spoke of – and it is exactly that one degree of freedom which the factor $\frac{N}{N-1}$ wins back.

Definition
An estimator S with finite variance that is unbiased for a function $\tau\colon \Theta\to\mathbb{R}$ to be estimated is called a **uniformly best unbiased estimator** for τ if for every further unbiased estimator T for τ with finite variance and for all $\theta\in\Theta$: $\mathbb{V}_\theta(S) \le \mathbb{V}_\theta(T)$

Unbiasedness Such an estimator therefore has the smallest spread around the value to be estimated among all unbiased estimators.

Note
Looking ahead (not part of the course proper): the Cramér–Rao inequality and the Fisher information in this section are background for why best estimators can exist at all. They are not needed in the course in order to use the results.

In general it is very hard to determine uniformly best unbiased estimators (in general there are none). Looking ahead: the *Cramér–Rao inequality* for *regular* statistical models bounds the variance of an estimator from below by the *Fisher information* (a quantity that can also be computed from the relative entropy), and so lets us recognise uniformly best estimators by the fact that they do not exceed this variance bound. For an unbiased estimator T of a real parameter θ it reads

$$\mathbb{V}_\theta(T) \ge \frac{1}{I(\theta)}, \qquad I(\theta) := \mathbb{E}_\theta\left[\left(\frac{\partial}{\partial\theta}\log p_\theta(X)\right)^2\right], \tag{176}$$

where $I(\theta)$ is the *Fisher information*. An unbiased estimator that attains this bound is therefore a uniformly best estimator. For distributions from a so-called *exponential family* (in a certain sense a common generalisation of the normal distribution, the binomial distribution, the gamma distribution and so on), the uniformly best estimators can be described directly in this way.

For many models that matter in practice, though, there is no closed formula for a good estimator at all – the contaminated normal distribution is the example that still awaits us in the course. What one needs then is a principle that yields, instead of a formula, an optimisation problem whose solution can be approximated numerically. That is exactly what the maximum likelihood principle does, and the iterative procedure that goes with it is the EM algorithm.

Definition
Given a statistical standard model $(\mathcal{X}, P_\theta : \theta \in \Theta)$ with density f_θ for P_θ, the map $\rho\colon \mathcal{X} \times \Theta \to \mathbb{R}_{\geq 0}$ with $\rho(x, \theta) := L_x(\theta) := f_\theta(x)$ is called the **likelihood function**, and for fixed x the map $L_x = \rho(x, \cdot)\colon \Theta \to \mathbb{R}_{\geq 0}$ is called the **likelihood** function for the observed value x.

Maximum likelihood So if we are given data x, the likelihood function for x expresses, for every parameter θ, how the probability density of P_θ comes out at the point x. If the density is large, then x is an observation that is probable under the parameter θ. If the density is small, then x is improbable under θ. As a reminder: if we take the hypothesis $H = (\theta$ is the true parameter) and write $D = x$ (for data), then $P(D|H) = L_x(\theta) = f_\theta(x)$ is the likelihood (cf. Bayes' theorem, version 3, in the chapter Covariance and dependence).

Definition
An estimator $T\colon \mathcal{X} \to \Theta$ for the parameter θ is called a **maximum likelihood estimator** (MLE) if for all $x \in \mathcal{X}$:

$$L_x(T(x)) = \max_{\theta \in \Theta} L_x(\theta), \tag{177}$$

that is, if $T(x)$ maximises the likelihood of x, and does so for every x. One also writes this as

$$T(x) = \operatorname*{argmax}_{\theta \in \Theta} L_x(\theta). \tag{178}$$

We have already worked out a first example of this without using the name: in the chapter The most important discrete distributions we determined, for k successes in n trials, the p under which that observation is most probable, and obtained $\hat{p} = \frac{k}{n}$. In the language of this chapter that is the maximum likelihood estimator in the binomial model $\mathcal{X} = \{0, \ldots, n\}$, $\Theta = [0, 1]$, $P_p = \mathrm{Bin}(n, p)$.

Example
In the N-fold univariate normally distributed model, $x \mapsto (\texttt{mean}(x), \sigma(x))$ is a maximum likelihood estimator for the parameter $(\mu(X), \sigma(X))$. The estimator σ^*, on the other hand, is not a maximum likelihood estimator for $\sigma(X)$. So unbiasedness and the maximum likelihood principle can rule each other out.

Remark
With the help of the relative entropy one can prove that under weak assumptions (Θ an interval, the P_θ pairwise distinct, and a somewhat technical condition) a sequence of maximum likelihood estimators is always consistent.

That means: if one estimates a distribution so that the probability of the observation under the estimated distribution is maximal (the maximum likelihood principle), then the estimate converges in probability to the actual distribution (as the sample size grows).

Remark

So for a family of distributions P_θ with densities f_θ we are interested in maximising the likelihood $f_\theta(x)$ of an observation x in θ. To maximise functions we like to look at the derivative (for which f should be differentiable in θ). Already in the example of the univariate normal density $\phi_{\mu,\sigma}$ we see that the derivative $\frac{d}{d\mu}\phi_{\mu,\sigma}(x) = \frac{(x-\mu)}{\sigma^2}\phi_{\mu,\sigma}(x)$ vanishes at $x = \mu$. So $\hat{\mu} = x$ is the maximum likelihood estimator for μ for a sample of size 1 – which one believes without any calculation: if you observe exactly one value, the normal distribution under which that value is most probable is the one whose expected value sits at exactly that point. That this is a maximum and not a minimum can be seen from the fact that $\phi_{\mu,\sigma}(x)$ tends to 0 as $\mu \to \pm\infty$. In general, though, the likelihood is harder to compute (think of an n-fold product model!), and one can help oneself with a simple trick:

Trick

Instead of maximising the likelihood of x over all possible θ, one can also take the logarithm of the likelihood, the so-called **log-likelihood** $\log \rho(x, \theta)$, and maximise that. Because the logarithm is strictly increasing, likelihood and log-likelihood attain their maxima in the same places.

If, for instance, ρ has the form $e^A e^B$ (as in a 2-fold product model), then $\log \rho$ has the form $A + B$ and setting derivatives to 0 becomes a great deal easier. Even with this trick, however, no closed analytic solution can be found in general – which is where computer science gets to shine with approximation algorithms, such as the *expectation maximization algorithm*, which we take up later in the course.

Example

Maximum likelihood for the exponential distribution. We observe n independent waiting times $x_1, \ldots, x_n \geq 0$ and model them as $\mathrm{Exp}(\lambda)$-distributed, with density $f_\lambda(x) = \lambda e^{-\lambda x}$. Because of the independence the likelihood is the product of the individual densities:

$$L(\lambda) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} = \lambda^n e^{-\lambda \sum_{i=1}^n x_i}. \tag{179}$$

Here the trick pays off at once – the product turns into a sum:

$$\ell(\lambda) := \log L(\lambda) = n \log \lambda - \lambda \sum_{i=1}^{n} x_i. \tag{180}$$

Differentiating and setting equal to 0 gives

$$\ell'(\lambda) = \frac{n}{\lambda} - \sum_{i=1}^{n} x_i \stackrel{!}{=} 0 \qquad \Longleftrightarrow \qquad \hat{\lambda} = \frac{n}{\sum_{i=1}^n x_i} = \frac{1}{\mathtt{mean}(x)}. \tag{181}$$

Since $\ell''(\lambda) = -\frac{n}{\lambda^2} < 0$, this really is a maximum. The result is intuitive again: the exponential distribution has expected value $\frac{1}{\lambda}$, so we estimate $\frac{1}{\lambda}$ by the sample mean. For the observations $1, 2, 3, 4$, say, $\mathtt{mean}(x) = 2.5$ and hence $\hat{\lambda} = 0.4$.

Proposition

If θ_0 is the 'true' parameter for X (the one x was sampled with), then the maximum likelihood estimate $\hat{\theta}$ for a sample of size n converges to the value θ for which the relative entropy between P_{θ_0} and P_θ is minimised (if the model is set up well, that minimum is exactly 0). To see this, one uses the trick just mentioned together with the law of large numbers and the fact that the expected value of the log-likelihood equals the negative cross-entropy $-H(P_{\theta_0}; P_\theta)$.

Example

An application: server log files. From the timestamps of a web application we compute the gaps between two requests, that is, waiting times $x_1, \ldots, x_n$. In order to estimate the load to be expected (for which the mean running time of a request is also needed), we model them as $\mathrm{Exp}(\lambda)$-distributed

and estimate λ as in the example above: $\hat{\lambda} = 1/\texttt{mean}(x)$. So we reckon with a mean waiting time of $\texttt{mean}(x)$ between two requests, and for the number of requests per unit of time with the Poisson distribution for the same λ (chapter The most important discrete distributions).

Example

The k-means algorithm (see the chapter k-Means) can be read as an estimator: the statistical model is $\mathcal{X} = \mathbb{R}^{nN}$, where n is the dimension of the data (so with 4 real variables, as in the Iris dataset, $n = 4$) and N the sample size (for Iris, $N = 150$).

The parameter space is $\mathbb{R}^{nk}$, and the distribution belonging to a parameter $\theta = (c_1, \ldots, c_k) \in \mathbb{R}^{nk}$ (with $c_i \in \mathbb{R}^n$) is a mixture model of multivariate normal distributions with expected value c_i and the $n \times n$ identity matrix as covariance matrix.

The 'perfect' k-means solution, that is the centroids $(c_1, \ldots, c_k)$ that the k-means algorithm tries to determine, are those for which the sum of the squared Euclidean distances from every point x_i of a sample $x \in \mathcal{X}$ to its nearest centroid is minimal. These are exactly the centroids for which the likelihood of the data under the corresponding mixture distribution becomes maximal.

That leads us to an obvious generalisation of the k-means procedure: we could allow more complicated covariance matrices and so push the likelihood up further. The problem that then arises, however, is that no closed form can be determined for the log-likelihood either, and that one has to think up a new algorithm for the estimation!

7.3 Dimensionality reduction: principal component analysis (PCA)

7.3.1 PCA

Principal component analysis (PCA) is an unsupervised method that applies a coordinate transformation to the data so that the new coordinate axes are ordered by how much variance the data show along them. A very common application is to project onto the first (few) coordinates after a PCA, reducing the number of dimensions without throwing away too much of the information in the data.

When somebody says "we do a PCA", they usually mean steps 1–2, often also 1–3:

1. **fit** – compute from the data a linear, invertible coordinate transformation – more precisely, an orthonormal basis. The new coordinate axes are called **principal components** or **principal axes**, and they are ordered by how much variance the data show along them.
2. **transform** – apply the transformation to data. These may well be data other than the ones it was computed from; that is exactly why there are two steps.
3. **dimensionality reduction** – project the transformed data onto the first principal components, that is, onto a low-dimensional linear subspace.

In scikit-learn the first two steps are called exactly that (`PCA().fit(x)`, `pca.transform(x)`, and `fit_transform` for both at once), and step 3 lives in the parameter `n_components`: `PCA(n_components=2)` throws away all but the first two principal components when transforming. Leave `n_components` out and you get nothing but a rotation of the coordinate system – and lose nothing.

Example
If you have data in three-dimensional space whose third coordinate z is always 1, then projecting onto the xy plane changes nothing about the information content of the data. If, on the other hand, the data vary slightly around 1 in the z coordinate but considerably more in the x and y coordinates, then projecting onto the xy plane changes the information content only a little.

The two figures show exactly these two cases: in blue the original data in space, in grey their projections onto the coordinate planes, and as an overlay the difference in total variance before and after the projection onto the xy plane.

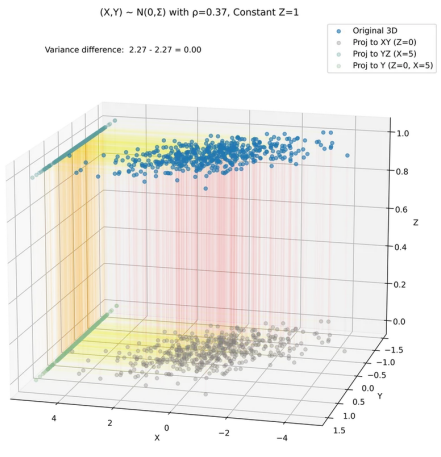


Figure 18: Case (a): z is constantly 1. The projection onto the xy plane has exactly the same total variance as the original data – the third coordinate carries no information.

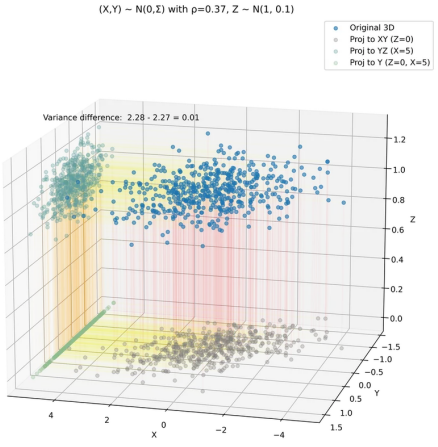


Figure 19: Case (b): $z \sim \mathcal{N}(1,\ 0.1^2)$. Now something is lost in the projection, but only 0.01 out of a total variance of 2.28 – less than half a per cent.

Case (b) is the realistic one: a coordinate is never exactly constant, it just varies very little. PCA makes this quantitative – it says not only *that* you may project, but also how much variance it costs (in scikit-learn, `pca.explained_variance_ratio_`).

A coordinate transformation is nothing but multiplication by an invertible square matrix. Such a matrix can compress or stretch in any direction, rotate about any axis by an angle, or shear. The goal of PCA is to find coordinates in which the first axis, the x axis, carries the highest variance. Recall: if a matrix is read as a linear map by matrix multiplication, the images of the basis vectors are exactly the columns of the matrix. If we can find a vector pointing in the direction of greatest variance, we can write it into a matrix as the first column and project onto the subspace orthogonal to it (that is, we consider all vectors at a right angle to our first matrix column, which leaves a vector space one dimension smaller). Carrying on inductively, we obtain a matrix M. The inverse map M^{-1} is then a transformation that maps the direction of greatest variance onto the first standard basis vector, that is, onto the x axis.

Because we normalise every principal axis and always look for the next one in the orthogonal complement of the previous ones, the columns of M are normalised and pairwise perpendicular: they form an orthonormal basis. Such a matrix is called **orthogonal** (also: orthonormal matrix). It satisfies $M^\top M = I$ (identity matrix), hence $M^{-1} = M^\top$: the back-transformation is simply the transpose, and the inversion comes for free. Why the principal axes come out perpendicular to each other of their own accord is something we shall see at the end of the derivation below.

Unfortunately we have not yet said how to find the direction of greatest variance.

Example

If we look at a sample from a bivariate standard normal distribution, we can draw a circle around the 95% quantile. The matrix $\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ is a stretching along the x axis, and it turns the pretty circle into an elongated ellipse. The direction of greatest variance is then exactly the x axis.

Example

If an arbitrary invertible matrix $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is applied to a sample from a bivariate standard normal distribution, the directions of greatest variance are the directions that M stretches most. For symmetric M this is a vector v with $Mv = \lambda v$ for the largest such λ (for arbitrary M it would be the direction of the largest singular value; we shall see in a moment that the symmetric case is all we need).

Derivation These examples suggest asking which matrix M could have produced the observed data from a standard normal distribution (or at least, for which matrix that would be most plausible). The

eigenvectors belonging to the largest eigenvalue of M are precisely the directions of greatest variance.

Example
If we look at the covariance matrix of a bivariate standard normal distribution, that is, $\begin{pmatrix} \mathbb{V}(X) & Cov(X,Y) \\ Cov(Y,X) & \mathbb{V}(Y) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and then at how the covariance matrix of a sample changes when we carry out the coordinate transformation $\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$, we observe a squaring: in the sample covariance $Cov(x,y) = \frac{1}{n}\sum_{i=1}^{n} x_i y_i$ the scaling by 2 enters only once, but in $\mathbb{V}(x) = Cov(x,x)$ it enters squared. The effect of the coordinate transformation is therefore the "square root" of the covariance matrix after the transformation (mind you, only because we had arranged for the covariance matrix beforehand to be the identity map, that is, the identity matrix).

Such a covariance matrix can always be set up. If we do that with our real data and could then take a "square root" of it, we would have found the matrix M. Fortunately, covariance matrices have a special property: they are positive definite and symmetric. (In the chapter The multivariate normal distribution we saw that the covariance matrix describes the ellipses of the density; the eigenvectors we are now computing are exactly the axes of those ellipses.)

Definition
A square matrix C is called **symmetric** if $C = C^\top$. A symmetric matrix C is called **positive definite** if $v^\top C v > 0$ for all vectors $v \neq 0$. For symmetric matrices this is equivalent to all eigenvalues of C being positive. If only $v^\top C v \geq 0$ holds for all v (equivalently: all eigenvalues are ≥ 0), then C is called **positive semidefinite**.

Remark
If D is the centred data matrix (one row per data point, every column with mean 0), then the empirical covariance matrix is $C = \frac{1}{n} D^\top D$. It is obviously symmetric, and for every vector v we have

$$v^\top C v = \frac{1}{n} v^\top D^\top D v = \frac{1}{n} \|Dv\|^2 \geq 0. \tag{182}$$

Here Dv is the vector of the coordinates of all data points in the direction v; so for a unit vector v, $v^\top C v$ is the variance of the data in the direction v. Hence C is always positive semidefinite, and positive definite exactly when $Dv = 0$ forces $v = 0$, that is, when the columns of D are linearly independent. With real data that is practically always the case (the eigenvalue 0 occurs only when the data do not spread out at all in some direction), which is why we work with "positive definite" in what follows.

Theorem
For every positive definite symmetric matrix C there exists an invertible matrix Σ with $\Sigma^2 = C$.

There is a very simple way to find this matrix Σ if C is a diagonal matrix with nothing but positive entries: then Σ is the diagonal matrix of the respective square roots. If C is not a diagonal matrix, there are procedures for changing coordinates so as to turn C into one. Mathematically this means that C can be replaced by a transformation T backwards, then a diagonal matrix C', then the transformation T forwards. In formulas: $C = TC'T^{-1}$.

Suppose we have found such a decomposition (also called an eigenspace decomposition). Then we can compute Σ' as a square root of C', and $\Sigma = T\Sigma' T^{-1}$ satisfies

$$\Sigma^2 = T\Sigma' T^{-1} T\Sigma' T^{-1} = T\Sigma'^2 T^{-1} = TC'T^{-1} = C. \tag{183}$$

The matrix T we are after satisfies $C = TC'T^{-1}$, or, rearranged, $CT = TC'$. For every column $v = T_{,i}$ of T and the i-th diagonal entry α_i of C' this gives an equation $Cv = v\alpha_i$: the columns of T are

exactly the eigenvectors of C, and the diagonal entries of C' are exactly the eigenvalues of C belonging to them.

Theorem
For every symmetric matrix C there is an orthonormal basis of eigenvectors of C (spectral theorem, without proof). If these eigenvectors are written as the columns of a matrix T, then $T^\top T = I$ and therefore $T^{-1} = T^\top$.

That eigenvectors belonging to different eigenvalues are perpendicular can be checked in one line: if $Cv_1 = \lambda_1 v_1$ and $Cv_2 = \lambda_2 v_2$ with $\lambda_1 \neq \lambda_2$, then, because $C = C^\top$,

$$\lambda_1 v_1^\top v_2 = (Cv_1)^\top v_2 = v_1^\top C v_2 = \lambda_2 v_1^\top v_2, \tag{184}$$

hence $(\lambda_1 - \lambda_2)\, v_1^\top v_2 = 0$ and therefore $v_1^\top v_2 = 0$. If the eigenvectors are normalised as well, then $T^\top T$ is the matrix of all inner products $v_i^\top v_j$, that is, the identity matrix. This is why the "rotation" of PCA (the matrix M from the section on coordinate transformations above) is inverted by mere transposition, $T^{-1} = T^\top$ – which the code below exploits too.

Remark
Let us summarise the derivation. We imagined the data X arising from a standard normal distribution Y by a coordinate transformation, $X = \Sigma Y$; the empirical covariance matrix then estimates $C = \Sigma^2$ (since $\Sigma = \Sigma^\top$, we have $\Sigma\Sigma^\top = \Sigma^2$). The direction of greatest variance is an eigenvector v belonging to the largest eigenvalue λ of Σ. But the square root Σ never needs to be computed for that, because

$$\Sigma v = \lambda v \quad \Rightarrow \quad Cv = \Sigma^2 v = \Sigma(\lambda v) = \lambda^2 v, \tag{185}$$

that is, v is also an eigenvector of C, belonging to the eigenvalue λ^2. So the eigenvectors of Σ and C coincide; only the eigenvalues get squared (that is exactly the "squaring" from the example above), and their order is preserved.

In the end, the normality assumption is not needed at all. For a unit vector v, $v^\top C v$ is the variance of the data in the direction v (see the remark above), and over all unit vectors this expression is maximised precisely by an eigenvector belonging to the largest eigenvalue of C, the maximum being that eigenvalue (Rayleigh quotient, see linear algebra; we do not prove it here). The eigenvector belonging to the largest eigenvalue of the empirical covariance matrix is therefore **always** the direction of greatest variance, for data from any distribution with finite variance.

So we have all the ingredients, provided only that we can somehow find the eigenvectors of the covariance matrix!

A good procedure for finding the eigenvector belonging to the largest eigenvalue: start with a random vector and apply the matrix to it. If it is an eigenvector – good, we note it down. If not, the matrix maps the vector into a direction that, with higher probability, points a little more towards the eigenvector belonging to the largest eigenvalue. So if we apply the matrix repeatedly, the chances are good that we get closer and closer to an eigenvector.

A less efficient procedure is the one you should have learned in linear algebra: first find the eigenvalues as the roots of the characteristic polynomial $\det(M - XId)$, then for every eigenvalue α find the eigenspace $Ker(M - \alpha Id)$ by Gaussian elimination, that is, by solving the equation $Mv = \alpha v$.

Numpy and scikit-learn fall back on LAPACK, which is written in Fortran and carries out a singular value decomposition (the decomposition $C = TC'T^{-1}$) in two steps. We shall not rack our brains over this any further; it belongs in a linear algebra or numerical analysis course. We do, of course, still want the computer's help:

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
from sklearn import datasets

C = np.array([[2.5, 0.5], [0.5, 1.75]])       # for example!
# C is symmetric, so we take eigh: real eigenvalues, guaranteed sorted in ascending order
# (np.linalg.eig guarantees no particular order of the eigenvalues).
```

```
eigenvalues, eigenvectors = np.linalg.eigh(C)
eigenvalues, eigenvectors = eigenvalues[:: -1], eigenvectors[:, :: -1]   # descending, as in the
    SVD
print(eigenvalues)
print(eigenvectors)
T, s, vh = np.linalg.svd(C) # SVD = Singular Value Decomposition; singular values descending
assert np.allclose(s, eigenvalues)
print("in this example the transformation T is:")
print(T)
# Since C is symmetric positive definite, T is orthogonal, so vh = T^{ -1} = T^T
# (eigenvectors are unique only up to sign, which is why eigenvectors and T
#  may differ in the signs of individual columns.)
assert np.allclose(vh, T.T)
```

```
[2.75 1.5 ]
[[ -0.89442719  0.4472136 ]
 [ -0.4472136  -0.89442719]]
in this example the transformation T is:
[[ -0.89442719 -0.4472136 ]
 [ -0.4472136   0.89442719]]
```

The eigenvalues are the variances of the data along the corresponding eigenvectors, the *principal components*. Their share of the total variance $\operatorname{tr}(C) = 2.5 + 1.75 = 4.25$ – here $2.75/4.25 \approx 65\%$ and $1.5/4.25 \approx 35\%$ – is called the *explained variance* of the respective component. This is the quantity you later use to decide how many components to keep.

Dimensionality reduction in five steps From a dataset $x_1, \dots, x_N \in \mathbb{R}^d$, PCA makes a dataset in $\mathbb{R}^k$ with $k < d$:

1. **Centring:** subtract its mean from every feature, so that the dataset has its centre of mass at 0 (usually with scaling as well, see below).
2. Compute the **covariance matrix** C of the centred data.
3. **Eigendecomposition:** determine the eigenvalues $\lambda_1 \geq \dots \geq \lambda_d$ and the corresponding normalised eigenvectors $v_1, \dots, v_d$ of C.
4. **Selecting:** keep the k eigenvectors belonging to the largest eigenvalues – they explain the fraction $(\lambda_1 + \dots + \lambda_k)/\operatorname{tr}(C)$ of the total variance.
5. **Projecting:** map every centred data point x to its new coordinates $\big(\langle x, v_1\rangle, \dots, \langle x, v_k\rangle\big)$.

Example
For $C = \begin{pmatrix} 5 & 3 \\ 3 & 5 \end{pmatrix}$ the eigenvalues are $\lambda_1 = 8$ and $\lambda_2 = 2$ with eigenvectors $v_1 = \frac{1}{\sqrt{2}}(1,1)^\top$ and $v_2 = \frac{1}{\sqrt{2}}(1,-1)^\top$; the first principal component explains $8/10 = 80\,\%$ of the variance. The centred point $x = (2,2)^\top$ has the new coordinates $\langle x, v_1\rangle = \frac{4}{\sqrt{2}} = 2\sqrt{2} \approx 2.83$ and $\langle x, v_2\rangle = 0$. A point on the diagonal has coordinate 0 in the direction v_2 – it lies exactly on the first principal axis, and reducing to one dimension loses nothing for it.

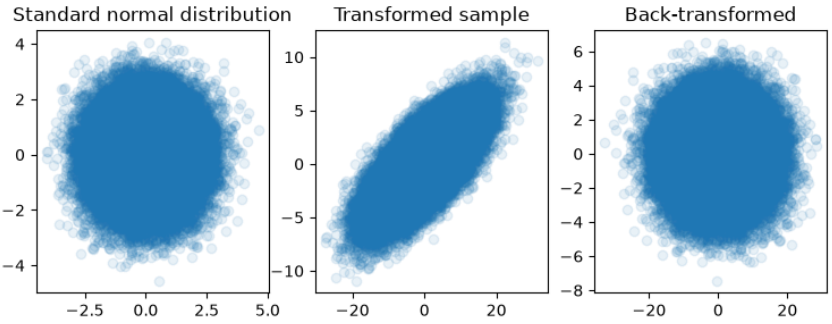
```
C = np.array([[5.0, 3.0], [3.0, 5.0]])
values, vectors = np.linalg.eigh(C)
values, vectors = values[:: -1], vectors[:, :: -1]        # descending
print("eigenvalues:", values, " explained variance of the 1st PC:", values[0] / values.sum())

x = np.array([2.0, 2.0])
new_coordinates = vectors.T @ x                          # step 5: inner products with v_1, v_2
print("new coordinates:", new_coordinates)
assert np.isclose(abs(new_coordinates[0]), 2 * np.sqrt(2)) and np.isclose(new_coordinates[1], 0)
```

```
eigenvalues: [8. 2.]  explained variance of the 1st PC: 0.8
new coordinates: [2.82842712 0.        ]
```

Now, instead of the made-up covariance matrix C, we look at a real sample x, distort it with a coordinate transformation, and then 'undo' that transformation with the method just discussed:

```
fig, axs = plt.subplots(1, 3, figsize=(9, 3))


x = st.multivariate_normal(mean=[0,0], cov=np.eye(2)).rvs(100000)
C = np.cov(x, rowvar=False)
assert np.all(np.isclose(np.eye(2), C, atol=0.1))
axs[0].scatter(x[:,0], x[:,1], alpha=0.1)
axs[0].set_title("Standard normal distribution")

# an arbitrary transformation:
Q = np.array([[6.75, 0.25], [1.75, 1.75]])
new_x = np.matmul(Q, x.T).T
axs[1].scatter(new_x[:,0], new_x[:,1], alpha=0.1)
axs[1].set_title("Transformed sample")
new_C = np.cov(new_x, rowvar=False)
print(new_C)

T, s, Tinv = np.linalg.svd(new_C)
newer_x = np.matmul(Tinv, new_x.T).T
axs[2].scatter(newer_x[:,0], newer_x[:,1], alpha=0.1)
axs[2].set_title("Back -transformed")

plt.show()
```

```
[[45.35696351 12.19917054]
 [12.19917054  6.1101751 ]]
```



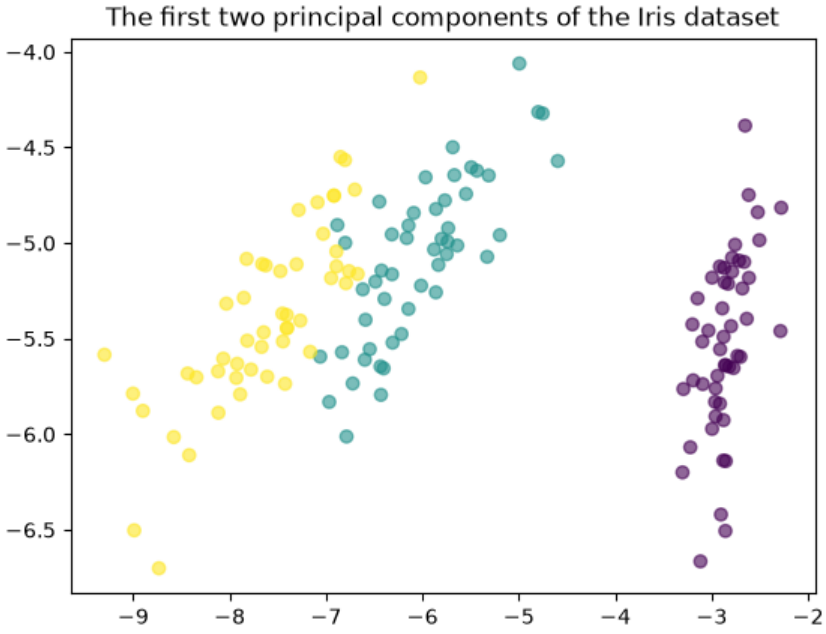
Look at the plots very closely. Have we undone the transformation completely? And either way, how can you tell?

Computation In just the same way we can, of course, proceed with data we did not transform ourselves – data that may never have been sampled from a standard normal distribution in the first place.

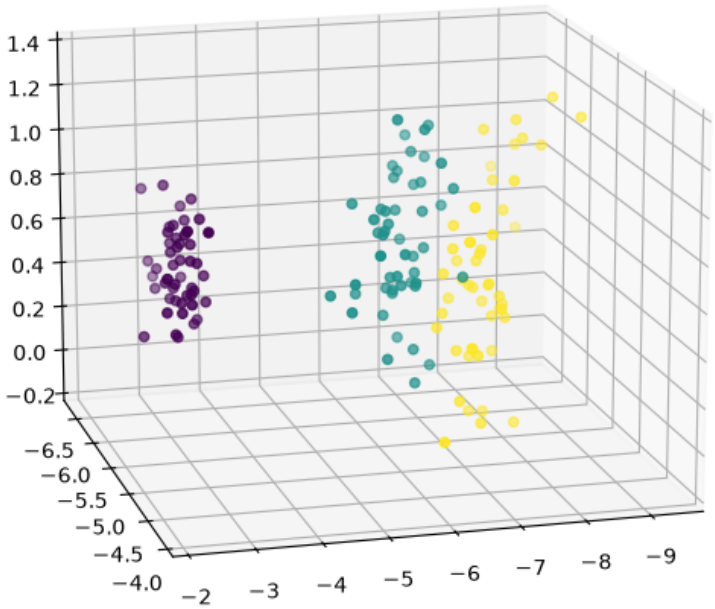
```
iris = datasets.load_iris()
x = iris.data
C = np.cov(x, rowvar=False)
print(C)
T, s, Tinv = np.linalg.svd(C)
pca_x = np.matmul(Tinv, x.T).T

fig, ax = plt.subplots()
ax.scatter(pca_x[:,0], pca_x[:,1], c=iris.target, alpha=0.6)
ax.set_title("The first two principal components of the Iris dataset")
plt.show()

fig = plt.figure(figsize=(7,7))
ax = plt.axes(projection='3d')
```

```
ax.view_init(15, 75)
ax.scatter3D(pca_x[:,0], pca_x[:,1], pca_x[:,2], c=iris.target)
ax.set_title("The first three principal components of the Iris dataset")
plt.show()
```

```
[[ 0.68569351 -0.042434    1.27431544  0.51627069]
 [ -0.042434    0.18997942 -0.32965638 -0.12163937]
 [ 1.27431544 -0.32965638  3.11627785  1.2956094 ]
 [ 0.51627069 -0.12163937  1.2956094   0.58100626]]
```

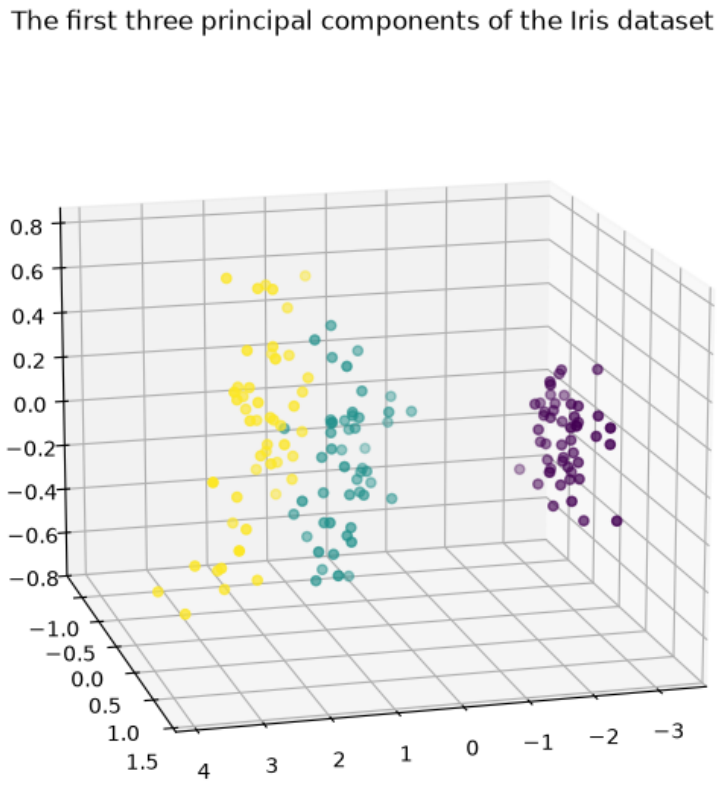


The first three principal components of the Iris dataset



Computation with scikit-learn We do not even have to compute the singular value decomposition or the eigenvectors of the covariance matrix ourselves – scikit-learn can do that for us too:

```
from sklearn.decomposition import PCA

pca_iris = PCA().fit_transform(iris.data)

fig = plt.figure(figsize=(7,7))
ax = plt.axes(projection='3d')
ax.view_init(15, 75)
ax.scatter3D(pca_iris[:,0], pca_iris[:,1], pca_iris[:,2], c=iris.target)
ax.set_title("The first three principal components of the Iris dataset")
plt.show()
```



A completely different approach to finding the direction of maximal variance is to minimise the sum of the squared distances of the points from a straight line.

Up to the factor $\frac{1}{n}$, the sum of the squared distances is a mean squared error. And for the `mse` we already know the best constant model: it is the mean (chapter Expected value and variance, section "Squared deviation"). In this sense the first principal component is the best *linear* model – the straight line through the mean that minimises the mean squared distance – and the animation shows that this is exactly the direction of greatest variance.

One easily missed detail distinguishes PCA from linear regression: here it is the *perpendicular* distance to the line that is minimised, whereas in regression it is the vertical distance, in the direction of the target variable. The two lines are therefore in general different.

Here is a picture of how this works:

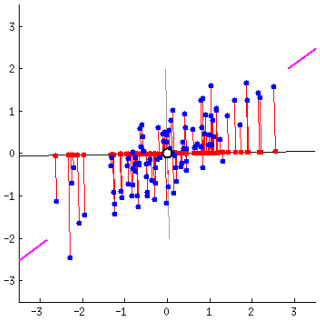
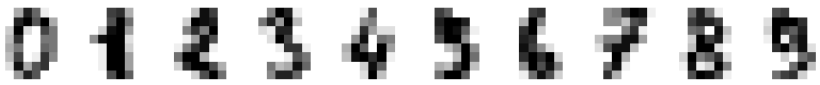


Figure 20: The line rotates until the sum of the squared distances is minimal – and that is the direction of greatest variance.

Application example: MNIST So far we have reduced the dimension only for plotting – but many algorithms need the dimension kept at least small enough that the program does not run for weeks. A dataset with rather more than 4 variables is the MNIST dataset of handwritten digits.

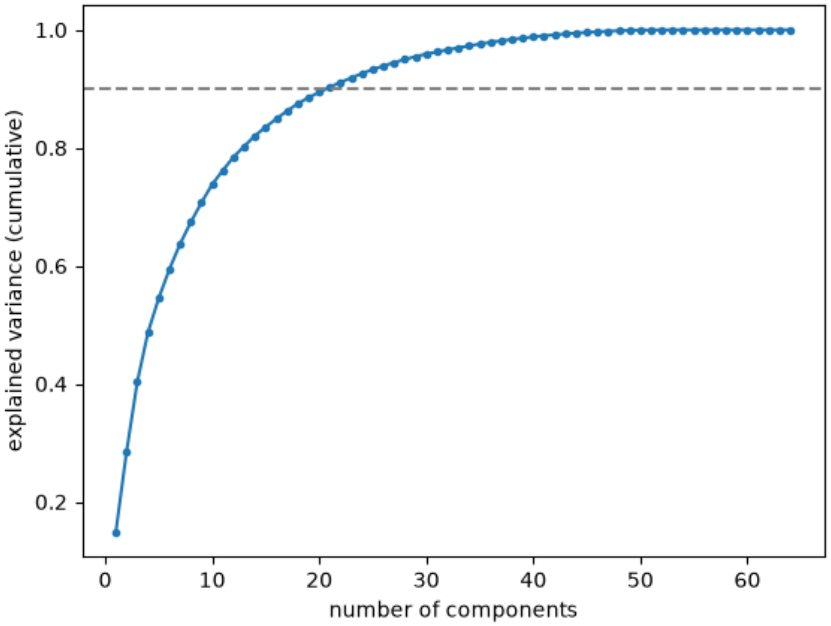
```
digits = datasets.load_digits()
fig, axs = plt.subplots(1,10, figsize=(10,1))
for i in range(10):
    axs[i].axis("off")
    axs[i].matshow(digits.images[i],
                   cmap='gray_r')
plt.show()
x = digits.data
pca_x = PCA().fit_transform(x)

from sklearn.cluster import KMeans
kmeans_result = KMeans(n_clusters=10, n_init=1).fit(x)
kmeans_after_pca = KMeans(n_clusters=10, n_init=1).fit(pca_x)

from sklearn import metrics
print(metrics.normalized_mutual_info_score(digits.target, kmeans_result.labels_))
print(metrics.normalized_mutual_info_score(digits.target, kmeans_after_pca.labels_))
print(metrics.normalized_mutual_info_score(kmeans_result.labels_, kmeans_after_pca.labels_))
```



```
0.7250791394789893
0.7437984495348842
0.8572992273791208
```

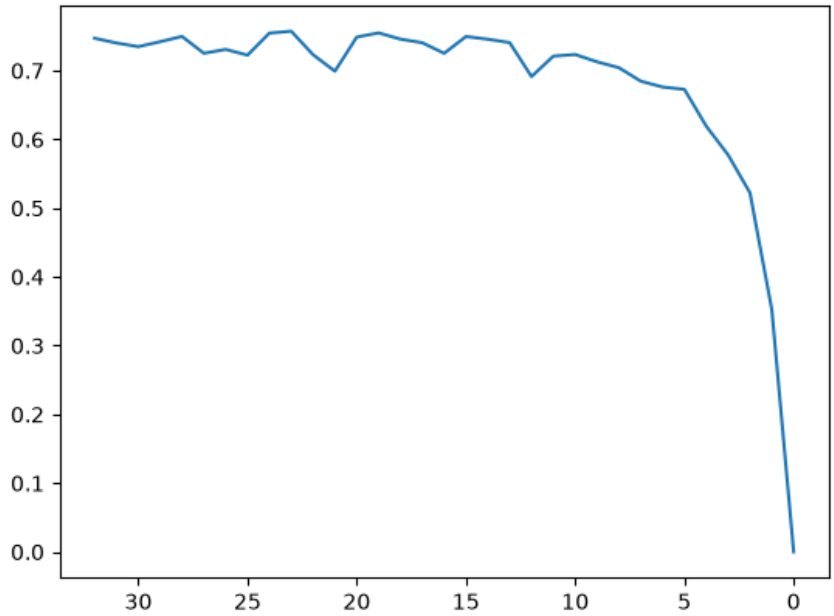
How many components do you need? Scikit-learn gives the explained variance of every component directly (`explained_variance_ratio_`: the eigenvalues of the covariance matrix divided by their sum). The cumulative curve shows how much variance the first n components explain together:

```
pca = PCA().fit(x)
ratio = pca.explained_variance_ratio_
cumulative = np.cumsum(ratio)

fig, ax = plt.subplots()
ax.plot(np.arange(1, len(ratio) + 1), cumulative, marker=".")
ax.axhline(0.9, linestyle=" - -", color="gray")
ax.set_xlabel("number of components")
ax.set_ylabel("explained variance (cumulative)")
plt.show()
print("components for 90% of the variance:", np.searchsorted(cumulative, 0.9) + 1, "out of", len
    (ratio))
```




```
components for 90% of the variance: 21 out of 64
```

```
# Now we throw dimensions away
# and analyse how the classification suffers as a result:
scores = np.full(33, 0.0)
pca_x = PCA().fit_transform(x)

def update_scores(scores, pca_x, maxn, minn):
    for n in range(maxn, minn -1, -1):
        pca_x = pca_x[:,:n]
        kmeans_after_pca = KMeans(n_clusters=10, n_init=1).fit(pca_x)
        scores[n] = metrics.normalized_mutual_info_score(digits.target, kmeans_after_pca.labels_
    )
    return scores, pca_x

# Work in batches to avoid Jupyter cell timeout after 30 seconds:
scores, pca_x = update_scores(scores, pca_x, 32, 17)

scores, pca_x = update_scores(scores, pca_x, 16, 1)

fig, ax = plt.subplots()
ax.plot(np.arange(0, 33), scores)
ax.invert_xaxis()
plt.show()
```



As the graph shows, the performance of 10-means as a classifier only suffers once we are down from the original 64 dimensions to no more than 10 and keep reducing.

In the Iris data we have already seen that the covariance matrices within the groups come out slightly different. Strictly speaking, for an optimal separation of the groups we do not want to favour the directions of greatest variance at all; we would much rather have the direction that separates best. To look at that, however, you have to know the classes already – which makes it supervised learning. One such method is linear discriminant analysis (LDA).

7.3.2 Differences in scale

Things become problematic when the data do not all lie on the same scale, as they did in the examples so far (metres in one place, centimetres in another), or when the scales are even unknown or incompatible (seconds, metres).

Then the variables with large numbers dominate, which is a distortion of the coordinate transformation we were actually after. To avoid this, you rescale every variable by its standard deviation, and likewise shift every variable so that its mean becomes 0. This is called 'standardisation'. It is particularly sensible when the variable is approximately normally distributed.

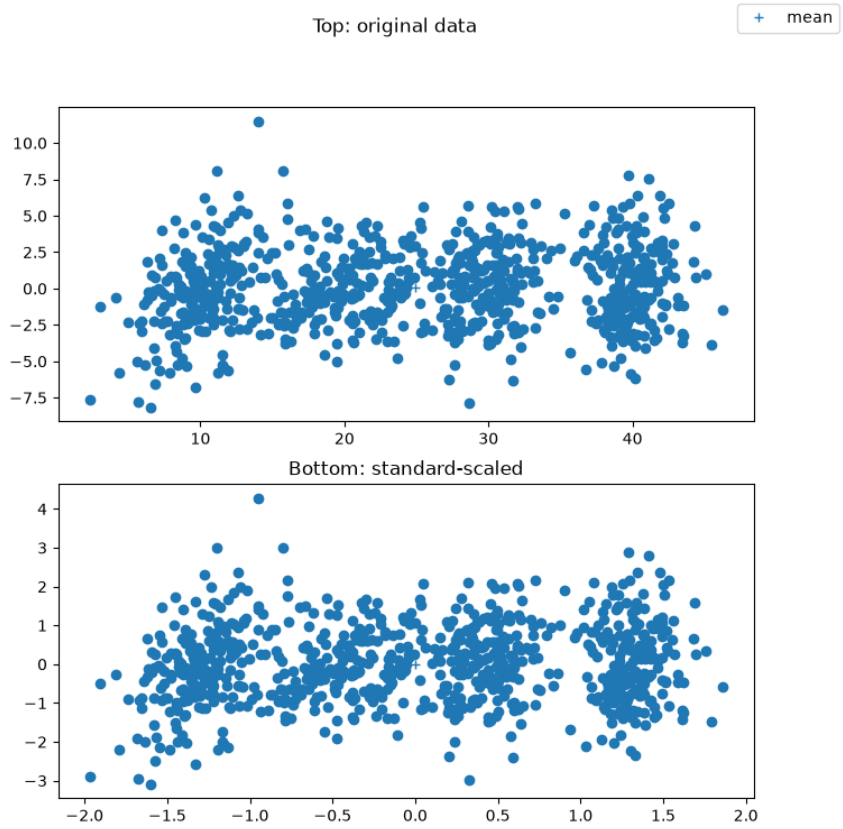
A 'normalisation' is a rescaling onto the interval $[0, 1]$. It is particularly sensible when the variable is not normally distributed.

Any such rescaling is called 'feature scaling'.

In Python, scikit-learn does this with a `Scaler` object:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

np.random.seed(1234)

# craft some dummy data:
dummycenters = [[10, 0.04, 3],
                [20, 0.03, 2],
                [30, 0.02, 1],
                [40, 0.01, 4]]
# make a random symmetric matrix with A@A.T for a random matrix A,
# make it positive definite by adding an identity matrix
dummycovs = (4*(A@A.T + np.eye(3))
             for A in (np.random.rand(3,3)
                       for _ in range(4)))
dummydata = np.concatenate([np.random.multivariate_normal(center,
                                                          cov,
                                                          size=200)
```

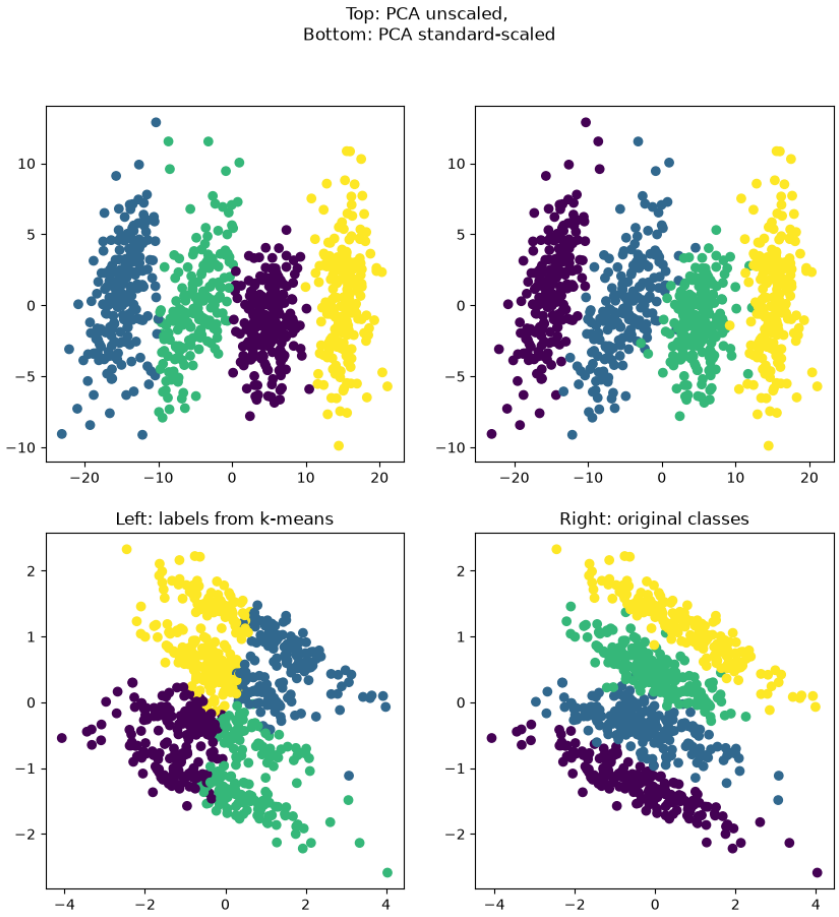
```
            for center, cov in zip(dummycenters, dummycovs)])
dummymu = np.mean(dummydata, axis=0)

# scale it:
scaler = StandardScaler()
scaled = scaler.fit_transform(dummydata)
scaledmu = np.mean(scaled, axis=0)

# let's take a look:
fig, axs = plt.subplots(2, 1, figsize=(8,8))
axs[0].scatter(dummydata[:,0], dummydata[:,1])
axs[0].plot(dummymu[0], dummymu[1], "+",
            label="mean")
axs[1].scatter(scaled[:,0], scaled[:,1])
axs[1].plot(scaledmu[0], scaledmu[1], "+")
axs[1].set_title("Bottom: standard -scaled")
fig.legend()
fig.suptitle("Top: original data")
plt.show()
```



We now look at how scaling affects PCA and the k-means clustering that follows it, because instead of simply putting everything through the StandardScaler every time, you should always look at what difference it makes:

```
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans

dummy_pca = PCA().fit_transform(dummydata)
scaled_pca = PCA().fit_transform(scaled)

dummy_kmeans = KMeans(4, n_init=1).fit(dummy_pca)
scaled_kmeans = KMeans(4, n_init=1).fit(scaled_pca)

# to compare:
original_labels = np.concatenate([np.full(200, k) for k in range(4)])
```

```
# let's take a look:
fig, axs = plt.subplots(2, 2, figsize=(10,10))
axs[0][0].scatter(dummy_pca[:,0], dummy_pca[:,1],
                  c=dummy_kmeans.labels_)
axs[1][0].scatter(scaled_pca[:,0], scaled_pca[:,1],
                  c=scaled_kmeans.labels_)
axs[0][1].scatter(dummy_pca[:,0], dummy_pca[:,1],
                  c=original_labels)
axs[1][1].scatter(scaled_pca[:,0], scaled_pca[:,1],
                  c=original_labels)
axs[1][0].set_title("Left: labels from k -means")
axs[1][1].set_title("Right: original classes")
fig.suptitle("Top: PCA unscaled,\n Bottom: PCA standard -scaled")
plt.show()
```



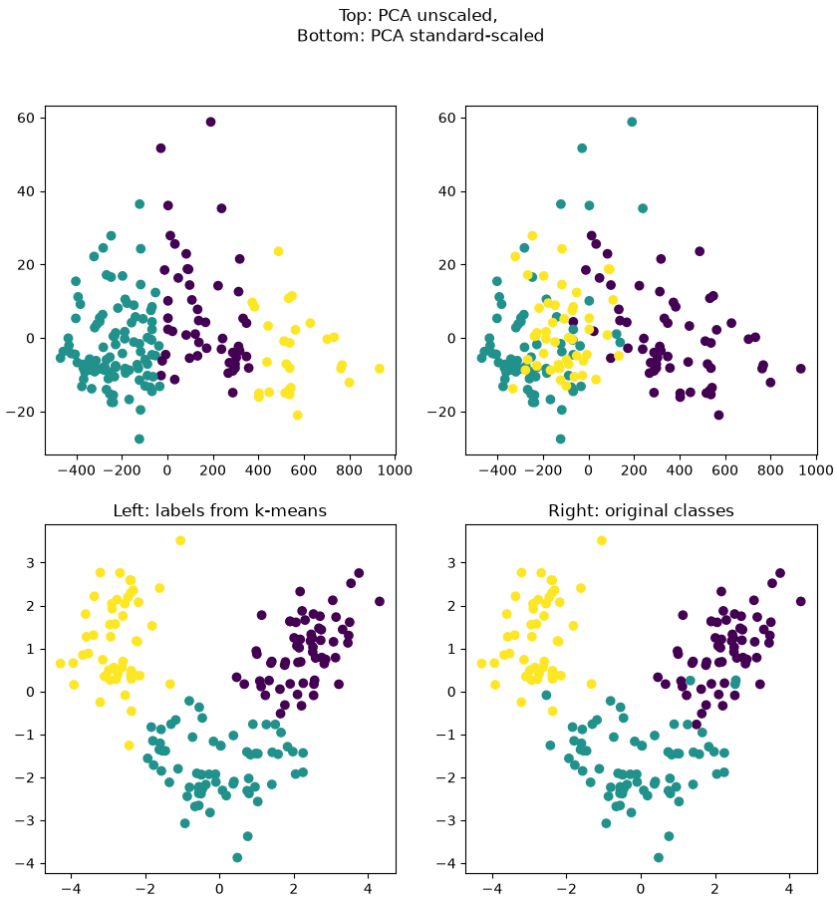
The top row clearly shows that k-means works well on the unscaled data. The bottom row shows that with standard scaling the principal components lie differently (which is why the picture looks rotated), but also that k-means has not divided the clusters up well.

This is ultimately an incarnation of Simpson's paradox: the "right" way to scale the data here would make the clusters as close to standard normally distributed as possible. To achieve that, however, you would first have to cluster, then compute the standard scaling of the clusters, and then scale all the data together by a suitable mean of those. Clustering after that yields a different picture again – and the procedure can be iterated. In this setting it pays not to use k-means at all, but to go straight to a method that does not assume standard normally distributed clusters (Gaussian mixture models, for example).

Application example: Wine Finally we look at a 'success story' for standard scaling:

```
import pandas as pd
from sklearn.datasets import load_wine

df, target = load_wine(return_X_y=True, as_frame=True)
```

```
assert type(df) == pd.DataFrame
assert type(target) == pd.Series

unscaled_pca = PCA(n_components=2).fit_transform(df)
unscaled_kmeans = KMeans(3, n_init=1).fit(unscaled_pca)

scaled = StandardScaler().fit_transform(df)
scaled_pca = PCA(n_components=2).fit_transform(scaled)
scaled_kmeans = KMeans(3, n_init=1).fit(scaled_pca)

fig, axs = plt.subplots(2, 2, figsize=(10,10))
axs[0][0].scatter(unscaled_pca[:,0], unscaled_pca[:,1], c=unscaled_kmeans.labels_)
axs[1][0].scatter(scaled_pca[:,0], scaled_pca[:,1], c=scaled_kmeans.labels_)
axs[0][1].scatter(unscaled_pca[:,0], unscaled_pca[:,1], c=target)
axs[1][1].scatter(scaled_pca[:,0], scaled_pca[:,1], c=target)
axs[1][0].set_title("Left: labels from k -means")
axs[1][1].set_title("Right: original classes")
fig.suptitle("Top: PCA unscaled,\n Bottom: PCA standard -scaled")
plt.show()
```



Rather than going by the picture alone, it is better to compute a summary measure as well. As before, we use an entropy-based measure: the *mutual information* $I(X;Y)$ of the two class variables (chapter Relative entropy), normalised by the mean of their two entropies:

```
from sklearn.metrics import normalized_mutual_info_score

scaled_score = normalized_mutual_info_score(scaled_kmeans.labels_, target)
unscaled_score = normalized_mutual_info_score(unscaled_kmeans.labels_, target)

print("The scaled version, with a similarity of %f to the 'true' class, is better than the
    unscaled version with a similarity of only %f" % (scaled_score, unscaled_score))
```

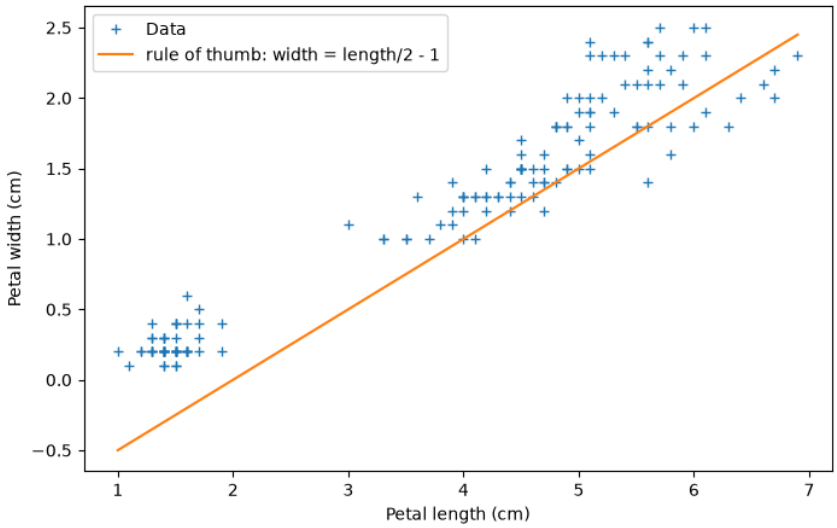
```
The scaled version, with a similarity of 0.865642 to the 'true' class, is better than the
    unscaled version with a similarity of only 0.423309
```

7.4 Regression analysis: least squares and regularisation

Once a dataset has been prepared well enough (missing data replaced, suitably rescaled, new features defined), you can venture into a regression analysis. A regression describes the relationship between two variables. One of them is treated as depending on the other, and the dependence is made explicit as a functional relationship – a mathematical map that computes, approximately, the values of the one variable from the values of the other.

An example, before we fix the vocabulary. In the Iris dataset we suspect a relationship between the length and the width of a petal: longer petals ought to be wider as well. We look at the cloud of points and draw a straight line through it – at first entirely by eye.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

iris_data = datasets.load_iris().data
petal_length, petal_width = iris_data[:, 2], iris_data[:, 3]

fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(petal_length, petal_width, "+", label="Data")
line = np.linspace(petal_length.min(), petal_length.max(), 2)
ax.plot(line, 0.5 * line - 1, label="rule of thumb: width = length/2 - 1")
ax.set_xlabel("Petal length (cm)")
ax.set_ylabel("Petal width (cm)")
ax.legend()
plt.show()
```



What we read off is the rule of thumb

```
predicted_petal_width = petal_length * 0.5 - 1
```

and with that, the example already contains every ingredient of a regression:

1. There is a **function** that computes the target variable from the explanatory variable – here $x \mapsto 0.5\,x - 1$.
2. The function has **parameters**, here the numbers 0.5 and -1, and we obtained them from the data, not from theory.
3. The prediction is **not exact**. For every single flower something is left over, and we have to keep track of it.

The third point is why a regression is not simply a function. And the second immediately raises the question of whether 0.5 and -1 really are the best numbers – the rest of this chapter answers it. (To anticipate: they are not. The method of least squares finds 0.416 and -0.363 and pushes the mean squared error down from 0.167 to 0.042.)

A **regression model** consists of:

- **regressors** x_i, that is, a (k-variate) 'independent' variable, the *explanatory variable* (here k is the number of regressors, not the number of clusters as in k-means)
- a **regressand** y_i, that is, a (univariate) 'dependent' variable, the *target variable*
- **parameters** $\beta_1, \ldots, \beta_k$
- **error terms** ϵ_i
- $Y_i = f(X_i, \beta) + \epsilon_i$

For concrete data (x_i, y_i) and a given regression model, $y_i - f(x_i, \beta)$ is also called **the i-th residual**.

Some fixed notation will help here. A sample consists of **instances** $x_1, \ldots, x_N$ together with the associated values of the target variable $y_1, \ldots, y_N$, which in machine learning are called **labels**. If the regressor is multidimensional we write x_{ij} for the value of the j-th feature at the i-th instance – the first index counts the rows of the table, the second the columns. The model's **prediction** is

$$y_i^{pred} := f(x_i; \beta), \tag{186}$$

so it depends on the estimated parameters and changes with them. The residuals are then exactly the differences $y_i - y_i^{pred}$, and so they are all we ever see of the error terms: the error term ϵ is a random variable we never observe; the residuals are its instances, once we have committed to a β. That is why you check a regression model by looking at its residuals – they ought to look like a sample from the distribution you assumed for ϵ.

A regression analysis consists of setting up a regression model and then estimating the function f that fits the data best – the one with the smallest residuals.

Particular regression methods differ in the kind of function f they permit. In linear regression, f is linear in the parameters β.

Example
For simplicity let $k = 1$, that is, let X_i be a univariate variable. If we choose the class of constant functions for f, then $f(X_i, \beta) = \beta_1$ and we are looking for the β_1 with $Y_i = \beta_1 + \epsilon_i$, that is, a β_1 that minimises the distance $d(\beta_1, y_i) = y_i - \beta_1$ across all the values y_i. This happens when $\texttt{mse}(y, \beta_1) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_1)^2$ is minimal, that is, for $\beta_1 = \texttt{mean}(y)$.

Example
k-means can be seen as a regression method: the parameters β describe the positions of the centroids, and the function f picks the class by the nearest centroid. The k-means method itself is the search for an approximation to the optimal parameters β.

In k-means the error terms are assumed to be normally distributed and uncorrelated.

Example
Every classification method is a regression in which the dependent variable is the class and the regressors are all the data used for the classification. This should not be taken literally, though: a linear regression directly on class numbers $0, 1, 2, \ldots$ is not a sensible classifier, because the numbers are not values on a scale. Classification methods such as *logistic regression* therefore interpose a link function that maps the range onto class probabilities.

7.4.1 Least squares

Definition
Minimising the **residual sum of squares** $\sum_{i=1}^{N}\left(Y_i - f(X_i, \beta)\right)^2$ in a regression model yields an estimate for β known as the **least squares estimator**.

The least squares estimator is a maximum likelihood estimator if the error terms are assumed to be normally distributed (the model for this is at the end of the section *Linear regression*).

Example
If, for points (x_i, y_i) in the plane, we compute a straight line that minimises the total distance to all the points (a so-called line of best fit), then this is exactly a regression with $f(x, \beta) = \beta_0 + \beta_1 x$. The regression equation $Y_i = f(X_i, \beta)$ becomes the equation of a line $y_i = \beta_0 + \beta_1 x_i$ in which β_0 is the intercept on the y-axis (at $x_i = 0$) and β_1 is the slope of the line.

The least squares method now estimates β_0 and β_1 so that $\sum_{i=1}^{N} (y_i - \beta_1 x_i - \beta_0)^2$ is minimised.

If $\beta_0 = 0$, the minimisation problem is one we can solve. We differentiate with respect to β_1 and look for zeros:

$$\begin{aligned} 0 &= \sum_{i=1}^{N} 2(y_i - \beta_1 x_i)(-x_i) \\ &= \sum_{i=1}^{N} 2\left(\beta_1 x_i^2 - y_i x_i\right) \\ &= 2\left(\beta_1 \sum_{i=1}^{N} x_i^2 - \sum_{i=1}^{N} x_i y_i\right) \end{aligned} \tag{187}$$

and so $\beta_1 = \dfrac{\sum_{i=1}^{N} x_i y_i}{\sum_{i=1}^{N} x_i^2}$.

In general, β_0 can be computed with the formula

$$\beta_0 = \texttt{mean}(y) - \beta_1 \texttt{mean}(x). \tag{188}$$

So to determine β_1 first in the general case, you shift the data by their means, replacing x_i by $x_i' = x_i - \texttt{mean}(x)$ and y_i by $y_i' = y_i - \texttt{mean}(y)$. That yields the formula

$$\beta_1 = \frac{\sum_{i=1}^{N} (x_i - \texttt{mean}(x))(y_i - \texttt{mean}(y))}{\sum_{i=1}^{N} (x_i - \texttt{mean}(x))^2} \tag{189}$$

Divide numerator and denominator by N and you are looking at two quantities we already know: on top, the empirical covariance of x and y; underneath, the empirical variance of x (both in the n-version, see the remark *For data* in the chapter Covariance). The slope of the line of best fit is therefore

$$\beta_1 = \frac{\mathrm{Cov}(x, y)}{\sigma^2(x)}, \qquad \beta_0 = \texttt{mean}(y) - \beta_1 \, \texttt{mean}(x). \tag{190}$$

This is the form that is quickest to compute by hand, and it also says something: the slope is the covariance, normalised by the spread of the regressor. Compare it with the correlation coefficient $\rho = \mathrm{Cov}(x, y)/(\sigma(x)\sigma(y))$ from the chapter Covariance and you get $\beta_1 = \rho\, \sigma(y)/\sigma(x)$ – correlation and regression slope differ only by the ratio of the two standard deviations.

Example
For the four points $x = (-2, -1, 1, 2)$ and $y = (-3, -1, 1, 3)$ both means are 0, so

$$\sigma^2(x) = \tfrac{1}{4}(4 + 1 + 1 + 4) = 2.5, \qquad \mathrm{Cov}(x, y) = \tfrac{1}{4}(6 + 1 + 1 + 6) = 3.5, \tag{191}$$

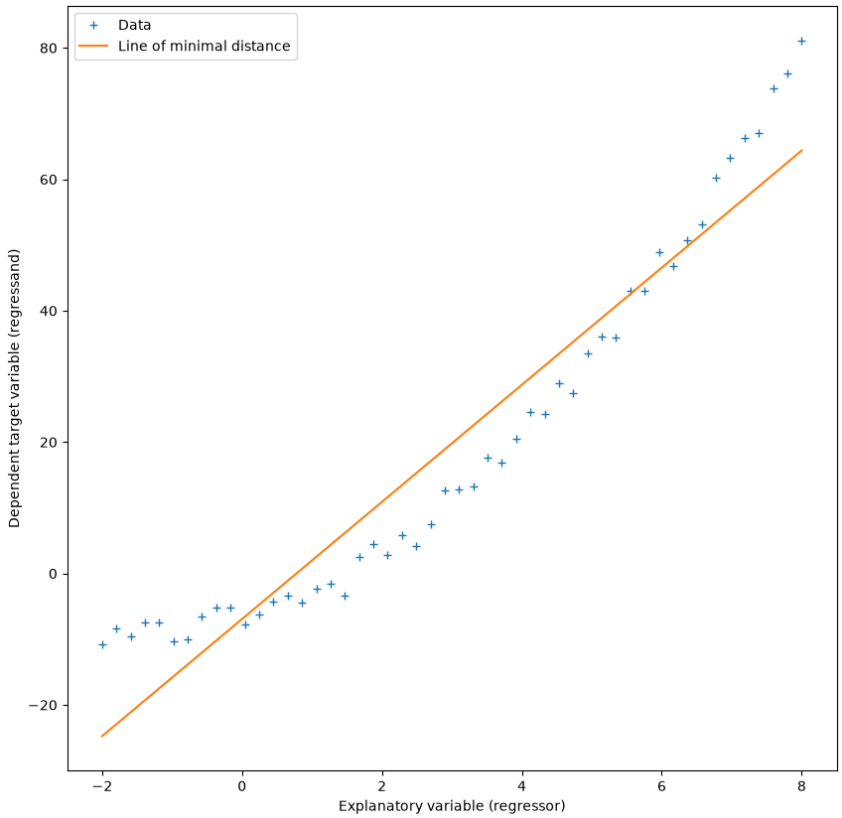
and therefore $\beta_1 = 3.5/2.5 = 1.4$ and $\beta_0 = 0 - 1.4 \cdot 0 = 0$. The line of best fit is $y = 1.4\, x$.

```
import numpy as np
np.random.seed(1234)

# generate synthetic data:
x = np.linspace( -2, 8)
y = x**2 + 3*x - 7 + np.random.uniform( -3, 3, size=x.shape)

# compute estimate from before:
```

```
x_centered = x - np.mean(x)
y_centered = y - np.mean(y)
beta_1 = ( x_centered.T @ y_centered ) / sum(np.square(x_centered))
beta_0 = np.mean(y) - beta_1*np.mean(x)
# the same number as covariance over variance (ddof=0: the n -version, as in np.var):
assert np.isclose(beta_1, np.cov(x, y, ddof=0)[0, 1] / np.var(x))
print("line equation is y = %f*x + %f" % (beta_1, beta_0))
```

```
line equation is y = 8.912806*x + -6.929693
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(10,10))
ax.plot(x, y, "+", label="Data")
ax.plot(x, beta_0 + x*beta_1, label="Line of minimal distance")
ax.set_xlabel("Explanatory variable (regressor)")
ax.set_ylabel("Dependent target variable (regressand)")
ax.legend()
plt.show()
```



The method of least squares can also be run directly in Scipy (so there is no need to remember the formulas or the derivation):

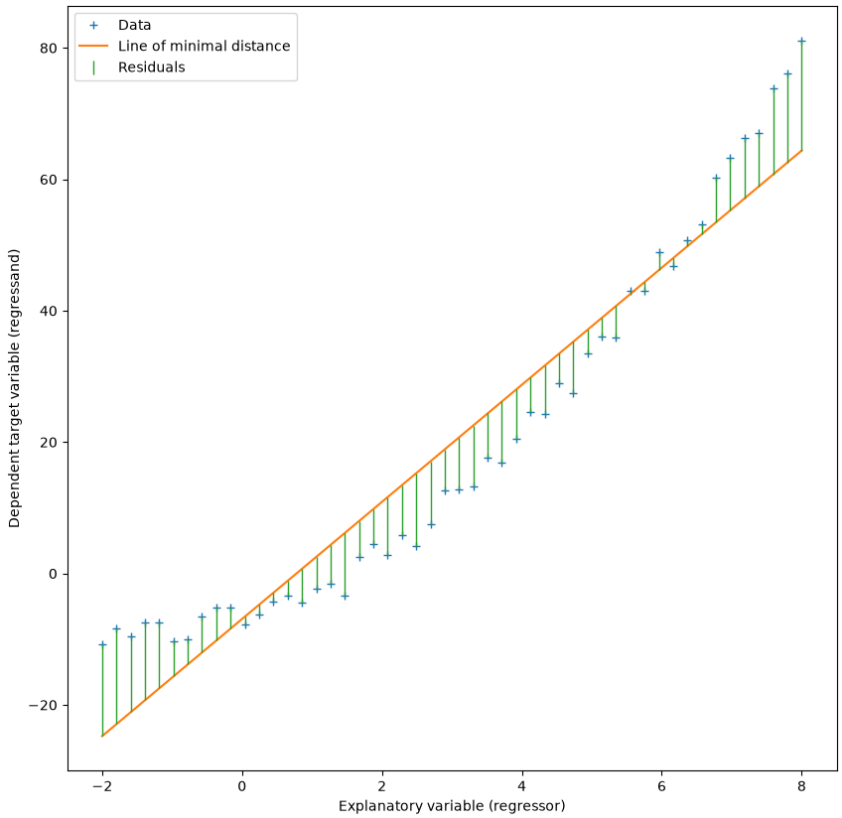
```
from scipy.optimize import curve_fit

# define function class for regression (linear):
def f(x, beta_0, beta_1):
    return beta_1*x + beta_0

# perform regression analysis:
parameters, covariance_matrix = curve_fit(f, x, y)
assert np.all(np.isclose((beta_0, beta_1), parameters))
```

From the Scipy documentation:
`scipy.optimize.curve_fit`

Use non-linear least squares to fit a function, f, to data.

Assumes ydata = f(xdata, *params) + eps.

```
fig, ax = plt.subplots(figsize=(10,10))
ax.plot(x, y, "+", label="Data")
ax.plot(x, f(x, beta_0, beta_1), label="Line of minimal distance")
yerr = np.stack((np.maximum(np.zeros_like(y), y - f(x, beta_0, beta_1)),
                 -np.minimum(np.zeros_like(y), y - f(x, beta_0, beta_1))))
ax.errorbar(x, y,
            yerr=yerr,
            linewidth=0, elinewidth=1,
            label="Residuals")
ax.set_xlabel("Explanatory variable (regressor)")
ax.set_ylabel("Dependent target variable (regressand)")
ax.legend()
plt.show()
```



Note that this is not the direction of maximal variance – that too minimises distances, but the distances of the data points (x, y) to the line. Linear regression minimises not the distances of the data points to the line but the residuals, which are the distances along the y-axis. The plot above shows this: the green bars run parallel to the y-axis, not perpendicular to the line (which is what the distance to the line would be).

This also makes clear that regression is an asymmetric method: regressor and regressand cannot be swapped without changing the model. When searching for the line along the direction of maximal variance, by contrast, the x- and y-coordinates are on an equal footing and the solution is invariant under rotations.

7.4.2 Linear regression

Definition
A regression of the form $Y_i = f(X_i, w) + \epsilon_i$ with $f(x, w) = w^T x$ is called a **linear regression**. The

entries of the vector w are called the **weights** of the model: w_j says how strongly the regressor X_j enters the prediction.

Definition
A regression of the form $Y_i = f(X_i, w, w_0) + \epsilon_i$ with

$$f(x, w, w_0) = w^T x + w_0 \tag{192}$$

is called a **linear regression with bias** w_0.

The second definition looks more general, but it reduces to the first. Append a constant one to the regressors and the bias to the weights:

$$X_+ := (1, X_1, \ldots, X_k), \qquad w_+ := (w_0, w_1, \ldots, w_k), \qquad (w_+)^T X_+ = w^T X + w_0. \tag{193}$$

This is the usual tacit convention: for $x = (a, b)$ we then have $f(x, w) = w^T x = (w_0, w_1, w_2) \cdot (1, a, b) = w_0 + w_1 a + w_2 b$. The w_0 term (denoted β_0 earlier) is also called the **bias term** (intercept). It has nothing to do with the bias of an estimator from the previous chapter; the word simply has two meanings in the machine learning literature.

Note
An example of such a model with a bias: the price Y of a house is to be predicted from the distance to the city centre X_1 and the age X_2,

$$Y = \beta_1 X_1 + \beta_2 X_2 + \beta_0 + \epsilon. \tag{194}$$

This is exactly the model we work through on the Melbourne dataset further down.

A word on terminology: in computer science and machine learning the function $f(x_1, x_2) = \beta_1 x_1 + \beta_2 x_2 + \beta_0$ is called "linear", although in the sense of linear algebra it is not – only $(x_1, x_2) \mapsto \beta_1 x_1 + \beta_2 x_2$ is linear, since a linear map sends zero to zero. In mathematics f is called an *affine linear* function. The X_+ trick above is exactly the translation between the two usages: the affine linear function on $\mathbb{R}^k$ is a linear function on $\mathbb{R}^{k+1}$.

A simple linear regression can be carried out with Scipy.stats:

```
from scipy.stats import linregress

linregress_result = linregress(x, y)
slope, intercept, r, p, se = linregress_result
print("Slope: %f" % slope)
assert np.isclose(slope, beta_1)
print("Intercept: %f" % intercept)
assert np.isclose(intercept, beta_0)
print("Correlation coefficient: %f" % r)
print("p -value: %.32f" % p)
print("Standard error of the slope: %f" % se)

Slope: 8.912806
Intercept: -6.929693
Correlation coefficient: 0.959514
p -value: 0.00000000000000000000000000046030
Standard error of the slope: 0.377630
```

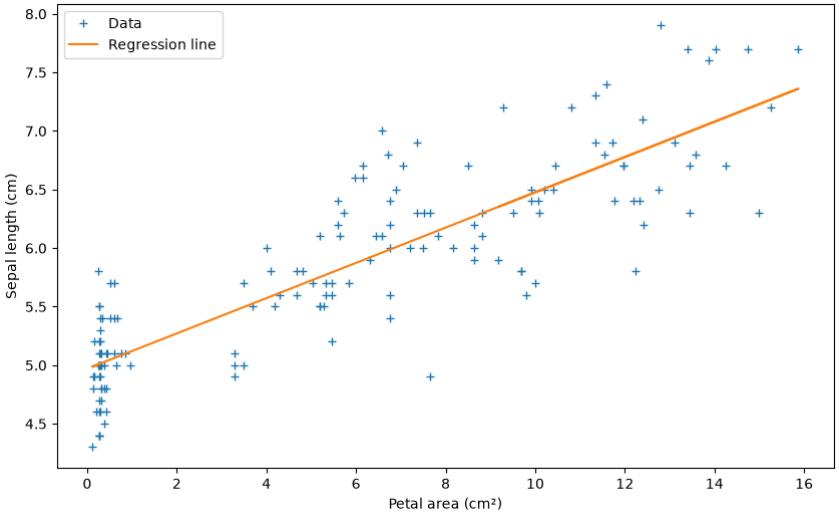
The five return values are worth a moment. `slope` is the slope β_1 and `intercept` is the intercept β_0, that is, the bias – those are the two parameters we were after. `r` is the correlation coefficient between x and y (see Covariance and dependence); its square r^2 is the proportion of the variance of y that the model explains, and it turns up again further down as the coefficient of determination R^2. `p` belongs to a test that Scipy ran along the way, unasked (more on that in a moment). And `se`, despite the name, is not a squared error but the **standard error of the slope estimate**: the estimated standard deviation

of `slope` if you were to repeat the whole experiment with a fresh sample. As a rule of thumb, the true slope lies within `slope` $\pm$ 2$\cdot$ `se` with a probability of about 95 %.

Scipy has also run a test for us straight away: a two-sided test with the null hypothesis that the slope `slope` equals 0. Since the p-value is considerably smaller than $\alpha = 0.05$, we reject the hypothesis and assume a slope $\neq 0$. (For now we simply take this as given, without going further into statistical tests – or we ignore this part of the Scipy.stats output.)

Before we write this down in general, here is an example in which the trick is immediately obvious. The Iris dataset gives us the petal length X_1 and the petal width X_2, and we would like to predict the sepal length Y. Now Y presumably depends less on length or width individually than on the *area* of the petal – and that, up to a shape factor, is the product of the two. So we build a new regressor $X_3 := X_1 X_2$ and use it to compute a perfectly ordinary linear regression $Y = w_1 X_3 + w_0 + \epsilon$.

```
from sklearn import datasets

iris = datasets.load_iris()
x_iris = iris.data
petal_area = x_iris[:, 2] * x_iris[:, 3]   # petal length times petal width
sepal_length = x_iris[:, 0]

result = linregress(petal_area, sepal_length)
print("Slope: %f" % result.slope)
print("Intercept: %f" % result.intercept)
print("Correlation coefficient: %f" % result.rvalue)
print("Standard error of the slope: %f" % result.stderr)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(petal_area, sepal_length, "+", label="Data")
ax.plot(petal_area, result.intercept + result.slope * petal_area, label="Regression line")
ax.set_xlabel("Petal area (cm²)")
ax.set_ylabel("Sepal length (cm)")
ax.legend()
plt.show()

Slope: 0.150646
Intercept: 4.970482
Correlation coefficient: 0.857300
Standard error of the slope: 0.007436
```



All we have done here is apply the function $\phi(X_1, X_2) := X_1 X_2$ to the regressors before the regression. The model stays linear – in the weights w, and that is what counts.

For comparison: the petal length alone gives $r = 0.872$, a little more than the area does. A basis function expansion is not automatically the better regressor just because it is more complicated – it is a hypothesis about the data, and whether it helps is an empirical question. Such preprocessing steps have a name of their own.

Definition
If a regressor variable has the form $\phi(X_i)$, where ϕ is an arbitrary function with a suitable domain, then this is still called a **linear regression with regressor X_i and basis function expansion**.

So in a linear regression only one thing is linear: the function f, in the variables w. This also lets us read the earlier tacit replacement of x by $(1, x)$ as a basis function expansion, with $\phi(x) = (1, x)$.

Since a linear regression analysis is very easy to carry out, the central question in linear regression becomes how to set up the model – and in particular the basis function – from the data.

The case $w = (w_0, w_1)$ is called a **simple linear regression**, the more general case a **multiple linear regression**, because there the regressor is a multivariate independent variable X. Scipy.stats's linear regression can only do the simple kind!

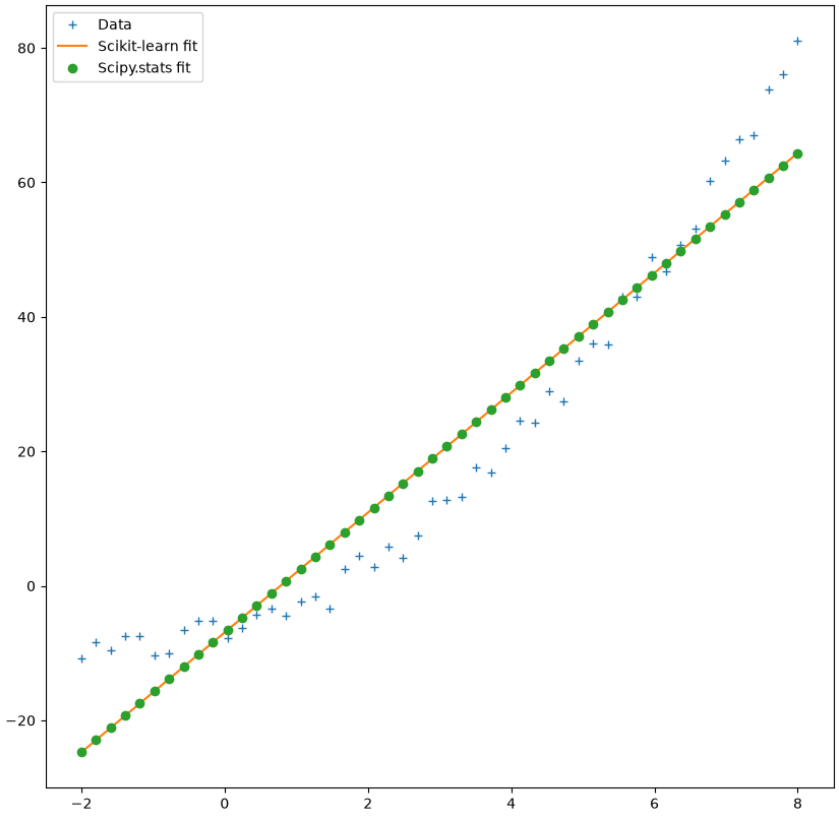
So we now turn to multiple linear regression with scikit-learn:

Caution: these terms elsewhere
Outside this chapter, its words are often used more narrowly than we have defined them here. Three narrowings you will meet regularly:

1. **Regression vs. classification.** For us, classification is a special case of regression (the target variable is then simply categorical). The widespread usage, by contrast, is "regression = continuous target variable, classification = discrete target variable", and libraries are mostly organised along those lines – in scikit-learn, `LinearRegression` and `LogisticRegression` sit in different corners of the documentation, although both are regressions in the sense used here.
2. **Regression = linear regression.** For us, a regression is any model with a function f. Someone who says "regression" without a qualifier, though, often means the linear one.
3. **Regression = regression line.** Regression works in arbitrarily many dimensions; what is meant, though, is often the one-dimensional case, whose result can be drawn as a straight line.

None of these narrowings is wrong; you only have to know which meaning the word is carrying at the moment. When in doubt, ask: what is the target variable, and how many regressors are there?

```
from sklearn.linear_model import LinearRegression

# first look at linear, to compare with the above:
linear_regressor = LinearRegression()
linear_regressor.fit(x.reshape( -1, 1), y)
Y_pred = linear_regressor.predict(x.reshape( -1, 1))

fig, ax = plt.subplots(figsize=(10,10))
ax.plot(x, y, "+", label="Data")
ax.plot(x, Y_pred, label="Scikit -learn fit")
ax.plot(x, f(x, beta_0, beta_1), "o", label="Scipy.stats fit")
ax.legend()
plt.show()
```



From the scikit-learn documentation:
`sklearn.linear_model.LinearRegression`

> From the implementation point of view, this is just plain Ordinary Least Squares (scipy.linalg.lstsq) or Non Negative Least Squares (scipy.optimize.nnls) wrapped as a predictor object.

Seen this way, scikit-learn is not 'better' than Scipy, just a layer on top of it.

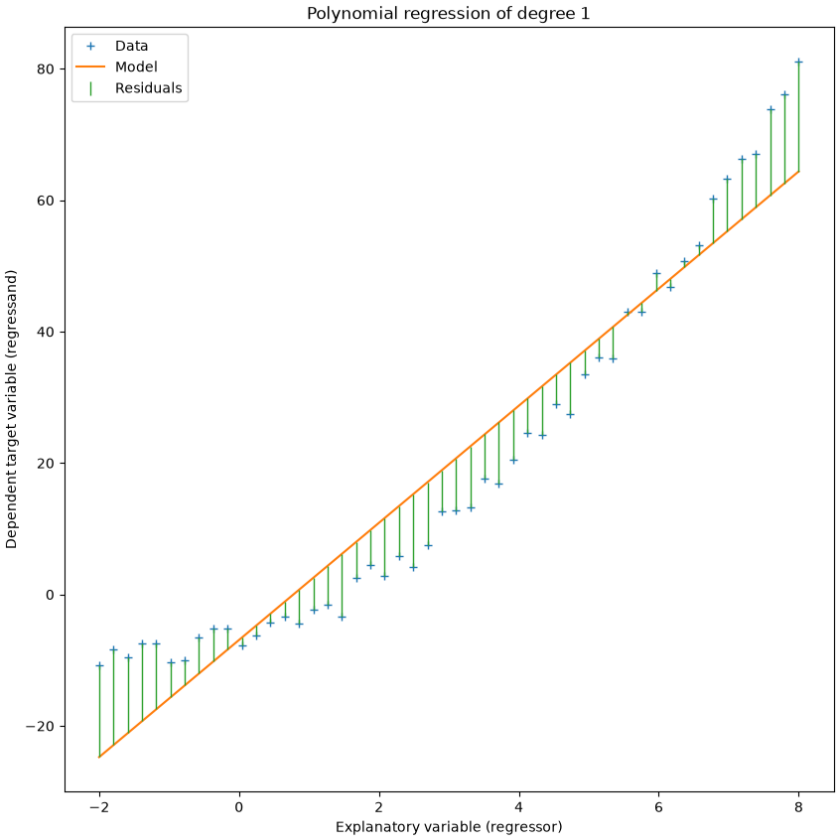
Example
Using the basis function $\phi(x) = (1, x, x^2, x^3, \ldots, x^d)$ with $d > 0$ is called **polynomial regression**. Concretely, for $d = 2$ this means $f(x, w) = w_0 + w_1 x + w_2 x^2$. Once again it is plain to see: f is linear in w, not in x.

One problem in choosing a basis function for a regression model is overfitting. With too many parameters available, the data can certainly be fitted very well, but the model does not generalise well. In practice this is checked by fitting on a training dataset and then evaluating on a test dataset.

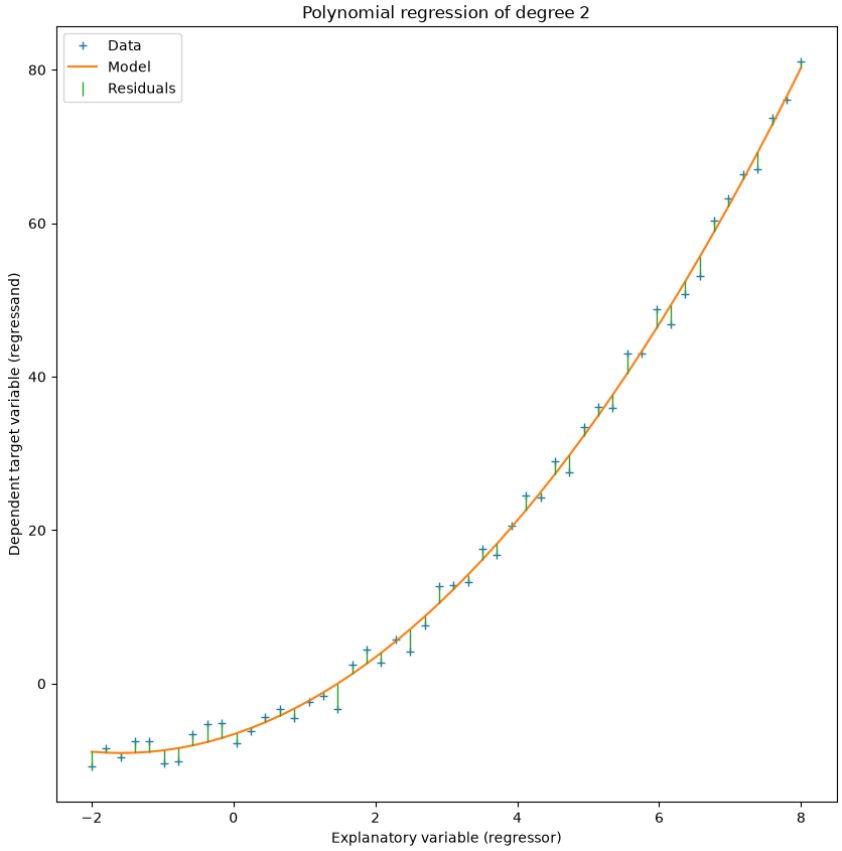
Example
We look at overfitting as a function of the parameter d of the polynomial regression, on the same dataset as in the earlier example with a regression line (the case $d = 1$).

```
# define basis function class
def phi(x, d):
    return np.dstack([np.power(x, i) for i in range(d+1)]).squeeze()

def plot_regression(x, y, Y_pred, ax, x_test=None, y_test=None):
    ax.plot(x, y, "+", label="Data")
    if(x_test is None):
        x_test = x
    ax.plot(x_test, Y_pred, label="Model")
    if(y_test is None):
```

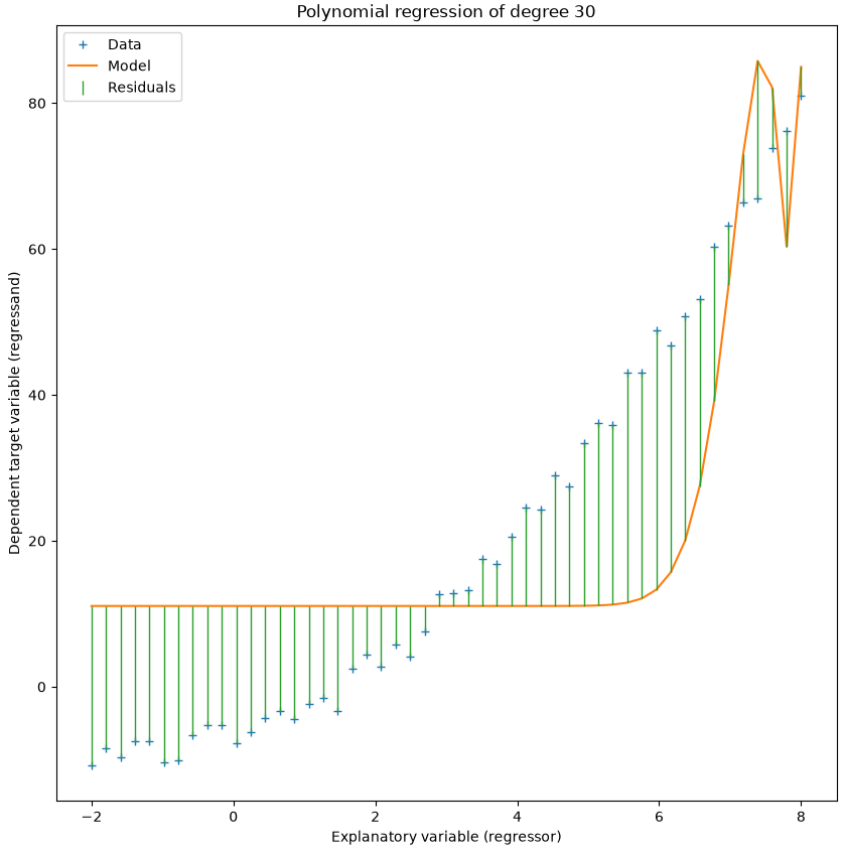
```
        y_test = y
    yerr = np.stack(( np.maximum(np.zeros_like(y_test), y_test - Y_pred),
                     -np.minimum(np.zeros_like(y_test), y_test - Y_pred)))
    ax.errorbar(x_test, y_test,
                yerr=yerr,
                linewidth=0, elinewidth=1,
                label="Residuals")
    ax.set_xlabel("Explanatory variable (regressor)")
    ax.set_ylabel("Dependent target variable (regressand)")
    ax.legend()
    return ax

def poly_regression_plot(x, y, d):
    # perform regression analysis:
    transformed_x = phi(x, d)

    linear_regressor = LinearRegression()
    linear_regressor.fit(transformed_x, y)
    Y_pred = linear_regressor.predict(transformed_x)

    fig, ax = plt.subplots(figsize=(10,10))
    plot_regression(x, y, Y_pred, ax).\
      set_title("Polynomial regression of degree %d" % d)
    plt.show()

poly_regression_plot(x, y, 1)
```




```
poly_regression_plot(x, y, 2)
```




```
poly_regression_plot(x, y, 30)
```



A linear regression model can also be formulated as

$$P(y|X,\theta) = \mathcal{N}(w^T X, \sigma^2) \tag{195}$$

or, with a basis function expansion,

$$P(y|X,\theta) = \mathcal{N}(w^T \phi(X), \sigma^2) \tag{196}$$

Here the θ_i are the parameters of the normally distributed errors ϵ_i.

7.4.3 An example application

We look at the Melbourne Housing Market dataset by Anthony Pino. It is licensed CC BY-NC-SA 4.0 and is therefore exempt from the licence of these notes: re-use only non-commercially and under the same terms.

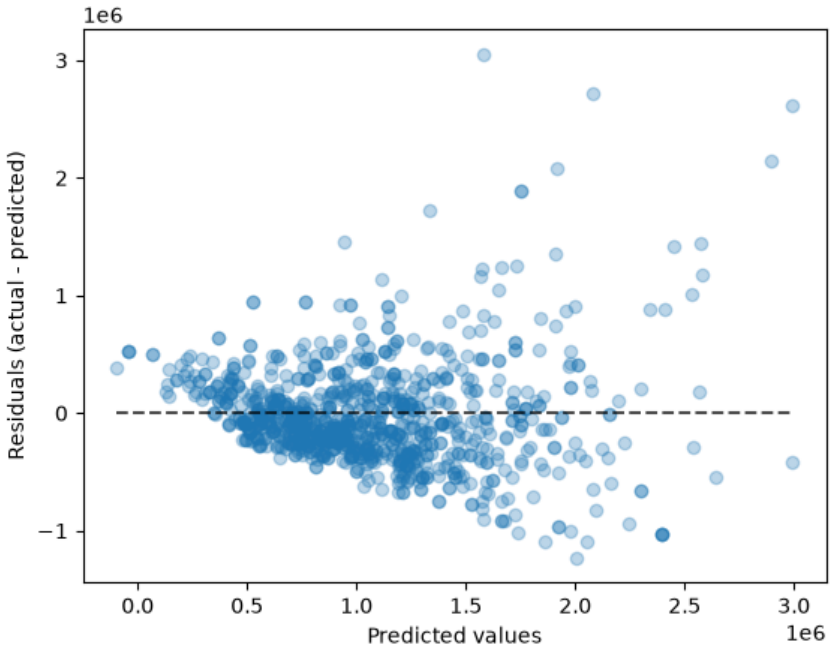
To get started, we clean the data a little.

```
import pandas as pd

df = pd.read_csv("Melbourne_housing_FULL.csv")
print(df.shape)
print(df.columns)

(34857, 21)
Index(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG',
       'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car',
       'Landsize', 'BuildingArea', 'YearBuilt', 'CouncilArea', 'Lattitude',
       'Longtitude', 'Regionname', 'Propertycount'],
      dtype='str')

# don't care about lat, lon here:
df = df.drop(columns=['Lattitude','Longtitude'])

# make Date a date:
df['Date'] = pd.to_datetime(df['Date'], format="%d/%m/%Y")

# Postcode is not really numeric:
df['Postcode'] = df['Postcode'].astype('category')

# make string variables categorical:
string_columns = df.select_dtypes(include=['object', 'string']).columns  # 'string' for pandas
    >= 3
df[string_columns] = df[string_columns].astype('category')

# Transform year built to age:
df['Age'] = 2017 - df['YearBuilt']
df = df.drop(columns='YearBuilt')

# kill faulty data:
df = df[df['BuildingArea'] != 0]
df = df[df['Age'] < 500]

# remove all suspects with missing data
df = df.dropna()

# remaining numerical variables:
numerical_columns = df.select_dtypes(['float']).columns
print(numerical_columns)

Index(['Price', 'Distance', 'Bedroom2', 'Bathroom', 'Car', 'Landsize',
       'BuildingArea', 'Propertycount', 'Age'],
      dtype='str')
```

```
regressor_columns = numerical_columns.drop('Price')
x = df[regressor_columns]
y = df['Price']

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test \
 = train_test_split(x, y, test_size = .20, random_state=0)

regressor = LinearRegression()
regressor.fit(x_train, y_train)
y_pred = regressor.predict(x_test)

from sklearn.metrics import r2_score
print("R^2 = %f%% of the variance is explained by the linear model"
      % (100*r2_score(y_test, y_pred)))

R^2 = 50.355458% of the variance is explained by the linear model
```

Before we interpret the coefficients, we check the model assumption that the error terms ϵ_i scatter around 0 independently of x: the residuals $y_i - f(x_i, \beta)$, plotted against the prediction, ought to lie around 0 with no discernible pattern. A funnel means that the spread depends on the price; curvature, that a linear relationship is not enough.

```
from sklearn.metrics import PredictionErrorDisplay

PredictionErrorDisplay.from_predictions(y_test, y_pred, kind="residual_vs_predicted",
                                        scatter_kwargs={"alpha": 0.3})
plt.show()
```



Definition

$$R^2 = 1 - \frac{\sum_{i=1}^{N} (y_i - f(x_i, \beta))^2}{\sum_{i=1}^{N} (y_i - \texttt{mean}(y))^2} \tag{197}$$

The quantity R^2 is also called the **coefficient of determination**. The numerator is the residual sum of squares of the model, the denominator that of the best constant model $\texttt{mean}(y)$; so R^2 measures what proportion of the spread of y the model explains away, compared with the constant. For least squares with an intercept this equals, on the training data, $\sum(f(x_i, \beta) - \texttt{mean}(y))^2 / \sum(y_i - \texttt{mean}(y))^2$,

the proportion of "explained" variance; on test data, as above, only the first form holds, and there R^2 can even come out negative (chapter Model selection).

```
cdf = pd.DataFrame(data = regressor.coef_,
                   index = regressor_columns,
                   columns = ['Coefficients'])
cdf
```

	Coefficients
Distance	-27377.550245
Bedroom2	109775.843441
Bathroom	264198.517970
Car	55837.834214
Landsize	23.997980
BuildingArea	2238.496526
Propertycount	-0.688488
Age	6081.670366

Interpretation:

If one house is a year older than another, the associated price difference is about 6081 (Australian) dollars. If one house is a kilometre closer to the centre (which Melbourne calls the *CBD*) – that is, its Distance variable is smaller by 1 – the associated price difference is about 27377 dollars.

Categorical regressors: one-hot encoding While cleaning the data we turned every text column into a `category` – and then quietly left them out of the model, since `numerical_columns` holds only the `float` columns. Yet `Type` (h = house, u = unit, that is a flat, t = townhouse) ought to have something to say about the price.

Simply setting $h = 1$, $u = 2$, $t = 3$ would be wrong: `Type` is on a nominal scale, with no order and no equal spacings, and the regression would compute with both (see Descriptive statistics, section Level of measurement). Instead, every level gets a 0/1 column of its own, the **one-hot encoding**; pandas calls the columns *dummies*. One of the columns is left out (`drop_first=True`): whatever is neither `u` nor `t` is `h`, and the omitted level becomes the *reference category* against which the other coefficients are measured. If all three columns were kept, their sum would always be 1 – the same information as the intercept β_0 – and the parameters would no longer be uniquely determined (the *dummy variable trap*).

```
dummies = pd.get_dummies(df['Type'], prefix='Type', drop_first=True, dtype=float)
print(dummies.head())

x_onehot = pd.concat([df[regressor_columns], dummies], axis=1)
x_train_oh, x_test_oh, y_train_oh, y_test_oh \
 = train_test_split(x_onehot, y, test_size=.20, random_state=0)

regressor_onehot = LinearRegression().fit(x_train_oh, y_train_oh)
print("R^2 without Type: %.1f%%" % (100 * r2_score(y_test, y_pred)))
print("R^2 with Type:    %.1f%%" % (100 * r2_score(y_test_oh, regressor_onehot.predict(x_test_oh
    ))))

pd.DataFrame(data=regressor_onehot.coef_, index=x_onehot.columns, columns=['Coefficients']).tail
    (2)
```

```
Type_t  Type_u
2      0.0     0.0
4      0.0     0.0
6      0.0     0.0
11     0.0     0.0
14     0.0     0.0
R^2 without Type: 50.4%
R^2 with Type:    50.7%
```

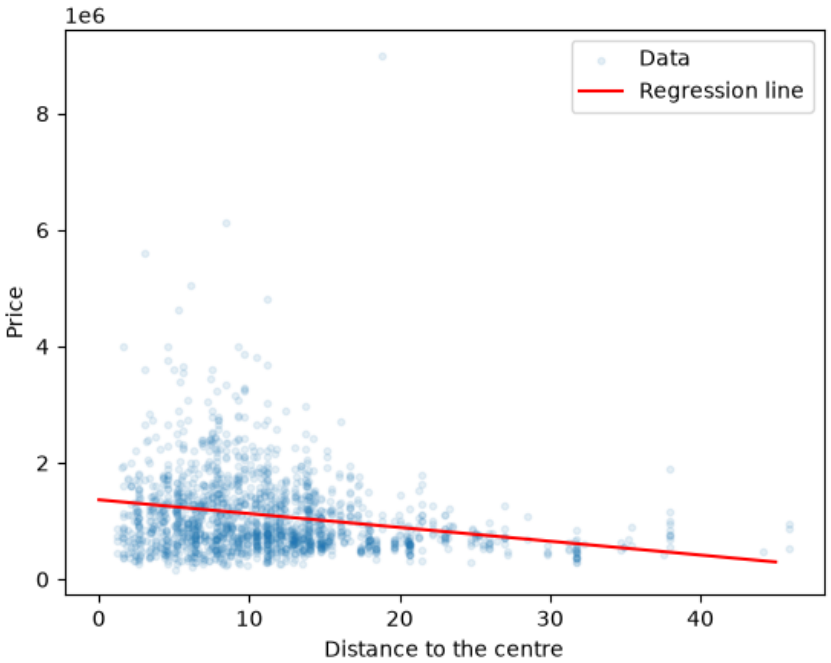
	Coefficients
Type_t	-66617.632753
Type_u	-178403.074899

The two new coefficients read as a price difference *relative to a house* with otherwise equal features: in the model, a flat with the same area, the same age and the same distance to the centre is around 178 000 dollars cheaper, a townhouse around 67 000 dollars. That R^2 barely rises is, on reflection, no surprise: most of what `Type` reveals about the price already sits in `BuildingArea`, `Landsize` and `Bedroom2` – flats are, above all, smaller. So the encoding did not invent the feature's value; it only made it readable.

An ordinal feature (a condition "poor / medium / good", say) could, by contrast, fairly have been encoded as $1, 2, 3$ – there the order is real, and only the equal spacings are an assumption. For nominal features with very many levels (`Suburb`, say, with several hundred suburbs) one-hot encoding becomes unwieldily wide; then other ideas are needed, which go beyond this course.

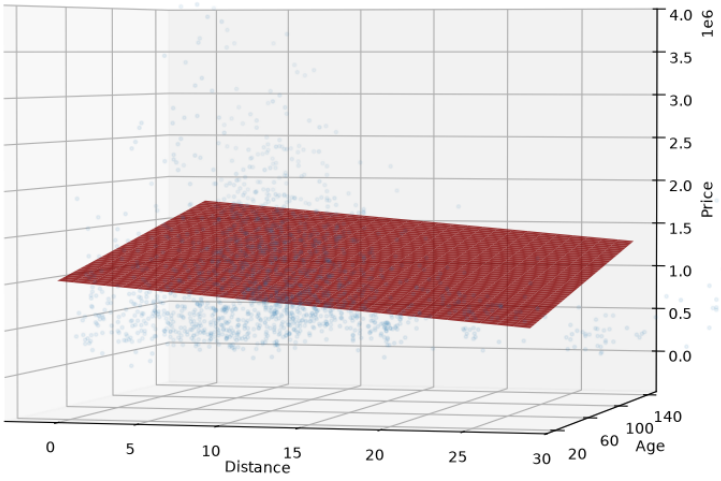
To close this example application, two more pictures of linear regression in the variables distance and age – first one of them, then both:

```
regressor_variable = x_train[["Distance"]]
regressor.fit(regressor_variable, y_train)
y_pred = regressor.predict(regressor_variable)

fig, ax = plt.subplots()
ax.scatter(x_test["Distance"], y_test,
           marker='.', alpha=0.1,
           label="Data")
xs = np.arange(np.max(x_test["Distance"]))
ys = regressor.coef_[0]*xs + regressor.intercept_
ax.plot(xs, ys, color='red', label="Regression line")
ax.set_xlabel("Distance to the centre")
ax.set_ylabel("Price")
ax.legend()
plt.show()
```




```
regressor_variable = x_train[["Distance", "Age"]]
regressor.fit(regressor_variable, y_train)
y_pred = regressor.predict(regressor_variable)

from mpl_toolkits.mplot3d import Axes3D

xs = np.arange(30)
```

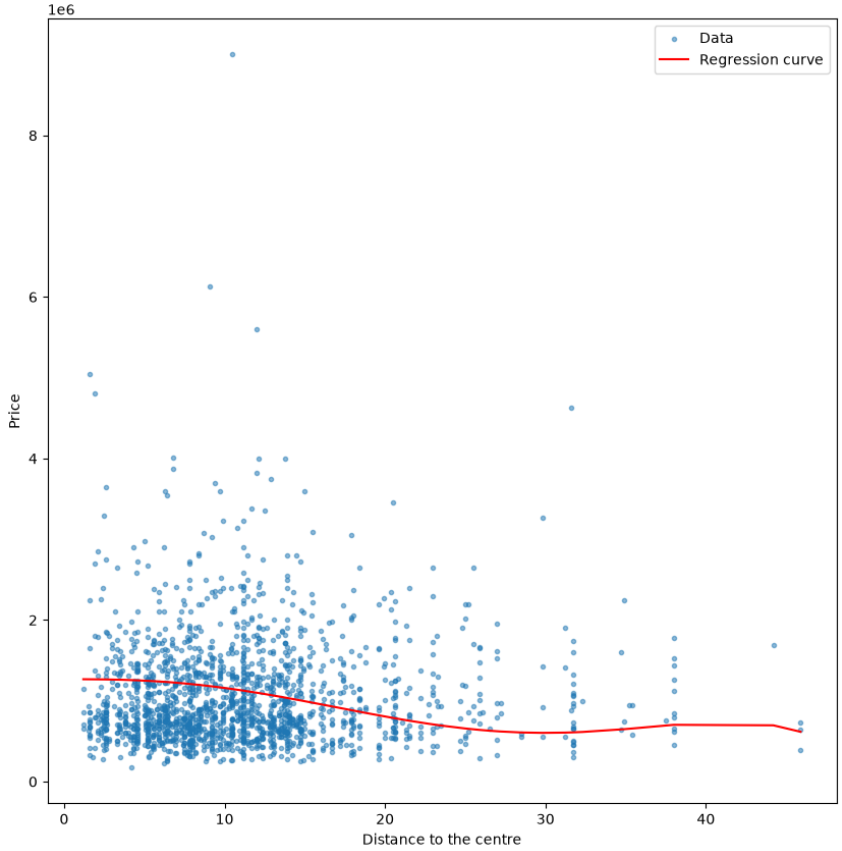
```
ys = np.arange(10, np.max(x_test["Age"]))
xs, ys = np.meshgrid(xs, ys)
zs =   xs*regressor.coef_[0] \
     + ys*regressor.coef_[1] \
     + regressor.intercept_

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111,projection='3d')
ax.plot_surface(xs, ys, zs,
                alpha=0.8, color='red')
ax.scatter(x_test["Distance"], x_test["Age"], y_test,
           marker='.', alpha=0.3)
ax.set_zlim3d(zmax=4000000)
ax.set_xlim3d(xmax=30)
ax.set_ylim3d(ymin=10)
ax.set_xlabel("Distance")
ax.set_ylabel("Age")
ax.set_zlabel("Price")
ax.set_yticks([20,60,100,140])
ax.view_init(elev=2., azim= -75)
plt.show()
```



Given how these data look, a polynomial regression would help here as well. We put together a pipeline to compute it.

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

degree = 5

polyreg = make_pipeline(PolynomialFeatures(degree), LinearRegression())
polyreg.fit(x_train[["Distance"]], y_train)
# for plotting purposes, sort:
x_test = x_test.sort_values("Distance")
y_pred = polyreg.predict(x_test[["Distance"]])
```

```
fig, ax = plt.subplots(figsize=(10, 10))
ax.scatter(x_test["Distance"], y_test,
           marker='.', alpha=0.5,
           label="Data")
ax.plot(x_test["Distance"], y_pred,
        color='red', label="Regression curve")
ax.set_xlabel("Distance to the centre")
ax.set_ylabel("Price")
ax.legend()
plt.show()
```

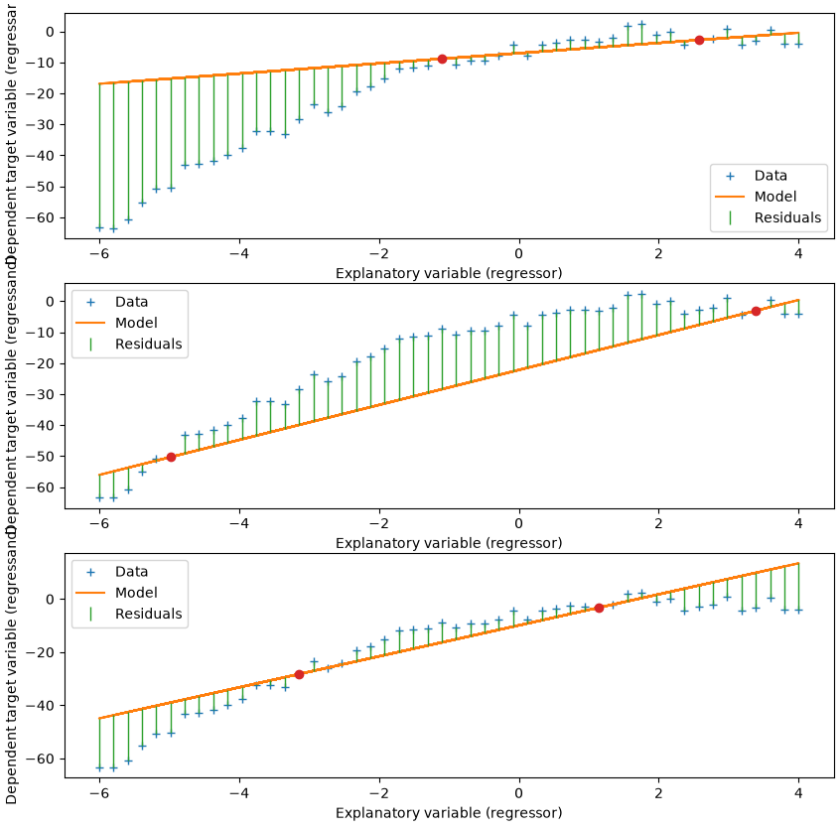


7.4.4 Regularisation

With only a few support points in the training data, overfitting becomes all the more likely. We look at this in a few examples:

```
# generate new synthetic data:
np.random.seed(123)
x = np.linspace( -6, 4)
y = -x**2 + 4*x - 5 + np.random.uniform( -4, 4, size=x.shape)

def plot_from_small_training(ax, random_state, model=LinearRegression()):
    # simulate having very few training data:
    x_train, x_test, y_train, y_test \
     = train_test_split(x, y, test_size = .95,
                        random_state=random_state)

    regressor = model
    regressor.fit(x_train.reshape( -1, 1), y_train)
    y_pred = regressor.predict(x_test.reshape( -1, 1))

    plot_regression(x, y, y_pred, ax, x_test, y_test)
    ax.plot(x_train, y_train, "o")
```

```
fig, axs = plt.subplots(3, 1, figsize=(10,10))
plot_from_small_training(axs[0], 3)
plot_from_small_training(axs[1], 4)
plot_from_small_training(axs[2], 5)
plt.show()
```



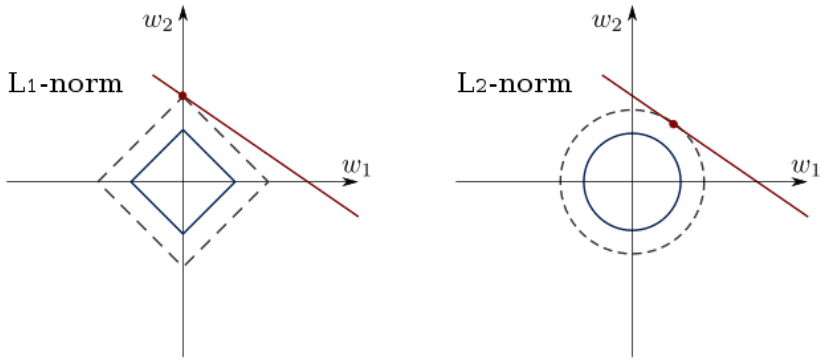
The trick of regularisation is not to fit the training data *too* well.

L2 regularisation, also known as **ridge regression**, minimises a penalty term $P = \alpha \sum_{i=1}^{n} \beta_i^2$ as an additional optimisation objective while searching for the best fit to the regression problem. Here α is a free parameter that sets the strength of the penalty.

The name L2 comes from P being the square of the L2 norm.

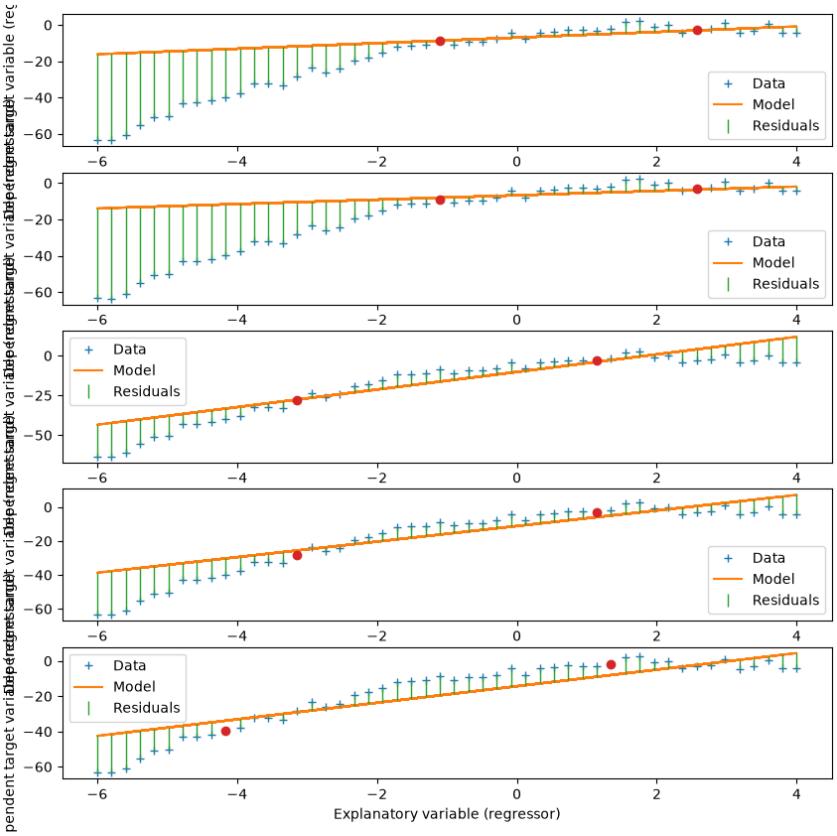
The same idea of penalising large model weights goes by the name *weight decay* in deep learning (the course *Introduction to Deep Learning*).

There is also **L1 regularisation**, which uses the summand $|\beta_i|$ instead of β_i^2 and so corresponds to the L1 norm; it is also known as **lasso regularisation**.




```
from sklearn.linear_model import Ridge

fig, axs = plt.subplots(5, 1, figsize=(10,10))
plot_from_small_training(axs[0], 3, model=Ridge(alpha=0.5))
plot_from_small_training(axs[1], 3, model=Ridge(alpha=2.5))
```

```
plot_from_small_training(axs[2], 5, model=Ridge(alpha=0.5))
plot_from_small_training(axs[3], 5, model=Ridge(alpha=2.5))
plot_from_small_training(axs[4], 10, model=Ridge(alpha=7))
plt.show()
```



The practical difference between L1 and L2: the L1 penalty sets coefficients to *exactly* 0 (those are the vertices of the L1 ball in the figure), so it selects regressors. With the polynomial terms $x, x^2, \ldots, x^8$ as regressors for the synthetic data above, you can see this directly:

```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import make_pipeline

def poly_fit(model):
    pipe = make_pipeline(PolynomialFeatures(degree=8, include_bias=False),
                         StandardScaler(), model)
    return pipe.fit(x.reshape( -1, 1), y)[ -1]   # the fitted model at the end of the pipeline

np.set_printoptions(precision=2, suppress=True)
ridge = poly_fit(Ridge(alpha=1.0))
lasso = poly_fit(Lasso(alpha=0.5))
print("Ridge coefficients:", ridge.coef_)
print("Lasso coefficients:", lasso.coef_)
print("Exactly 0 under Lasso:", np.sum(lasso.coef_ == 0), "of", len(lasso.coef_), "coefficients"
    )

Ridge coefficients: [10.6  -7.56  2.39 -1.92 -0.35  0.04 -0.72  0.36]
Lasso coefficients: [11.34 -8.76  0.71 -0.    0.   -0.    0.   -0.  ]
Exactly 0 under Lasso: 5 of 8 coefficients
```

7.4.5 Summary

A regression is a model $Y = f(X, \beta) + \epsilon$; *linear* means linear in the parameters, which is why basis functions and one-hot columns do not take us out of the model. The method of least squares yields $\beta_1 = \mathrm{Cov}(x, y)/\sigma^2(x)$, and it is maximum likelihood when the error terms are normally distributed;

residual plots check that assumption. Regularisation (ridge, lasso) penalises large coefficients and is the remedy for overfitting when there are too many basis functions.

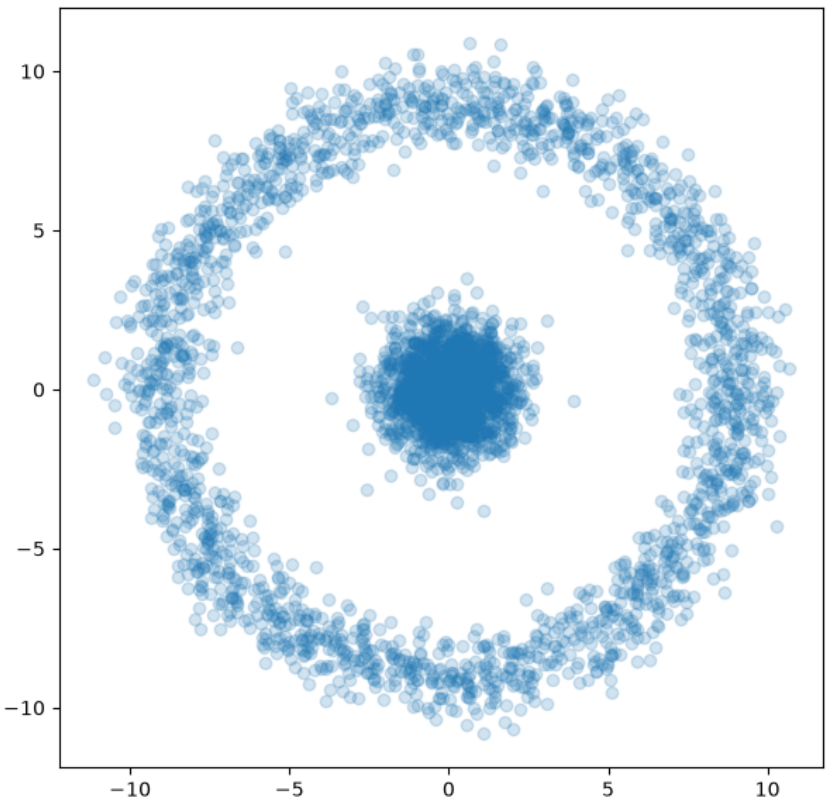
Video: Crash Course Statistics Regression

7.5 Kernel methods and the kernel trick

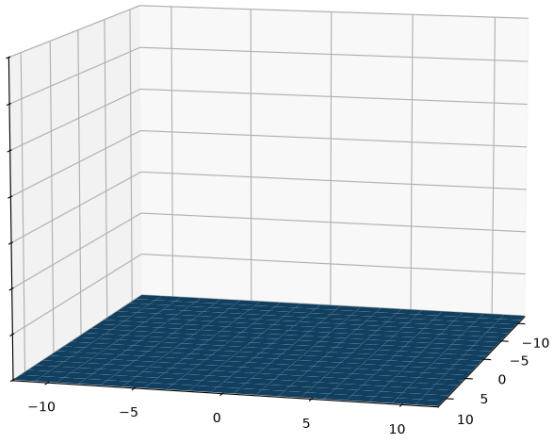
7.5.1 Motivation: the radial basis function

Example
We look at a synthetic dataset that is badly suited to k-means, and 'make' it suitable with a trick.

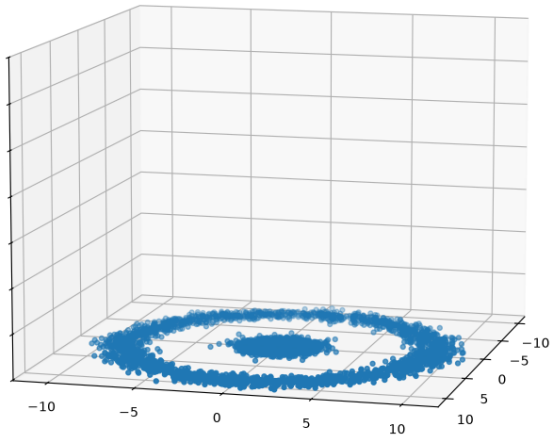
```
import numpy as np
import scipy.stats as st

np.random.seed(1234)  # scipy's .rvs draws from the global generator

# craft synthetic data:
def rings(n, radius=(8, 9)):
    rs = st.multivariate_normal(mean=radius[0]+np.ptp(radius),
                                cov=np.ptp(radius)/2
                               ).rvs(size=n)
    phis = st.uniform.rvs(size=n) * 2 * np.pi
    x, y = np.cos(phis), np.sin(phis)
    return (rs * np.stack([x, y])).T

n = 2000
sample = rings(n)
secondsample = st.multivariate_normal.rvs(mean=[0,0], cov=np.eye(2), size=n)
sample = np.concatenate([sample, secondsample])

import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(7,7))
ax.scatter(sample[:,0], sample[:,1], alpha=0.2)
plt.show()
```



```
# this produces the same as meshgrid, but is more explicit:
ticks = 20
frames = 70
x = np.outer(np.linspace( -12, 12, ticks), np.ones(ticks))
y = x.copy().T # transpose
rbf_surface_in_time =   np.linspace(0, 1, frames).reshape(1, 1, -1) \
                      * ((x**2 + y**2).reshape(ticks, ticks, 1))

import matplotlib.animation as animation
from IPython import display as ipydisplay
import io, base64

def init(ax):
    ax.clear()
    ax.set_xlim( -12, 12)
    ax.set_ylim( -12, 12)
    ax.set_zlim( -1, 350)
    ax.set_zticklabels([])

def update_surface(i, ax):
    init(ax)
    ax.view_init(10, 15+i)
    ax.plot_surface(x, y, rbf_surface_in_time[:,:,i])

def update_data(i, ax):
    init(ax)
    ax.view_init(10, 15+i)
    ax.scatter(sample[:,0], sample[:,1],
               2*i/frames * (sample[:,0]**2 + sample[:,1]**2),
               marker='.', s=50)

def plot_ani(func=update_surface):
    fig = plt.figure(figsize=(7, 7))
    fig.set_size_inches(9, 9)
    ax = plt.axes(projection='3d')
    ani = animation.FuncAnimation(fig, func, fargs=(ax,), interval=100,
                                  frames=np.hstack([np.arange(frames),np.arange(frames)[:: -1]]))
    video = ani.to_html5_video()
    # Also output the last frame as a PNG: the web export shows the video,
    # the PDF export (which cannot do HTML) uses the still image.
    png = io.BytesIO()
    fig.savefig(png, format='png')
    ipydisplay.display({'text/html': video,
                        'image/png': base64.b64encode(png.getvalue()).decode('ascii')},
                       raw=True)
    plt.close()

plot_ani(update_surface)
```

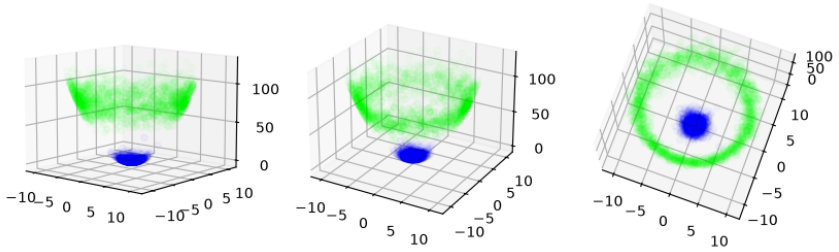



```
plot_ani(update_data)
```



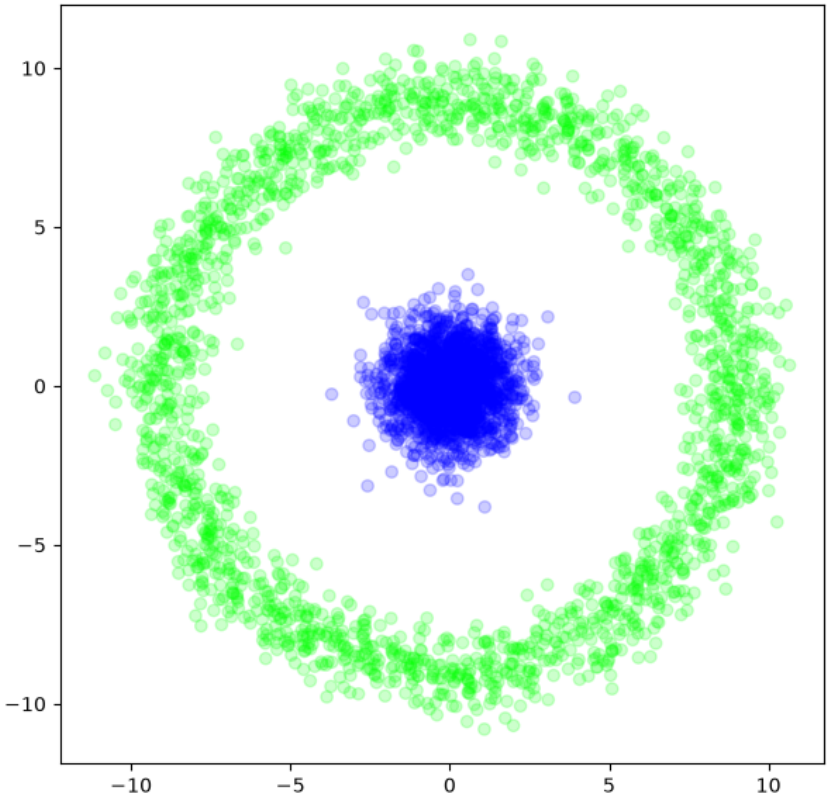
Data transformed in this way can now be clustered with 2-means!

```
def radial(x, y):
    """Squared distance from (x,y) to the origin: a single coordinate
    of the feature map \varphi, which we introduce in general further below."""
    return x**2 + y**2

xs, ys = sample[:,0], sample[:,1]
zs = radial(xs, ys)

from sklearn.cluster import KMeans
kmeans_fit = KMeans(n_clusters=2, n_init=1, random_state=0).fit(np.stack([xs,ys,zs]).T)

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(331, projection='3d')
ax2 = fig.add_subplot(332, projection='3d')
ax3 = fig.add_subplot(333, projection='3d')
for axis in (ax, ax2, ax3):
    axis.scatter(xs, ys, zs,
                 alpha=0.2, c=kmeans_fit.labels_,
                 cmap='brg')
ax.view_init(elev=10., azim= -50)
ax2.view_init(elev=25., azim= -60)
ax3.view_init(elev=75., azim= -70)
plt.show()
```



We can also run only the k-means computation in this larger space (with the radial feature $x^2 + y^2$), then carry the labels back and show them in our 'old' two-dimensional plot:

```
fig, ax = plt.subplots(figsize=(7,7))
ax.scatter(sample[:,0], sample[:,1], alpha=0.2,
           c=kmeans_fit.labels_,
           cmap='brg')
plt.show()
```



What we actually needed The k-means algorithm measures *distances* all the time. It is therefore invariant under translation and rotation. And if we do not transform our data at all, but merely measure distances with the notion of distance from the transformed space, that could, depending on the transformation, be less expensive to compute.

Other algorithms, too – k-nearest neighbours and support vector machines, for instance (neither of which we have discussed) – rest on nothing but measurements of distance.

7.5.2 Describing data by distances

Example
Two points A, B in the plane can be described by their coordinates. Alternatively one can remember a single datum, the distance $d(A, B)$, and from it recover the dataset (A, B) up to translation and rotation.

Example
Three points A, B, C in three-dimensional space can be described by their coordinates. Alternatively one can remember the datum of the distances $d(A, B)$, $d(B, C)$, $d(A, C)$ and from it recover the dataset (A, B, C) up to translation and rotation.

The kernel trick is now to describe the data x_i not by the symmetric matrix $d(x_i, x_j)$ but by some other function $\kappa(x_i, x_j)$ that plays the same role as a measurement of distance in a transformed coordinate system.

The Euclidean distance is

$$d(a, b) = \|b - a\| = \sqrt{\langle b - a, b - a\rangle} = \sqrt{(b - a)^T (b - a)} \tag{198}$$

If one applies a linear transformation M to the data, what one computes is

$$d(Ma, Mb)^2 = \|Mb - Ma\|^2 = \|M(b-a)\|^2 = \langle M(b-a), M(b-a)\rangle = (b-a)^T M^T M(b-a) \quad (199)$$

so that for a measurement of distance in the transformed space we do not actually have to compute Ma and Mb at all – 'only' apply $M^T M$, rather often. With luck, that is the easier job.

Often the inner product $\langle a, b\rangle$ itself is a good notion of distance too, or $\langle Ma, Mb\rangle = b^T M^T Ma$.

An aside, by way of explanation: we can also compute

$$\|b-a\|^2 = \langle b-a, b-a\rangle = \langle b, b\rangle - 2\langle a, b\rangle + \langle a, a\rangle = \|a\|^2 + \|b\|^2 - 2\langle a, b\rangle \quad (200)$$

which for normalised vectors a, b (that is, $\|a\| = 1$ and $\|b\| = 1$) shows a close relationship between $\langle a, b\rangle = \|a\|\|b\|\cos(\measuredangle(a, b)) = \cos(\measuredangle(a, b))$ and $d(a, b) = \sqrt{2}\sqrt{1 - \langle a, b\rangle}$.

One now abstracts away from the linear situation and allows an arbitrary map into a vector space with an inner product.

7.5.3 Kernels

Definition
A map $\kappa\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is called a **kernel** if there is a vector space F with an inner product $\langle , \rangle$ together with a map $\phi\colon \mathcal{X} \to F$ such that $\kappa(x, y) = \langle \phi(x),\ \phi(y)\rangle$ holds for all $x, y \in \mathcal{X}$. The vector space F is called the **feature space** and ϕ the feature map.

With a kernel one need not know the space F and the map ϕ at all, because for symmetric positive definite functions $\kappa\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ there is always a construction of a $\phi\colon \mathcal{X} \to F$ that represents κ as a kernel (*Mercer's theorem*).

One also speaks of a **Mercer** kernel; more generally, functions κ that are not necessarily positive definite or symmetric are also called **kernel functions**.

One speaks of the **kernel method** in general when an algorithm is reformulated so that it works only with the inner products of the feature vectors, and these are then replaced by a kernel function.

Definition
Kernels of the form

$$\kappa(x, y) = e^{\left(-\frac{\|x-y\|^2}{2\sigma^2}\right)} \quad (201)$$

are called **Gaussian kernels** with parameter $\gamma = \frac{1}{2\sigma^2}$. Alternatively one speaks of the **squared exponential kernel**, the **quadratic exponential kernel** or the **radial basis function kernel**.

Gaussian kernels Note that we need not compare $(x - \mu)$ with $(y - \mu)$ here, because the two have the same difference $x - y$. The usual normalisation drops out as well, because we are dealing not with probabilities but with a notion of distance.

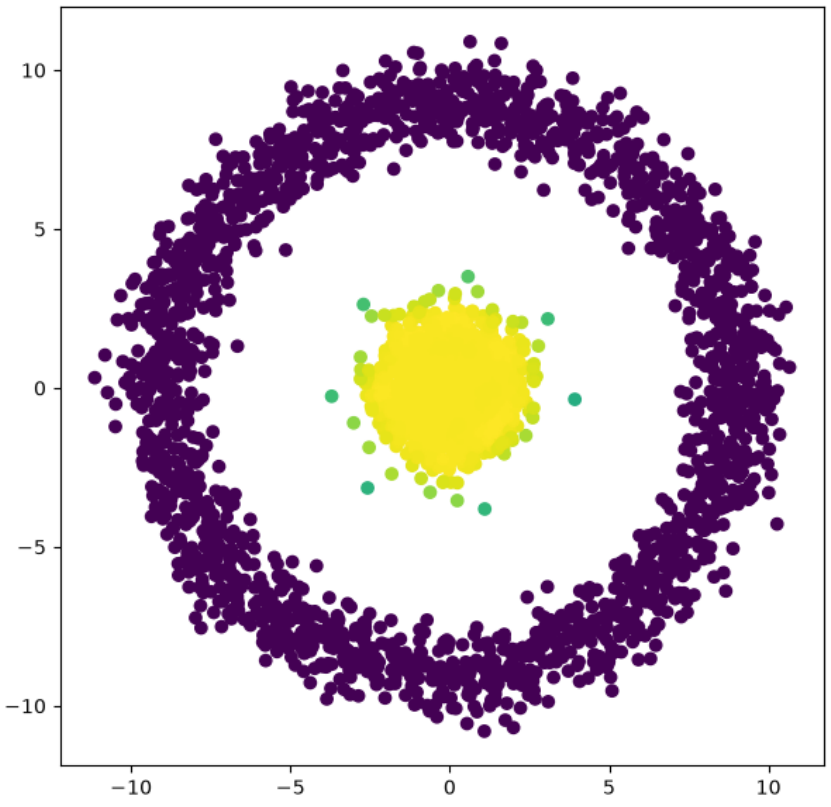
```
from sklearn.metrics.pairwise import rbf_kernel

k = rbf_kernel(sample, gamma=None)
# from the scikit -learn documentation:
# gamma: If None, defaults to 1.0 / n_features.

k

array([[1.00000000e+00, 4.13904520e -58, 2.16272228e -01, ...,
        1.22811834e -23, 2.13172511e -14, 5.18538063e -21],
       [4.13904520e -58, 1.00000000e+00, 1.21371818e -57, ...,
        4.35762550e -19, 2.06913343e -19, 2.23830019e -21],
       [2.16272228e -01, 1.21371818e -57, 1.00000000e+00, ...,
        1.19980400e -19, 2.05065431e -12, 4.27055216e -17],
```

```
       ...,
       [1.22811834e -23, 4.35762550e -19, 1.19980400e -19, ...,
        1.00000000e+00, 1.64084897e -02, 7.84350348e -01],
       [2.13172511e -14, 2.06913343e -19, 2.05065431e -12, ...,
        1.64084897e -02, 1.00000000e+00, 3.60038828e -02],
       [5.18538063e -21, 2.23830019e -21, 4.27055216e -17, ...,
        7.84350348e -01, 3.60038828e -02, 1.00000000e+00]],
      shape=(4000, 4000))
```

But what does one do with the kernel?

Linear regression with ridge regularisation, for instance:

```
from sklearn.kernel_ridge import KernelRidge

# target variable is #cluster,
# given how we created the sample, it simply is:
y = np.hstack([np.zeros(len(sample)//2),np.ones(len(sample)//2)])

clf = KernelRidge(kernel="precomputed")

clf.fit(k, y)
y_pred = clf.predict(k)

fig, ax = plt.subplots(figsize=(7,7))
ax.scatter(sample[:,0], sample[:,1], c=y_pred)
plt.show()
```

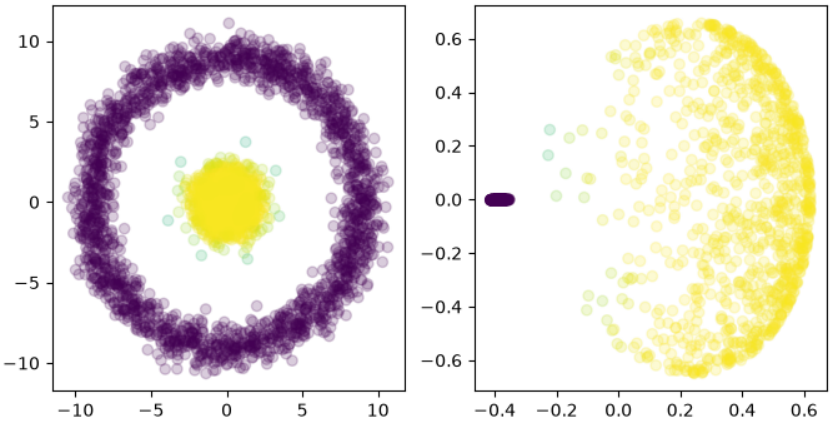


7.5.4 Kernel PCA

Principal component analysis computes the eigenvectors of the covariance matrix. One can also transform the data nonlinearly first – or else use the correspondingly transformed covariance matrix.

```
from sklearn.decomposition import PCA, KernelPCA

pca = PCA(n_components=2)
kernel_pca = KernelPCA(
```

```
    n_components=None, kernel="rbf", gamma=0.2, alpha=2.5
)
# these parameters gamma, alpha needed some manual tweaking!

sample_pca = pca.fit_transform(sample)

# The following may need more than 16 GB of RAM,
# if the sample is large, so we're not going there:
#sample_kernel_pca = kernel_pca.fit_transform(sample)

# Keep it calm, train_size=2000 suffices:
from sklearn.model_selection import train_test_split
smaller_sample, remainder_sample, smaller_y, remainder_y \
  = train_test_split(sample, y_pred, train_size=2000, random_state=0)

sample_kernel_pca = kernel_pca.fit_transform(smaller_sample)

fig, axs = plt.subplots(ncols = 2, figsize=(8,4))
axs[0].scatter(sample_pca[:,0], sample_pca[:,1],
               c=y_pred, alpha=0.2)
axs[1].scatter(sample_kernel_pca[:,0], sample_kernel_pca[:,1],
               c=smaller_y, alpha=0.2)
plt.show()
```



Strictly speaking, kernel PCA first increases the number of dimensions (with the nonlinear feature map), then transforms them linearly onto principal components, so that unimportant dimensions can be thrown away again. The first principal component in the feature space no longer corresponds to a straight line in the original observation space but to a nonlinear curve.

7.6 Statistical tests: the z-test, the t-test and the p-value

Exploratory data analysis leaves us with hypotheses about which relationships are hidden in the data. When we build systems with machine learning we have a great many decisions to make: whether we may treat a variable as normally distributed, say, or whether two groups or clusters have roughly the same distribution or very different ones.

Tests appear here in two quite different roles, and it helps to keep them apart. In **exploratory analysis** they are one tool among many and answer questions such as "Is this attribute normally distributed?", "Do samples A and B have the same distribution for this attribute? The same mean?" – the answer then decides the next step of the analysis, for instance whether a method that presupposes normality may be used at all. In **communicating results**, by contrast, they are the argument itself: "The test shows that the groups have different distributions, so we use a mixture model" (see Expectation Maximization). In the first role you may test generously, in the second you may not; the section Significance at the end of this chapter explains why.

Strictly speaking, statistical tests are not part of "machine learning", but in the running order of the lecture course this is where they fit naturally.

A test can be read as an estimator that estimates not a single value but a whole range of possible values, followed by a decision rule that interprets the estimate in the light of a hypothesis. We will treat the subject only superficially – just far enough to judge what a test can achieve and where the limits of statistical tests lie – but before that we look at what the difference between single values and ranges of values means for communicating scientific results, data science results included.

7.6.1 Prediction intervals

By way of motivation we look at the figure Figure 21 and then set it against Figure 22.

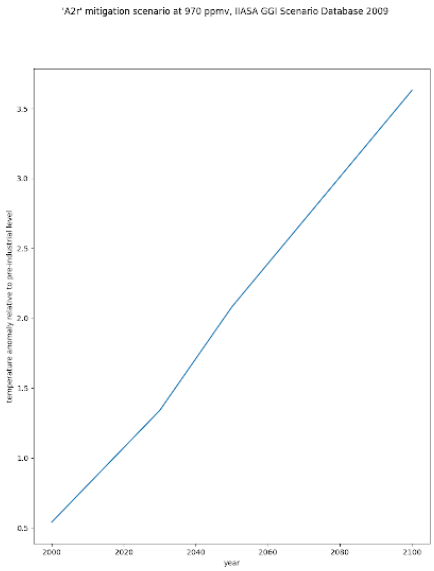


Figure 21: Global warming, projection (A2r scenario at 970 ppmv, IIASA 2009)

The figure shows the projection of a statistical model for global warming – given here as the temperature change relative to the pre-industrial level, defined as the mean temperature of the years 1850 to 1900 – on the assumption that the CO2 content of the atmosphere rises to 970 ppmv (parts per million by volume; for comparison, 30 May 2022: 421 ppmv, and 1900: 296 ppmv). For context: the Paris climate agreement sets a limit of 1.5°C or 2°C – not by 2040, but by 2100.

Let us look at this figure with a data scientist's eyes: what is there to criticise?

A common objection to models and figures of this kind is that things could turn out this way, but just as easily quite differently – these are only theoretical models, after all. The figure expresses a certainty about the course of the curve that cannot possibly exist, which makes the claim "this is how it will be" implausible from the outset. Granted, what it means to say is "this is how it could be", but on its own that is about as useful as "it might rain or it might stay dry".

To make decisions on the basis of a statistical prediction we also need to know how plausible it is. For comparison, look at Figure 22, a figure from the IPCC report SR15, Summary for Policymakers, Figure 1 a) Allen et al. [2018]:

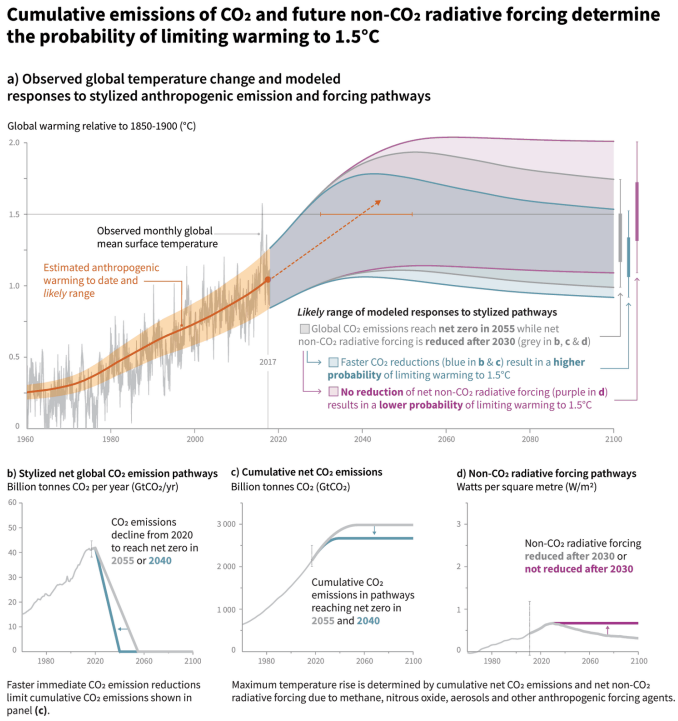


Figure 22: Global warming, projection (IPCC SR15, Fig. 1)

To read this figure properly we also need to know the *IPCC Calibrated Language*, or more precisely the *Likelihood scale* (AR4 and AR5 here refer to the 'assessment reports' that appeared before the special report on 1.5°C):

Table 1. Likelihood Scale

Term*	Likelihood of the Outcome
Virtually certain	99-100% probability
Very likely	90-100% probability
Likely	66-100% probability
About as likely as not	33 to 66% probability
Unlikely	0-33% probability
Very unlikely	0-10% probability
Exceptionally unlikely	0-1% probability

* Additional terms that were used in limited circumstances in the AR4 (*extremely likely* – 95-100% probability, *more likely than not* – >50-100% probability, and *extremely unlikely* – 0-5% probability) may also be used in the AR5 when appropriate.

Figure 23: The *Likelihood scale* of the IPCC Calibrated Language Allen et al. [2018].

Now we can see that the grey shaded area is where the actual measurements will lie with probability at least 2/3 (according to the model). With that information we can judge far better what warming to expect even if we 'get lucky', and how bad it could get if we 'are unlucky'. Strictly speaking, the grey shaded area is a **prediction interval**: the range in which the actual future value is presumed to lie with high probability. Closely related is the **confidence interval** for a parameter: an interval computed from the sample and constructed so that in (say) 95% of all samples it covers the true parameter.

For the mathematical theory of set estimators and confidence intervals we refer you to the statistics lectures. In what follows our main aim is to understand the notion of a test well enough to judge whether a claimed interpretation of a statistical test is serious. Once we can do that, we also want to recognise the situations in which a hypothesis test can deliver useful insight. Above all, that means getting clear about the notion of significance.

7.6.2 From the sample to the population

Before we define tests, let us be clear about what we actually want. Every summary measure from the exploratory analysis – mean, proportion, median, correlation coefficient – is computed on a **sample**. What we are interested in is the **population** the sample came from: all passengers, all customers, all the measurements that could have been made. A different sample would have yielded a different number. Inferring from the sample to the population therefore always takes three steps:

1. **Estimate:** compute the summary measure from the sample (the proportion, the mean, the difference of two means).
2. **State the uncertainty:** an interval in which the population's summary measure plausibly lies. A number without an interval is not an inference about the population but a description of the sample.
3. **Decide:** hold the question you had up against the interval. "Is the coin fair?" means: does $\frac{1}{2}$ lie in the interval? "Do women survive more often?" means: does 0 lie in the interval for the difference?

There are two routes to step 2.

The standard error. By the central limit theorem the mean of a sample of size n is approximately normally distributed with standard deviation $\sigma/\sqrt{n}$, the *standard error*. In 95 % of samples the population mean then lies in the interval

$$\texttt{mean}(x) \pm 1.96 \cdot \frac{\sigma}{\sqrt{n}}, \tag{202}$$

where σ is replaced by the sample standard deviation. For a proportion $\hat{p}$ we have $\sigma = \sqrt{\hat{p}(1-\hat{p})}$. Coin-toss example: with 45 successes in 100 tosses, $\hat{p} = 0.45$ and the standard error is $\sqrt{0.45 \cdot 0.55/100} \approx 0.05$, so the interval is roughly $[0.35,\ 0.55]$. The $\frac{1}{2}$ lies inside it, so the coin could be fair. To reject that, the interval would have to be narrower, and it shrinks only with $\sqrt{n}$: halving the width takes four times the sample. (With the cruder Chebyshev inequality in place of the normal approximation one gets an interval of ± 0.22 and a required sample of over 2000 tosses; that calculation is in the section *Looking ahead: confidence intervals* below.)

The bootstrap. For summary measures that have no formula, or whose formula you do not know, you pretend that the sample is the population: draw from it, *with replacement*, a new sample of the same size, compute the measure, and repeat many thousands of times. The spread of these bootstrap measures estimates the spread of the measure across real samples, and the 2.5 and 97.5 percentiles give a 95% interval. With the arrays `survived` and `sex` from the chapter Titanic EDA:

```
rng = np.random.default_rng(0)
f = survived[sex == 'female']
m = survived[sex == 'male']
difference = f.mean() - m.mean()
bootstrap = np.array([rng.choice(f, len(f)).mean() - rng.choice(m, len(m)).mean()
                      for _ in range(10000)])
print(difference, np.percentile(bootstrap, [2.5, 97.5]))
```

```
0.553  [0.495 0.609]
```

The women's survival rate is 55 percentage points above the men's, and even at the unfavourable end of the interval still 50. The 0 lies far outside: the difference is no accident of the sample. `scipy.stats.bootstrap` does the same thing in one line, with better-corrected percentiles.

What the recipe assumes. Both routes assume that the sample is a *random sample* from the population, that is, that every row of the population had the same chance of ending up in the sample. The Titanic list is a random sample of exactly one population: the passengers of that ship (and even then only the Kaggle extract). About other shipping disasters the interval says nothing, however narrow it is. Anyone who runs a survey on campus and wants to draw conclusions about all students has the same problem, and no sample size in the world cures it.

A statistical **test**, the subject of this chapter, is step 3 made formal: a rule, fixed in advance, that decides from the sample whether a hypothetical value is still plausible. The z-test and the t-test below are exactly the standard-error calculation above, read the other way round: instead of asking whether μ_0 lies in the interval around $\texttt{mean}(x)$, they ask how many standard errors $\texttt{mean}(x)$ is away from μ_0.

7.6.3 Looking ahead: confidence intervals

Note

Looking ahead (not part of the course proper): this section and the two sections *From set estimator to test* and *Best simple tests* are background that helps make sense of statistical tests. They explain where the tests come from, and taken on their own they are not part of the course proper. If you skip them, read straight on at the section *Tests* and then at *The z-test*.

Definition

Let $(\mathcal{X}, P_\theta : \theta \in \Theta)$ be a statistical model, Σ an arbitrary set and $\tau\colon \Theta \to \Sigma$ a quantity to be estimated that depends on the parameter θ. Let $0 < \alpha < 1$ be a real number. A map $C\colon \mathcal{X} \to \mathcal{P}(\Sigma)$ that assigns to every possible observation $x \in \mathcal{X}$ a set $C(x) \subseteq \Sigma$ is called a **confidence region for τ at error level α** if

$$\inf_{\theta\in\Theta} P_\theta\left(x \in \mathcal{X}\colon \tau(\theta) \in C(x)\right) \geq 1 - \alpha. \tag{203}$$

If in addition $\Sigma = \mathbb{R}$ and every $C(x)$ is an interval, we speak of a **confidence interval**.

Example

The trivial case: The constant confidence region $C(x) := \Sigma$ always satisfies $\tau(\theta) \in C(x)$ for every θ, so $P_\theta\left(x \in \mathcal{X} : \tau(\theta) \in C(x)\right) = 1 \geq 1-\alpha$ for every α. Read: we are certain that the 'right' estimate lies in the range of τ. That is not a particularly useful confidence region, of course, but it makes clear that the region is perfectly allowed to be 'too big'. It is in our interest to find a C whose $C(x)$ are as small as possible, because only then does the statement become interesting.

Example

Coin toss: n repeated tosses of an unfair coin with success probability p can be modelled with the **binomial model** (see The most important discrete distributions): $\mathcal{X} = \{0, \ldots, n\}$ is the number of successes, the unknown parameter is $\theta = p \in \Theta = [0, 1]$ and $P_\theta = \mathrm{Bin}(n, \theta)$ (and $\tau = id$, so we are estimating the parameter θ itself).

We have already computed the maximum likelihood estimator for θ there: $\hat{\theta} = \frac{x}{n}$, the proportion of successes. Concretely, if we toss the coin 100 times and get tails 45 times, we estimate θ as $\frac{45}{100}$ (and in particular that the coin really is unfair).

But what is our confidence – our trust – in this? Do we still regard values far from $\frac{x}{n}$ as halfway plausible, or can we pin down a very narrow range around $\frac{x}{n}$ that we hold to be plausible? In the concrete example with the estimate $\theta = \frac{45}{100}$, would $\theta = \frac{1}{2}$ still be halfway plausible?

We now want to construct intervals $C(x) = (\frac{x}{n} - \epsilon, \frac{x}{n} + \epsilon)$ with $\epsilon > 0$ as small as possible, such that the condition on a confidence region from the definition is satisfied:

$$\begin{aligned}
&\inf_{\theta\in\Theta} P_\theta\left(x\in\mathcal{X}\colon \tau(\theta)\in C(x)\right)\geq 1-\alpha\\
\Leftrightarrow &\inf_{\theta\in[0,1]} P_\theta\left(x\in\mathcal{X}\colon \left|\theta-\frac{x}{n}\right|<\epsilon\right)\geq 1-\alpha\\
\Leftrightarrow &\inf_{\theta\in[0,1]} P_\theta\left(x\in\mathcal{X}\colon \left|\theta-\frac{x}{n}\right|\geq\epsilon\right)\leq\alpha
\end{aligned}\tag{204}$$

In words: we are to determine ϵ (as a function of α, and as small as possible) so that $\left|\theta-\frac{x}{n}\right|\geq\epsilon$ holds for fewer than a proportion α of the possible samples x. With $\alpha=0.05$, say (a common choice), that means our estimation error exceeds ϵ in only 5% of samples.

The shape of this inequality may remind you of Chebyshev's inequality, which we met right after introducing the variance. For the estimator $T(x)=\frac{x}{n}$ it says

$$P_\theta\left(|\mathbb{E}(T)-T|\geq c\right)\leq\frac{\mathbb{V}T}{c^2}\tag{205}$$

Naturally we would like to take $c=\epsilon$ and choose ϵ as a function of α so that nothing but α is left on the right-hand side. We have $\mathbb{V}T=\mathbb{V}\left(\frac{x}{n}\right)=\frac{\theta(1-\theta)}{n}$, which annoyingly still depends on θ – exactly the thing we do not know!

So we bound $\theta(1-\theta)\leq\frac{1}{4}$ (easy to check: equality at $\theta=\frac{1}{2}$, strict inequality everywhere else).

Putting it all together, ϵ is well chosen if

$$\begin{aligned}
&\frac{\mathbb{V}T}{\epsilon^2}\leq\alpha\\
\Leftarrow&\frac{1}{4n\epsilon^2}\leq\alpha\\
\Leftrightarrow&\frac{1}{4n\alpha}\leq\epsilon^2\\
\Leftrightarrow&\frac{1}{2\sqrt{n\alpha}}\leq\epsilon\\
\Leftarrow&\frac{1}{2\sqrt{n\alpha}}=:\epsilon.
\end{aligned}\tag{206}$$

In our concrete example, with $\alpha=0.05$ and $n=100$, $\epsilon:=\frac{1}{2\sqrt{5}}\approx 0.2236$ does what we want. For us that means: for the concrete observation $k=45$, not only the point estimate $\frac{45}{100}$ is plausible; we also regard roughly $\frac{23}{100}$ and $\frac{67}{100}$ as plausible. So the coin could well be fair, and we have merely caught an unlucky observation.

So confidence intervals obtained from Chebyshev's inequality are very wide. There are several better (and considerably more complicated!) methods for obtaining narrower ones.

But we also see a simple relationship between ϵ, n and α: making α smaller (that is, demanding more certainty) makes ϵ bigger; making n bigger (taking larger samples) makes ϵ smaller.

With that we can compute, for instance, how large the sample has to be before we can say with certainty at least $1-\alpha$ that our coin is not fair. Again, concretely, suppose one sample has given us the estimate $\theta=0.45$.

$$\begin{aligned}
&0.45+\epsilon<0.5\\
\Leftrightarrow&0.45+\frac{1}{2\sqrt{n\alpha}}<0.5\\
\Leftrightarrow&\frac{1}{2\sqrt{n\alpha}}<0.05\\
\Leftrightarrow&2\sqrt{n\alpha}>\frac{100}{5}\\
\Leftrightarrow&\sqrt{n\alpha}>10\\
\Leftrightarrow&n\alpha>100\\
\Leftrightarrow&n>\frac{100}{\alpha}
\end{aligned}\tag{207}$$

and putting in $\alpha=0.05$, the sample has to be larger than 2000 before we can reject the claim that the coin is fair. With a cleverer procedure for determining narrower confidence intervals, a smaller sample would do.

Example

A sample from a normal distribution: Suppose the real-valued random variable Z is normally distributed with unknown expected value μ and unknown variance σ^2. We draw n independent, identically distributed samples of Z and obtain a random variable X with values in $\mathbb{R}^n$, that is $X = (X_1, \ldots, X_n)$, where $X_i \sim Z$.

We write $\texttt{mean}(X) = \sum_{i=1}^{n} \frac{X_i}{n}$ for the sample mean (careful: this is now a random variable, computed from the random sample X) and $S^*(X)^2 := \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \texttt{mean}(X))^2$ for the sample variance (with the Bessel correction, i.e. $n-1$ in the denominator, so that the estimate is unbiased).

We now want to know how well $\texttt{mean}(X)$ estimates $\mu(Z)$ – how far does our confidence, our trust, reach?

The random variable $\texttt{mean}(X)$ is normally distributed, with mean $\mu(Z)$ and standard deviation $\frac{\sigma(Z)}{\sqrt{n}}$. The estimation error, measured with the standard deviation of $\texttt{mean}(X)$ as the 'unit', is

$$\frac{\texttt{mean}(X) - \mu(Z)}{\sigma(Z)/\sqrt{n}} \tag{208}$$

and is therefore standard normally distributed. The trouble with this quantity is that we do not actually know $\sigma(Z)$, and yet we would like to look at the error of our estimate of $\mu(Z)$. So we have to replace $\sigma(Z)$ by its estimate $S^*(X)$.

The random variable 'sample variance' $S^*(X)^2$ is distributed so that $\frac{n-1}{\sigma(Z)^2} S^*(X)^2$ follows a χ^2 **distribution** with $n-1$ degrees of freedom (a special case of the gamma distribution).

The probability density function of the χ^2 distribution need not concern us at the moment (and in particular is not part of the course proper); for completeness, here it is:

$$f(x) = \frac{1}{2^{\frac{k}{2}} \Gamma\left(\frac{k}{2}\right)} x^{\frac{k}{2}-1} e^{-\frac{x}{2}} \tag{209}$$

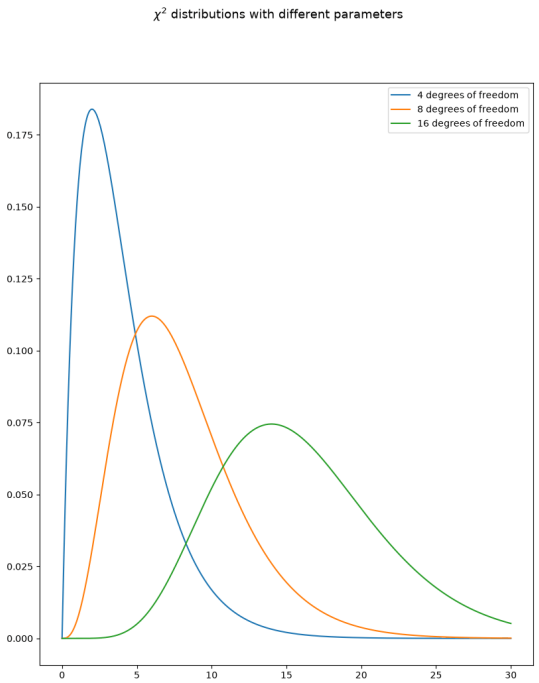


Figure 24: Chi-squared distributions with various parameters (degrees of freedom)

The difference between the estimated mean and the actual expected value, measured with the sample standard deviation as the 'unit', is the estimated estimation error T, the statistic we look at next:

$$T = \frac{\texttt{mean}(X) - \mu(Z)}{S^*(X)/\sqrt{n}}. \tag{210}$$

The random variable T follows a **Student t distribution**, which at first glance looks like a normal distribution but has heavier *tails*: it takes values far from the expected value with higher probability. Its only parameter is the number of degrees of freedom $\nu > 0$, here $\nu = n - 1$. For ν of at least 3 the expected value is 0 and the variance is $\nu/(\nu - 2)$.

The probability density function of the Student t distribution matters little to us for now; for completeness, this is what it looks like:

$$f(t) = \frac{1}{\sqrt{\nu} B\left(\frac{1}{2}, \frac{\nu}{2}\right)} \left(1 + \frac{t^2}{\nu}\right)^{-\frac{(\nu+1)}{2}} \tag{211}$$

where $B(x, y) = \dfrac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$ is the beta function and the gamma function Γ, defined on all of $\mathbb{R}$, interpolates $\Gamma(n+1) = n!$ so that $\Gamma(x)x = \Gamma(x+1)$ always holds (all of which we may promptly forget again).

As $n \to \infty$ the Student t distribution converges to the standard normal distribution. So the gap between the t distribution and the normal distribution encodes exactly the uncertainty that a small sample brings with it.

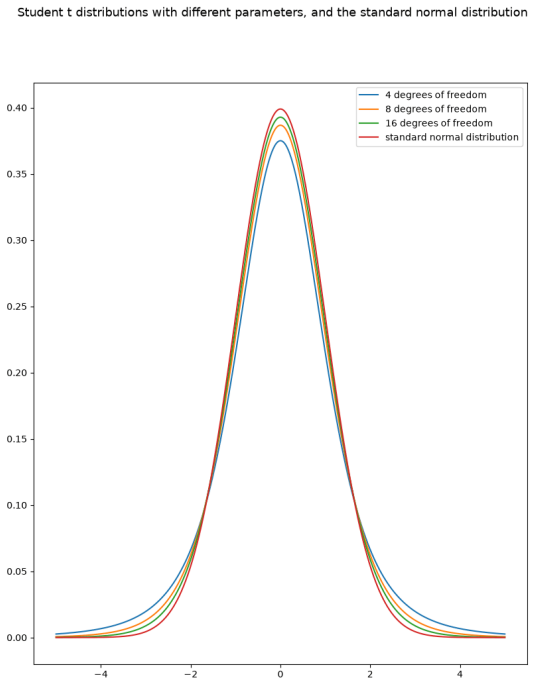


Figure 25: Student t distributions with various parameters (degrees of freedom)

We are now interested in where the probability of a large estimation error $T = \dfrac{\texttt{mean}(X) - \mu(Z)}{S^*(X)/\sqrt{n}}$ is small. Now this estimation error does not depend on $\mu(Z)$ – more precisely, T is independent of $\mu(Z)$. That may seem counter-intuitive, since $\mu(Z)$ appears in the formula, but it is true all the same. The statistic is said to be **distribution-free**.

For a given α we now choose t as the $(1 - \frac{\alpha}{2})$-*quantile* of the t distribution with $n - 1$ degrees of freedom, that is, the number $t > 0$ with $P(T \leq t) = 1 - \frac{\alpha}{2}$; by the symmetry of the distribution we then have $P(|T| \leq t) = 1 - \alpha$. This used to be looked up in a table; today we ask the computer (`scipy.stats.t.ppf(1 - alpha/2, n - 1)`). Then, with $\epsilon := t\frac{S^*(x)}{\sqrt{n}}$, a confidence interval for $\mu(Z)$ is given by

$$C(x) = (\texttt{mean}(x) - \epsilon, \texttt{mean}(x) + \epsilon). \tag{212}$$

So we see that constructing a confidence region can get complicated quickly. On the other hand, the procedure described here (standardise, then apply the quantile transformation) carries over to other situations as well – which we would rather leave to the statisticians.

By approximating the binomial distribution with a normal distribution and then applying the method just discussed for confidence intervals of a normal distribution, one also gets better confidence intervals for a binomial distribution – though here again more complicated procedures yield considerably better ones.

It is enough for us to know where the χ^2 distribution and the Student t distribution come from; the concrete probability densities we may forget again – fortunately the statisticians have worked them out for us and put them into our Python packages. What matters above all is that the naive assumption that the estimation error is in turn normally distributed does not hold in general. We will meet these distributions again when we discuss statistical tests, which in a sense build on the construction of confidence intervals.

7.6.4 Tests

Definition
Let $(\mathcal{X}, P_\theta : \theta \in \Theta)$ be a statistical model and $\Theta = \Theta_0 \cup \Theta_1$ a disjoint decomposition of the parameter space into the **null hypothesis** $\theta \in \Theta_0$ and the **alternative** $\theta \in \Theta_1$. Every statistic $\phi\colon \mathcal{X} \to [0,1]$ (which we read as a decision rule) is called a **test** of Θ_0 against Θ_1. A test is called **non-randomised** if $\phi(x) = 0$ or $\phi(x) = 1$ for all $x \in \mathcal{X}$, and **randomised** otherwise.

Example
Weighing apples. A supplier promises apples of at least 150 g on average; we assume the weights to be normally distributed and weigh a sample x. Three tests for the same sort of question:

1. $\Theta_0 = (-\infty, 150]$ ("the apples weigh at most 150 g on average"), $\Theta_1 = (150, \infty)$. One test: $\phi(x) = 1$ if $\texttt{mean}(x) > 150$, otherwise 0.
2. The same hypothesis, but a more cautious test: $\phi(x) = 1$ only once $\texttt{mean}(x) > 160$. It rejects less often, so it errs less often when Θ_0 is true and misses more often when Θ_1 is true.
3. $\Theta_0 = \{150\}$ ("exactly 150 g"), $\Theta_1 = \mathbb{R} \setminus \{150\}$. One test: $\phi(x) = 0$ if 150 lies in the interval $\texttt{mean}(x) \pm 1.96\,\sigma/\sqrt{n}$ from the section *From the sample to the population*, otherwise 1.

The next definition gives names to the two errors in version 2; version 3 is the z-test described below.

Definition
The probability of a **type I error** – rejecting the null hypothesis (that is, $\phi(x) = 1$) even though the true parameter is $\theta \in \Theta_0$ – is $\sup_{\theta \in \Theta_0} \mathbb{E}_\theta(\phi)$ and is also called the **effective level** of the test. For any α greater than or equal to the effective level, the test is also called a test of level α.

The function $G_\phi\colon \Theta \to [0,1]$ given by $G_\phi(\theta) = \mathbb{E}_\theta(\phi)$ is called the **power function** of the test. $1 - G_\phi(\theta)$ is the probability of a **type II error** – accepting the null hypothesis (that is, $\phi(x) < 1$) even though the true parameter is $\theta \in \Theta_1$. For $\theta \in \Theta_1$, $G_\phi(\theta)$ is also called the **power** of ϕ at θ.

In the apple example the test with threshold 160 has a smaller effective level than the one with threshold 150, but also less power.

Definition
A test ϕ is called a **uniformly best test at level** α if it is of level α and every other test of level α has smaller or equal power.

So a uniformly best test has probability at most α of a type I error and, among all tests with that property, the smallest probability of a type II error.

7.6.5 Looking ahead: from set estimator to test

Note
Looking ahead, not part of the course proper (see the note at *Confidence intervals*).

When we discussed a confidence interval for the binomial model we already saw that it can help us reach a decision (concretely: should I regard a coin with 45/100 successes as 'unfair', or does sampling error explain the deviation from 50/100? Or indeed: how many more samples do I need to settle the question for good?)

This relationship can be systematised: using the vocabulary of the shape parameter, a set estimator can be constructed from any test. Here we look only at the other direction, how a test is constructed from a set estimator.

Our starting point is (for simplicity) a parametric statistical model $(\mathcal{X}, P_\theta : \theta \in \Theta)$ and an interval estimator (that is, a confidence interval at level $1-\alpha$) $C\colon \mathcal{X} \to \mathcal{P}(\Theta)$ for the parameter θ. In addition we have a hypothetical parameter value $\theta_0 \in \Theta$ that does *not* come from the data but from somewhere else: $\theta_0 = \frac{1}{2}$ for the claim 'the coin is fair', say, or a value from the literature or from an earlier experiment.

The hypothesis is now that θ_0 is the true parameter. Strictly speaking, this hypothesis is of course probably always false; what interests us is when it can be rejected with *certainty*. So we set $\Theta_0 := \{\theta_0\}$ and $\Theta_1 := \Theta - \{\theta_0\}$.

From this we construct the test $T_C\colon \mathcal{X} \to [0,1]$ by

$$T_C := 1_{\{x \in \mathcal{X} \mid \theta_0 \notin C(x)\}} \text{ (indicator function)} \tag{213}$$

$$T_C(x) = \begin{cases} 0 & \theta_0 \in C(x) \\ 1 & \theta_0 \notin C(x) \end{cases} \tag{214}$$

So the procedure is: from the data $x \in \mathcal{X}$ we compute the confidence interval $C(x)$ and ask whether θ_0 lies in it. If it does, we keep the hypothesis 'θ_0 is the true parameter'; if not, we conclude 'θ_0 cannot be the true parameter'. Since C is a confidence interval at level $1-\alpha$, we have $P_{\theta_0}(\theta_0 \notin C(x)) \leq \alpha$, so the test T_C has level α.

What matters is that θ_0 was not itself estimated from the same data: a point estimator $\hat{\theta}(x)$ – the midpoint of $C(x)$, say – always lies in $C(x)$, and such a test would never reject. The test is interesting precisely when θ_0 comes from an independent source, for example when we ask whether or not the data are uniformly or normally distributed.

7.6.6 Looking ahead: best simple tests

Note
Looking ahead, not part of the course proper (see the note at *Confidence intervals*).

We have already seen the definition of level and power, and the usual procedure of looking for the most powerful test at a fixed level.

Definition
If there are only two possible parameters (one then speaks of **simple hypotheses**), that is $\Theta_0 = \{\theta_0\}$ and $\Theta_1 = \{\theta_1\}$, and f_0 and f_1 are the density functions of the distributions that belong to these parameters, then

$$R(x) := \frac{f_1(x)}{f_0(x)} \tag{215}$$

is called the **likelihood ratio**, and a test $\phi\colon \mathcal{X} \to [0,1]$ of the form

$$\phi(x) = \begin{cases} 0 & R(x) < c \\ 1 & R(x) > c \end{cases} \tag{216}$$

with $c \in \mathbb{R}$ is called a **Neyman–Pearson test** with threshold c.

Why is $R(x)$ called the likelihood ratio? Because $f_1(x)$ is a measure of the probability $P(D|H_1)$ and $f_0(x)$ is a measure of the probability $P(D|H_0)$, so R is the ratio of the two likelihoods. If $f_0(x) = 0$, we simply set $R = \infty$.

The case $R(x) = c$ usually has probability 0, but it can be handled separately if need be (with a suitable value of $\phi(x)$ between 0 and 1).

The **Neyman–Pearson lemma** says that for every simple hypothesis (that is, every Θ with two elements) and every level α there is a Neyman–Pearson test (that is, a suitable $c \in \mathbb{R}$) that uses up the level in full (has effective level α), and that every such test is a uniformly best test at level α.

Stein's lemma says something about how much additional samples help to make a test more informative. Fix a level α. Take a sequence of observations and, for each n, a Neyman–Pearson test ϕ_n of level α that depends only on the first n of them. Then the power of ϕ_n tends to 1 as the observations accumulate (that is, as $n \to \infty$), and it does so exponentially fast. More precisely:

$$\lim_{n\to\infty} \frac{1}{n} \log\left(1 - \mathbb{E}_1(\phi_n)\right) = -H(P_0 \| P_1) \tag{217}$$

So the higher the relative entropy $H(P_0\|P_1)$ between the two possible distributions, the more power every single additional sample buys.

The connection runs through the definitions of the Neyman–Pearson test and of relative entropy: the expression $-f_0(x) \log R(x) = f_0(x) \log\left(\frac{f_0(x)}{f_1(x)}\right)$ appears in the definition of continuous relative entropy.

The take-home message: for simple hypotheses best tests always exist, and relative entropy quantifies how far enlarging the sample pushes down the probability of a type II error.

7.6.7 The z-test

If your data points are independent (and identically distributed) – a sample of a random variable, say – and normally distributed, and you know the standard deviation σ, you can run a **z-test** (also called the **Gauss test**). It is the 'simpler' variant of the t-test (see below). If you do not know σ and put in the sample standard deviation instead, that is still defensible for large n, because the t distribution is then practically a normal distribution; how crude the widespread rule of thumb "$n > 30$" is for this purpose is described in the chapter The normal distribution under *The "$N \geq 30$" myth.*

The test asks whether a sample x follows a normal distribution with mean μ_0. For $\mu_0 = \texttt{mean}(x)$ (the maximum likelihood estimator) there is nothing to test – we conclude at once that $\mu = \mu_0$ is plausible. But what about values $\neq \texttt{mean}(x)$? Suppose our sample *almost* follows a standard normal distribution; then it would make sense to test whether the hypothesis of a standard normal distribution should be accepted or rejected, even though the maximum likelihood estimator gives something other than 0 for μ.

The **z-statistic** is

$$Z(x) = \frac{\texttt{mean}(x) - \mu_0}{\sigma/\sqrt{n}} \tag{218}$$

In the z-test σ is treated as known; for large samples the sample standard deviation $\sigma^*(x)$ takes its place. The unknown is μ, and the null hypothesis reads H_0: $\mu = \mu_0$. In the t-test below, by contrast, σ is treated as unknown, and the error made in estimating σ is built into the distribution of the test statistic. The random variable Z is standard normally distributed.

In a two-sided z-test the null hypothesis is $H_0 := (\mu = \mu_0)$, which corresponds to $\Theta_0 = \{\mu_0\}$, and the alternative is $H_1 = (\mu \neq \mu_0)$, which corresponds to $\Theta_1 = \mathbb{R} \setminus \{\mu_0\}$.

We choose a level α, say $\alpha = 0.05$. Since we want a two-sided test, we look up the z-value for $1 - \alpha/2$ (in a table of z-values, say, or with the quantile function `scipy.stats.norm.ppf(1 - alpha/2)`). For $\alpha = 0.05$ the z-value is ± 1.96. The z-value $z(p)$ is the p-quantile of the standard normal distribution, so $P(Z \leq z(p)) = p$, here concretely $P(Z \leq 1.96) = 0.975$. Because the standard normal distribution is symmetric, we also have $P(Z \geq -1.96) = 0.975$ and $P(-1.96 \leq Z \leq 1.96) = 0.95$.

So we compute the z-statistic of our sample and compare it with the z-value 1.96. If it lies in between, that is $-1.96 \leq Z(x) \leq 1.96$, we cannot reject the null hypothesis (which does not mean we accept it; see Significance below). If the z-statistic lies outside $[-1.96, 1.96]$, we reject the hypothesis.

7.6.8 The Student t-test

The two tests side by side:

	z-test	t-test
σ	known (for large n: $\sigma^*(x)$ substituted)	unknown, estimated along the way
distribution of the statistic under H_0	standard normal distribution	Student t distribution with $n-1$ degrees of freedom
when	σ known, otherwise only for large n	always; for large n practically the same test
SciPy	`norm.ppf`	`t.ppf`, `ttest_1samp`

With small samples the bias in the estimated parameters of a normal distribution makes itself felt, and we have to work with the actual distribution of the **t-statistic**. So if μ_0 is the parameter we want to test for, we consider

$$t(x) = \frac{\texttt{mean}(x) - \mu_0}{\sigma^*(x)/\sqrt{n}} \tag{219}$$

Here $\sigma^*(x)$ is the unbiased estimator of σ.

As we saw when constructing confidence intervals, t follows a Student t distribution. To carry out the test at level α we therefore look up the t-value for that level in a table, or compute it in Python with `scipy.stats.t.ppf(p, n-1)` (ppf stands for **percent point function**; it gives us the p-quantile of the t distribution with $n-1$ degrees of freedom).

We compute the t-statistic on the data x and compare it with the 'critical' t-value computed beforehand. If the t-statistic lies within $\pm$ the critical t-value, we cannot reject the hypothesis (that μ_0 is the expected value of the distribution); if it lies outside, we reject it.

We can also have SciPy compute this two-sided t-test directly: `scipy.stats.ttest_1samp(x, mu)`

Strictly speaking the t-test can do more: instead of comparing the distribution of x with a normal distribution of mean μ and the same variance as x, we can compare two samples x, y (not necessarily of equal size) and ask whether their means agree, with `scipy.stats.ttest_ind(x, y)` – assuming both samples are normally distributed with equal variance and independent of each other. The case $y =$ `mu` is a special case of this (but calling it with that y only returns `NaN`s, because the variance of a one-element sample is 0; hence `ttest_1samp`).

The return value of `scipy.stats.ttest_1samp(x, mu)` is a `TtestResult` object with the attributes `statistic`, `pvalue` and `df` (the number of degrees of freedom, that is $n-1$) as well as the method `confidence_interval(level=0.95)`. (Older SciPy versions returned a `namedtuple` with only `statistic` and `pvalue`; those two attributes are unchanged.) The attribute `statistic` is the computed t-statistic (no α has been involved yet, so you would still have to compare it by hand with the critical t-value to get the test result). The attribute `pvalue` is the **p-value** (also called the **significance**) of the test result: the probability, under H_0, of observing a test statistic at least as extreme as the one actually observed – for the two-sided test, $p = P(|T| \geq |t_{\text{obs}}| \,|\, H_0)$. In particular the p-value is *not* the probability of the concrete observation (which for continuous distributions would always be 0), nor the probability $P(H_0|D)$ that the null hypothesis is true. The name comes from 'probabilitas', and the word 'significance' naturally carries certain associations. If $p < \alpha$ we reject the null hypothesis; otherwise we cannot reject it, which is not the same as accepting it: a high p-value only says that the sample cannot tell the deviation apart from a sampling error (see the second warning in the section Significance below).

For the apple example from the section *Tests*, with a simulated sample of 40 normally distributed weights and the null hypothesis $\mu_0 = 150$ as in the third test there:

```
from scipy.stats import ttest_1samp, norm
x = norm.rvs(155, 20, size=40, random_state=0)
res = ttest_1samp(x, 150)
res.statistic, res.pvalue, res.df, res.confidence_interval()
```

```
(3.3007, 0.00207, 39, ConfidenceInterval(low=154.36, high=168.15))
```

The p-value is below 0.05, so we reject $\mu_0 = 150$; fittingly, 150 lies outside the confidence interval. `confidence_interval()` is exactly the interval from the section *From the sample to the population*, $\texttt{mean}(x) \pm t \cdot \sigma^*(x)/\sqrt{n}$, except that the quantile $t = \texttt{t.ppf}(0.975, 39) \approx 2.02$ of the t distribution replaces 1.96.

For the two-sample case there is also a variant that needs only the summary measures:

```
ttest_ind_from_stats(mean1, std1, nobs1,
                     mean2, std2, nobs2,
                     equal_var=True,
                     alternative="two -sided")
```

where `alternative` says whether a two-sided or a one-sided (`less` or `greater`) test is to be run, and `nobs` is the number of observations. If `equal_var` is set to `False`, the assumption of equal variance is dropped, and what runs is then not the ordinary t-test but Welch's t-test, which computes the degrees of freedom differently.

If two samples do not satisfy the basic assumption of equal variance, a transformation of the scale (taking logarithms, for instance) can sometimes bring it about, at least approximately, and so make the t-test useful in these cases too. If the variances differ markedly, keep your hands off the t-test altogether: it is not informative.

7.6.9 Looking ahead: ANOVA

ANOVA, short for **Analysis of Variance**, generalises the t-test to more than 2 sample groups. Its assumptions are the same as for the t-test:

- The observations are independent of one another
- The residuals (the differences $x_i - \texttt{mean}(x)$, an estimate of the actual sampling error) are normally distributed
- Homoscedasticity, that is, homogeneity of variance: the variances of the sample groups are equal.

It is usually applied to a categorical variable with k levels: the sample is split into k groups by level and ANOVA is run on these groups. If a significant deviation shows up (that is, a p-value below 0.05), we reject the null hypothesis that the groups have equal expected values – and conclude that the categorical variable probably influences the expected value.

If no significant deviation shows up (p-value of 0.05 or more), we may well see differences between the various means, but no larger than a sampling error already explains well – no influence of the categorical variable can be seen this way.

Careful: that does not mean the categorical variable has no influence on the means! It only means that our sample cannot tell such an influence apart from a sampling error. 'We know that we know nothing'.

In Python we use `scipy.stats.f_oneway`.

7.6.10 Significance

Of all the notions in this chapter this is the most important, because it leaves statistics behind and turns up in newspapers, studies and meetings. There it is almost always used as though there were a number behind it. There is not.

Definition
The result of a statistical test is called **statistically significant** if the sample data deviate from the null hypothesis so strongly that, by a rule *fixed in advance*, that assumption is rejected.

Remark
There is no number in this definition. The rule may be arbitrarily strict or arbitrarily silly; "significant" only says that it was triggered. Whether that means anything can only be judged once you know the rule, the level and the null hypothesis. So anyone who says "the difference is significant" has thought up a context in which that sentence is permitted; without it, the sentence is empty. From now on, use the word the way you use "continuous": only in its technical sense.

The usual rule is "$p < \alpha$" with level $\alpha = 0.05$. That is a convention of the journals, not a constant of nature. It does at least have a rationale: a normally distributed quantity deviates from its expected value by more than just under two standard deviations ($1.96\,\sigma$) with probability only 5 %, so a result

beyond that line is "conspicuous". For delicate questions more is demanded: $p \leq 0.01$ is called **very significant**, $p \leq 0.001$ **highly significant** (about three standard deviations). Particle physics counts in standard deviations outright and demands five of them for a discovery.

Three warnings go with the definition:

1. **Rejecting H_0 does not mean accepting H_1.** The test has only said that the data are improbable under H_0; it has said nothing about the alternative, and certainly nothing about whether H_1 is the only other explanation.

2. **A high p-value says nothing.** "Not significant" only means that the sample cannot tell the deviation apart from a sampling error. It does not mean that there is no deviation (see the remark on ANOVA above).

3. **Without a well-chosen null hypothesis the p-value means nothing.** Anyone who picks an H_0 that nobody believes anyway gets a significant result for free. Which hypothesis to formulate as H_0 is the subject of the next section.

Once more, as a reminder: a **type I error** is a false rejection of the null hypothesis. Small p-values thus tempt you into a type I error, and anyone who runs enough tests will eventually find a significant one:

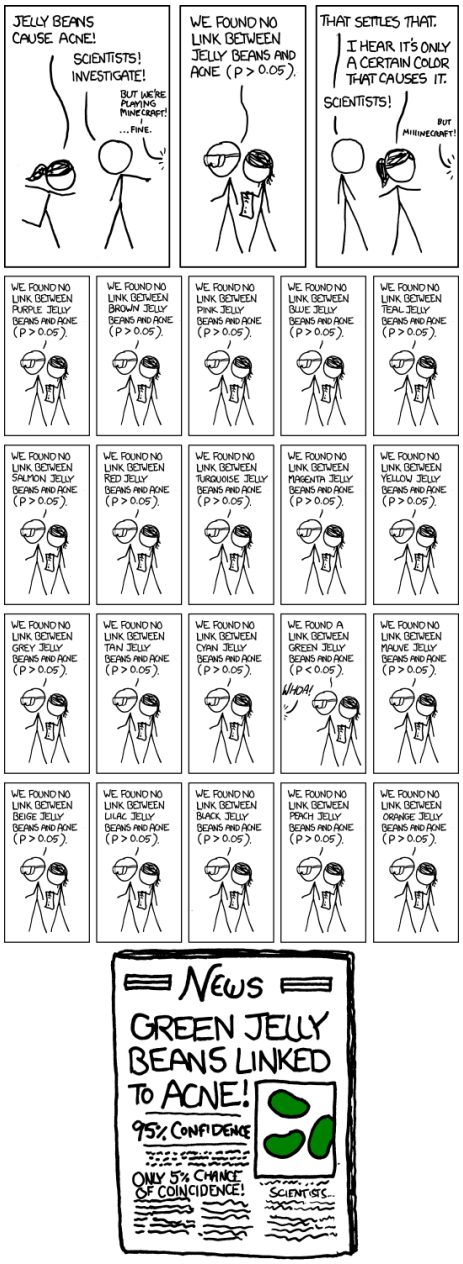


Figure 26: Randall Munroe, xkcd 882 "Significant" (CC BY-NC 2.5) (So, uh, we did the green study again and got no link. It was probably a– "RESEARCH CONFLICTED ON GREEN JELLY BEAN/ACNE LINK; MORE STUDY RECOMMENDED!")

To be published in important journals, studies often have to show significant results. That is a problem, because

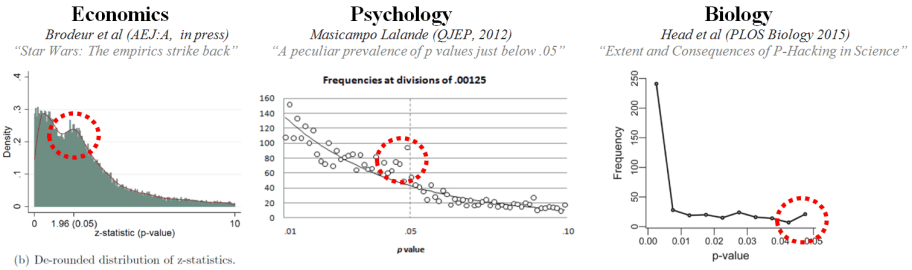
- significant results, too, can be pure accident (the comic),
- there are tricks for shrinking the p-value systematically without learning anything more about the question (**p-value hacking**, cf. Figure 27): testing many hypotheses and reporting only the significant one, extending the sample until p drops below 0.05, removing outliers after the fact,
- non-significant results can be scientifically relevant too, but are published less often (**publication bias**),
- the p-value is very often misread, and not only by students: in the survey of Haller and Krauss [2002], even most of the methodology lecturers got at least one of six statements about the meaning of a significant result wrong.

Most Published Research Findings Are False
Ioannidis [2005]

> "the probability that a research finding is indeed true depends on the prior probability of it being true (before doing the study), the statistical power of the study, and the level of statistical significance"

The less plausible the hypothesis (the lower its prior probability), the more likely an exciting finding is to turn out a false alarm, regardless of the p-value.

A very important safeguard against p-value hacking is the pre-registration of studies: hypothesis, level and analysis are fixed before the data exist. That is exactly what the definition above demands with its "rule fixed in advance".



(b) De-rounded distribution of z-statistics.

Figure 27: Compilation by Uri Simonsohn, 2015

7.6.11 What to formulate as hypothesis, what as alternative?

The usual procedure is to formulate the scientific insight you would like to establish as the rejection of the null hypothesis. It makes a difference which statement becomes the hypothesis and which the alternative, because we defined best tests so that the type I and type II errors play different roles: the type I error is held exactly at the level α, and the type II error is minimised subject to that constraint.

Example
Imagine we have found a new drug that we believe makes people cleverer. To show this, we proceed as follows: we design an experiment in which the claim that our drug makes people cleverer could be refuted. To that end we operationalise "becoming cleverer" by, say, an examination situation with cognitive tasks. The participants are randomly split into two groups. One group gets a placebo, the other the drug; everything else is the same, and nobody (not even the people running the study) knows who is in which group. Afterwards the data are analysed – performance in the examination is one variable, group membership the other.

The null hypothesis is now: the drug has no effect. The alternative: the drug makes people cleverer. If we assume performance to be normally distributed, we can run a two-sample t-test. If $p < 0.05$ we reject the null hypothesis and conclude that the observation cannot be chance – that the variable 'group membership' has a significant influence on performance.

Even with $p < 0.05$ a type I error can of course still occur, because $0.05 \neq 0$. The null hypothesis was $\mu_1 = \mu_2$, where the μ_i are the means of the two groups; the alternative is $\mu_1 \neq \mu_2$. When we derived the t-test we set up the t-statistic by comparing the sample mean with a fixed μ_0. And there the roles of hypothesis and alternative cannot be swapped!

You can see that the two cannot be swapped in the ordinary-language version of the hypothesis too: "the drug has no effect" really means "whether or not you take the drug has no influence on test performance". How would you formalise "has an influence on test performance" mathematically?

What you can do instead is test for an explicitly described relationship, for example the hypothesis that after taking the drug you score exactly 10 points more in the test – that is, the equation $\mu_1 + 10 = \mu_2$. Note that this is again an equation, whereas $\mu_1 \neq \mu_2$ cannot be tested in the same way.

7.6.12 Looking ahead: applying further useful tests

There is a whole series of further tests we might need, along with their assumptions and how to run them in Python on a pandas dataset. The applications lie in exploratory data analysis (how is a variable

distributed?), in data cleaning (where are the outliers? which variables are so strongly correlated that I keep only one of them?), in data analysis (which statistical dependencies can be modelled?) and in prediction (can I express one variable – the target variable – through the others? that is supervised learning).

Examining one sample

- Shapiro–Wilk: H_0: the sample is normally distributed Assumption: the sample is iid `scipy.stats.shapiro`

Comparing two samples

- Pearson correlation: H_0: the two samples are uncorrelated ($\rho = 0$) Assumption: both samples are iid and jointly normally distributed `scipy.stats.pearsonr`
- Chi-squared: H_0: the two samples are independent Assumption: the variable under investigation is categorical, with an expected frequency of at least 5 in every cell of the contingency table `scipy.stats.chi2_contingency`

7.6.13 Digression: could a poll have answered the five-per-cent question?

> **Note**
> **Digression (not part of the course proper):** an application of the section *From the sample to the population* to a question everybody was asking in February 2025.

In the German federal election of 23 February 2025 the Bündnis Sahra Wagenknecht won 4.981 % of the second votes (the party vote). It was 9,529 votes (in German) short of entering the Bundestag, out of some 49 million. In the last week before the election twelve polls (in German), each with 1000 to 2500 respondents, had put the BSW between 4 and 6 %. Afterwards, the polls were said to have got it wrong. Had they?

Let us check with the recipe. With n respondents, a proportion $\hat{p} = 0.05$ has standard error $\sqrt{\hat{p}(1-\hat{p})/n}$:

```
import numpy as np
p = 0.05
for n in [1000, 2000]:
    se = np.sqrt(p * (1 - p) / n)
    print(n, "respondents: standard error", round(100 * se, 2), "points,",
          "95% interval \pm", round(196 * se, 2), "points")
```

```
1000 respondents: standard error 0.69 points, 95% interval \pm 1.35 points
2000 respondents: standard error 0.49 points, 95% interval \pm 0.96 points
```

So a poll with 2000 respondents that reports 5 % is saying: somewhere between 4 and 6. All twelve polls lay in that range, and so did the election result. None of them "got it wrong"; they answered a question nobody had asked, namely whether the BSW stood at three per cent or at seven. The question people were asking was whether it lay above or below 5.000 %, with a true value of 4.981 %. To resolve two hundredths of a point the interval would have to shrink to ± 0.02 points, and because it shrinks only with $\sqrt{n}$ that would take

$$n \geq \left(\frac{1.96 \cdot \sqrt{0.05 \cdot 0.95}}{0.0002} \right)^2 \approx 4.6 \text{ million respondents,} \tag{220}$$

almost one in every ten eligible voters. That is no longer a poll, that is an election.

The recipe gives only the *lower* bound on the uncertainty, the pure sampling error. On top of that comes everything that violates the random-sample assumption: the people who answer the telephone or sit in an online panel are not a random selection of the electorate, so the institutes weight the raw data by age, region and past voting behaviour, each by its own method. Someone who says "BSW" on Tuesday may vote differently on Sunday, or not at all. And the question is "Who would you vote for if there were an election on Sunday?", not "Who did you vote for?". The spread from 4 to 6 between the institutes in the same week is partly sampling error and partly these differences of method, and the two can only

be told apart once you know how the institute weighted. That is exactly the context without which the number “5%” is not yet a datum.

What this leaves for reading polls is this: a value without an error statement is incomplete; the error statement for 1000 respondents and small parties is about ±1.4 points; and a question whose answer is finer than that interval cannot be answered by any poll, however reputable the institute.

7.7 Model selection: cross-validation and quality measures

When a method leaves one or more parameters open, there is not always a natural way to choose them – usually you simply have to try out what works best on the dataset in front of you.

For many real datasets several models are conceivable, and again there is not always a natural choice of the one likely to do best – here too, comparing different approaches is usually worth the effort.

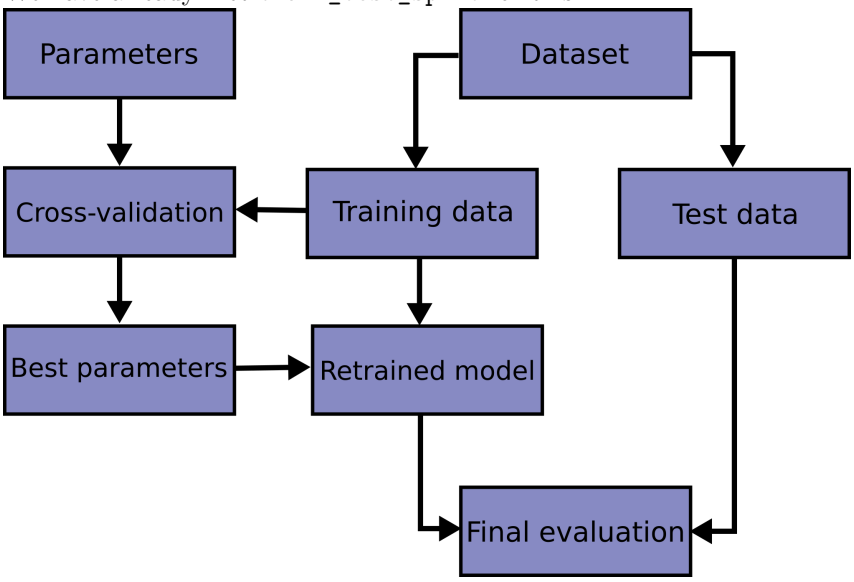
There are various ways of assessing which model is 'better' than the others. We have already seen that overfitting can be a problem: simply comparing the number of correctly classified samples does not necessarily pick out the model that generalises best.

7.7.1 Training, test and new data

The 'classical situation' in machine learning is this: while an ML system is being developed, you have access to data (the training data), which may already be labelled – that is, the desired output of the ML system is already part of the data. Afterwards (once training is over) the system is supposed to work on newly arriving data – and those are not labelled.

Since unlabelled data give no way of judging how well the ML system has done, part of the labelled training data is set aside for that judgement – the test data. What distinguishes the test data from the 'real' data that come later is precisely that they are labelled.

We have already met `train_test_split` for this.



(Figure from the scikit-learn documentation, BSD-3-Clause)

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import load_iris
x, y = load_iris(return_X_y=True)
```

Training, validation and test Suppose that, while developing an ML system, you compare 200 different variants by fitting each on the training set and analysing its performance on the test set. That inevitably produces an implicit fit to the test set, and hence an ML system that generalises badly. The test set is said to **leak** (information leakage).

To be able to compare variants and still keep a test set for judging how well the system works, the test set (or rather the training set) is split up further.

A typical setting: a competition withholds the target variable of the test set. To compare different models before the competition closes all the same, you can split off part of the training set as a validation set and keep it out of training.

The course *Introduction to Deep Learning* treats the train-test split, the validation set and the k-fold cross-validation of this chapter as known.

k-fold cross-validation Choosing a test set or a validation set brings two fundamental problems:

- you could be unlucky and end up with subsets of the data that are hardly representative

- less data for training will lead to worse results.

k-fold cross-validation tries to solve both problems by running the ML system several times – k times, to be precise – each time with a different split into training and test data. To do this, the training set is divided into k equal parts (called *folds*), and in the i-th of the k runs the i-th part serves as the test set (and is left out of training).

This optimises for models that generalise well to unseen data.

```
from sklearn.model_selection import KFold

from sklearn.datasets import load_iris
x, y = load_iris(return_X_y=True)

cv = KFold(n_splits=3, shuffle=True, random_state=23)

for train_i, test_i in cv.split(x):
    print(test_i)
    test_x = x[test_i]
    test_y = y[test_i]
    print("mean of class label:", np.mean(y[test_i]))
```

```
[  2   3   5   7   9  10  16  17  18  19  20  24  28  30  34  44  45  46
  47  51  52  54  59  63  64  67  69  73  75  81  87  93  95  99 100 102
 106 113 115 120 122 125 126 131 133 135 137 139 140 146]
mean of class label: 0.94
[  1   4   8  11  23  26  27  29  33  37  42  50  53  55  61  62  68  70
  71  72  76  77  80  82  84  85  88  89  92  97 101 104 105 107 109 110
 111 112 114 117 119 121 127 128 130 132 134 136 142 148]
mean of class label: 1.18
[  0   6  12  13  14  15  21  22  25  31  32  35  36  38  39  40  41  43
  48  49  56  57  58  60  65  66  74  78  79  83  86  90  91  94  96  98
 103 108 116 118 123 124 129 138 141 143 144 145 147 149]
mean of class label: 0.88
```

Stratified k-fold cross-validation In the supervised learning setting – that is, when you have labelled training data – it can make sense for each individual 'fold' of the k-fold cross-validation to hold all the classes in representative proportions. This is called **stratified k-fold cross-validation**.

```
from sklearn.model_selection import StratifiedKFold

cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

print("The following %d numbers should be quite similar:"
      % cv.get_n_splits())
for train_i, test_i in cv.split(x, y):
    print("mean of class label:", np.mean(y[test_i]))
```

```
The following 3 numbers should be quite similar:
mean of class label: 0.98
mean of class label: 1.0
mean of class label: 1.02
```

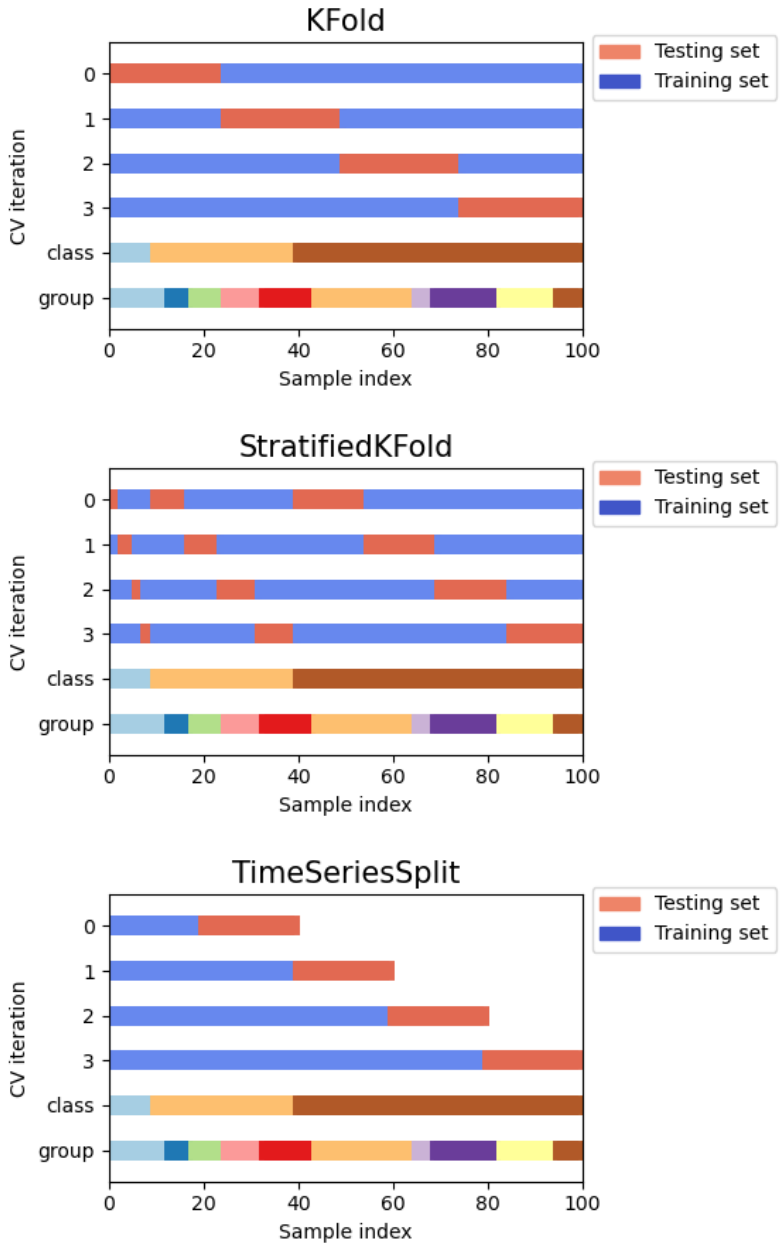
Time-series-aware cross-validation The cross-validation methods discussed so far assume that the data are independent and identically distributed – in particular, that every single sample is 'worth' as much as every other. If the data were collected over a period of time, however, that assumption is usually wrong. In many cases it can make sense to assume that past data have an influence on future data.

The naive approach would be to train on the first half of the data (in time) and use the second, later half as the test set.

Example
A model that predicts the mean surface temperature of the Earth from the concentrations of greenhouse gases in the atmosphere is fitted on the period 1800 to 2000 and then used to 'predict' the years 2000–2022, so that we can compare it with other models.

In scikit-learn you can use `TimeSeriesSplit`, which has the same syntax and semantics as `KFold` and `StratifiedKFold`; only the number of samples in the individual steps differs. There is also the integer parameter `gap`, which sets how many samples to leave out at the end of each training set. With continuous data, this stops the data points at the end from 'knowing' too much about the data points that follow them in the test set.

Visualising the cross-validation strategies Extract from the scikit-learn documentation (BSD-3-Clause):



7.7.2 Assessing model performance

We have already seen three essential approaches to assessing the quality of models:

- When the labels match – that is, when the model gives the classes $1, 2, ..$ with original labels $A, B, ..$ those same labels $A, B, ..$ and not, say, a permutation of them – we compared how often the model is right and how often wrong. That is what `sklearn.metrics.accuracy_score` does.
- When the labels are permuted (as we saw with k-means), we can work out which permutation gives the model the most 'correct' hits and work with that one (in effect undoing the permutation of the labels).
- When the labels are permuted, we can use a permutation-invariant score, such as the `normalized_mutual_info_scor` NMI. As a reminder:

$$NMI(U,V) = \frac{I(U;V)}{\texttt{mean}\big(H(U), H(V)\big)}, \qquad I(U;V) = H(U) + H(V) - H(U,V) \tag{221}$$

with the *mutual information* $I(U;V)$ from the chapter Relative entropy: it is 0 when the clustering and the classes have nothing to do with each other, and equals $H(U) = H(V)$ when either determines the other; the denominator normalises to $[0, 1]$.

Assessing a clustering If the true classes are known, we can assess whether each cluster holds samples of a single class only, or how far the clustering departs from that (homogeneity), and whether all the samples of one class really do land in a single cluster:

```
from sklearn.cluster import KMeans

y_pred = KMeans().fit(x).predict(x)

from sklearn.metrics import homogeneity_score,\
                            completeness_score

print("Homogeneity:", homogeneity_score(y, y_pred))
print("Completeness:", completeness_score(y, y_pred))

Homogeneity: 0.9073325384241996
Completeness: 0.5331535221818433
```

Why is the completeness so poor in this example? Because we did not give KMeans a number of clusters, so it settles on more than 3:

```
print("Number of clusters:", np.max(y_pred)+1)
print("Number of classes:", np.max(y)+1)

Number of clusters: 8
Number of classes: 3
```

We now repeat this with 3 clusters and watch the completeness rise while the homogeneity falls:

```
y_pred = KMeans(3).fit(x).predict(x)
assert np.max(y) == np.max(y_pred)

print("Homogeneity:", homogeneity_score(y, y_pred))
print("Completeness:", completeness_score(y, y_pred))

Homogeneity: 0.7514854021988338
Completeness: 0.7649861514489815
```

The harmonic mean (possibly weighted) of homogeneity and completeness is also called the **V-measure**, `v_measure_score`; one can show that it is equivalent to the normalised mutual information NMI.

Another way of summarising the classification quality of a clustering:

```
from sklearn.metrics import classification_report
print(classification_report(y, y_pred))
```

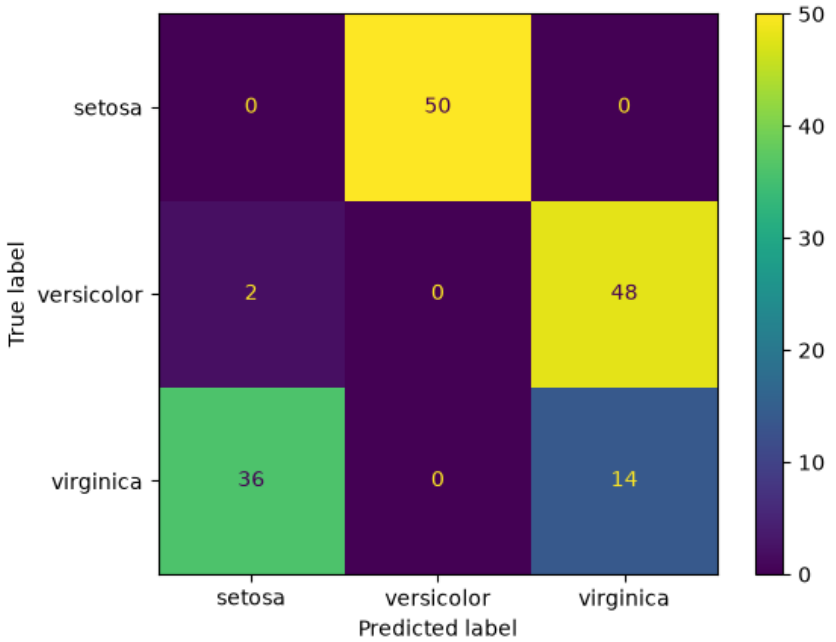
```
precision    recall  f1 -score   support

           0       0.00      0.00      0.00        50
           1       0.00      0.00      0.00        50
           2       0.23      0.28      0.25        50

    accuracy                           0.09       150
   macro avg       0.08      0.09      0.08       150
weighted avg       0.08      0.09      0.08       150
```

The columns of the `classification_report` are the summary measures from the chapter Titanic EDA, only now one set per class instead of a single overall set: **precision** is the proportion of correct predictions among all predictions of that class, **recall** the proportion of instances recognised among all instances of that class, **f1-score** the harmonic mean of the two, and **support** the number of instances of the class in the ground truth. The rows underneath average over the classes, once unweighted (*macro avg*) and once weighted by support (*weighted avg*).

For a clustering these numbers have to be read with care: if the cluster holding the class *versicolor* happens to carry the number 2, the report counts every one of those predictions as wrong, even though the clustering has found the class. The confusion matrix shows such a swap directly, as swapped rows:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

print(confusion_matrix(y, y_pred))
ConfusionMatrixDisplay.from_predictions(y, y_pred, display_labels=load_iris().target_names)
plt.show()
```

```
[[ 0 50  0]
 [ 2  0 48]
 [36  0 14]]
```



Assessing a regression A regression is assessed with the R^2 value, `r2_score`.

Despite what the name suggests, the R^2 value can perfectly well be negative; the formula is

$$R^2 = 1 - \frac{\sum_i (y_i - f(x_i, \beta))^2}{\sum_i (y_i - \texttt{mean}(y))^2} \tag{222}$$

If the model predicts the data perfectly, the numerator – the residual sum of squares – is 0, and hence $R^2 = 1$.

The constant model that always returns $f(x_i, \beta) = \texttt{mean}(y)$ (and so is the best fit among all constant models) has $R^2 = 0$. Models that do worse – any other constant model, for example – have negative R^2 values.

7.7.3 Hyperparameter search

Once you have worked out which estimators or methods you would like to use, which cross-validation you want and which scoring function is suitable, you still need a search strategy.

One possibility would be to pick two different hyperparameter settings at random and compare which performs better (in the sense of the chosen scoring function). Then pick two new ones at random near the better one, repeat, and so on.

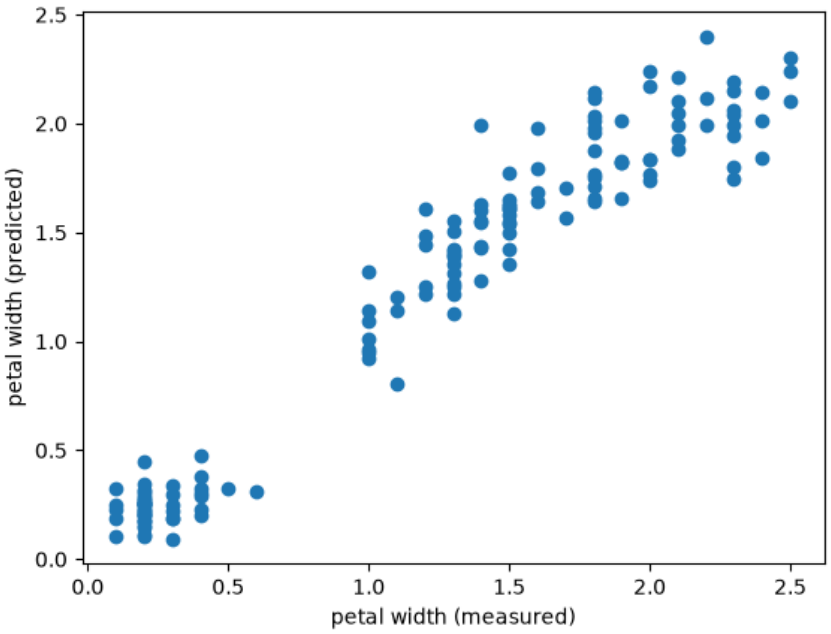
> **Example**
> Hyperparameters are, for instance, the number of clusters in k-means (the k), or the number of principal components to keep in PCA, or the regularisation constant α of a linear regression, or the regularisation parameter C of an SVM, or the parameters of the kernel in use, such as the degree of a polynomial kernel.

Grid search If we want to optimise several parameters at once, one option would be to hold one parameter fixed, optimise the other, and then optimise the one we held fixed at first. Clearly, though, that does not reach an optimum in the 'loss landscape'. It is better to search all possible combinations of the two parameters.

As an example we take a kernel ridge regression with an RBF kernel (see the chapter Kernel methods) and search for the kernel width γ and the regularisation α. The regression task: predict the petal width of the Iris data from the other three features, scored with R^2. (The class labels y would be the wrong target here: $0, 1, 2$ are categories, not numbers, and they call for classifiers and the scoring functions of the previous section.) Since both γ and α can be sensible across several orders of magnitude, we lay out the grid logarithmically.

```
from sklearn.model_selection import GridSearchCV
from sklearn.kernel_ridge import KernelRidge

# regression: petal width (column 3) from the other three features
x_reg, y_reg = x[:, :3], x[:, 3]

reg = KernelRidge(kernel='rbf')

param_grid = {
    "gamma": np.logspace( -3, 1, num=20),
    "alpha": np.power(2, np.arange( -6, 5, dtype=float)),
}

grid_search = GridSearchCV(reg, param_grid=param_grid, scoring='r2')
grid_search.fit(x_reg, y_reg)

print("best parameters:", grid_search.best_params_)
print("r2 =", grid_search.best_score_)

est = grid_search.best_estimator_

fig, ax = plt.subplots()
ax.scatter(y_reg, est.predict(x_reg))
ax.set_xlabel("petal width (measured)")
ax.set_ylabel("petal width (predicted)")
plt.show()
```

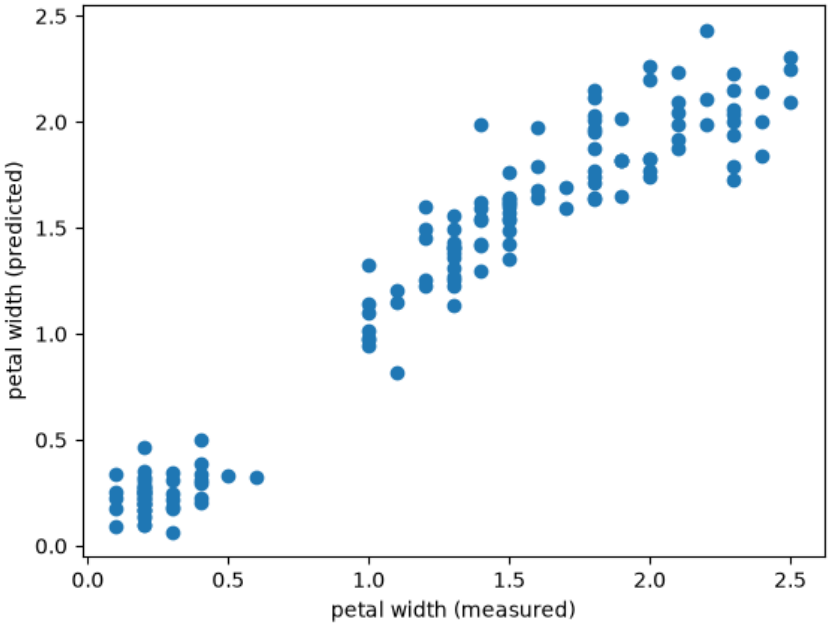
```
best parameters: {'alpha': np.float64(0.03125), 'gamma': np.float64(0.011288378916846888)}
r2 = 0.379506240834378
```



Randomised search One can also examine pairs of parameters sampled at random from a given distribution.

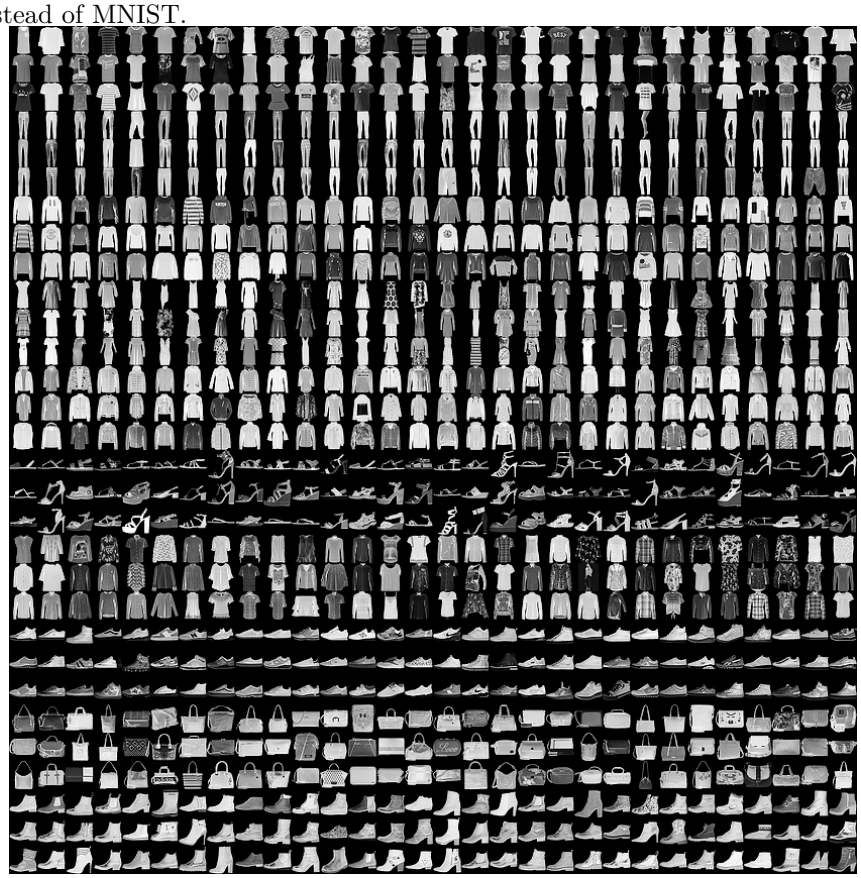
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import loguniform

param_dist = {
    "gamma": loguniform(1e -3, 1e1),
    "alpha": loguniform(1e -3, 1e0)
}

random_search = RandomizedSearchCV(
    reg,
    param_distributions=param_dist,
    n_iter=40
)

random_search.fit(x_reg, y_reg)

print("best parameters:", random_search.best_params_)
print("r2 =", random_search.best_score_)

est = random_search.best_estimator_

fig, ax = plt.subplots()
ax.scatter(y_reg, est.predict(x_reg))
ax.set_xlabel("petal width (measured)")
ax.set_ylabel("petal width (predicted)")
plt.show()
```

```
best parameters: {'alpha': np.float64(0.015121396655907246), 'gamma': np.float64
    (0.006684141999245426)}
r2 = 0.37632841090763786
```



7.7.4 An example application: Fashion MNIST

Every three rows form one class (T-shirts, trousers, and so on). The data come from Zalando Research. The trouble with the classical 'handwritten digits' MNIST dataset is that it is a little too easy to classify correctly. Fashion-MNIST is meant as a drop-in replacement, a benchmark for ML algorithms to use instead of MNIST.



```
from itertools import product
import numpy as np

# The loader lives in fashion_mnist.py (in the folder of these notes) and is also used in
# the chapters Expectation -Maximization and Model selection. The data are downloaded on the
```

```
# first call only and after that read from the working directory.
from fashion_mnist import load_mnist

x_train, y_train = load_mnist(kind='train')
x_test, y_test = load_mnist(kind='t10k')
```

We have to reduce the dimensionality, or training takes too long:

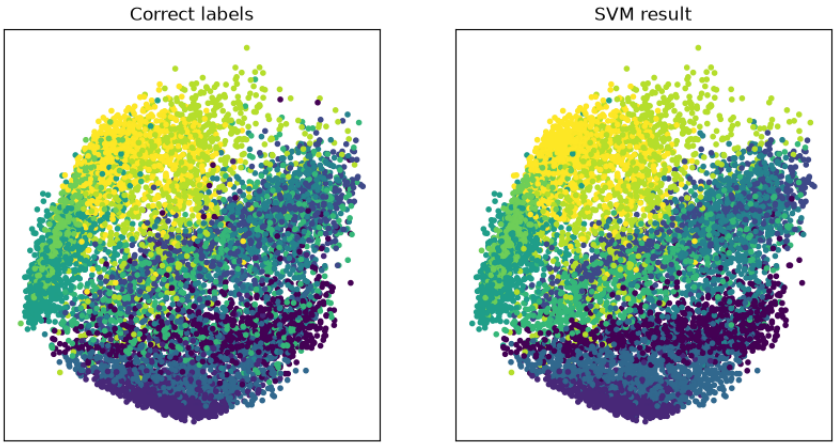
```
from sklearn.decomposition import PCA
pca = PCA(40).fit(x_train)
x_train_pc = pca.transform(x_train)
x_test_pc = pca.transform(x_test)
```

First, a look at the result of a support vector classification (SVC – a method we have not discussed and simply use as a black box here) without any hyperparameter fine-tuning:

```
from sklearn.svm import SVC

# trim doum some more dimensions to speed this up:
pca2 = PCA(6).fit(x_train_pc)
x_train_pc2 = pca2.transform(x_train_pc)
x_test_pc2 = pca2.transform(x_test_pc)
svc = SVC().fit(x_train_pc2, y_train)
y_pred = svc.predict(x_test_pc2)

from sklearn.metrics import normalized_mutual_info_score
print(classification_report(y_test, y_pred))
print("NMI = %.2f" % normalized_mutual_info_score(y_test, y_pred))
```

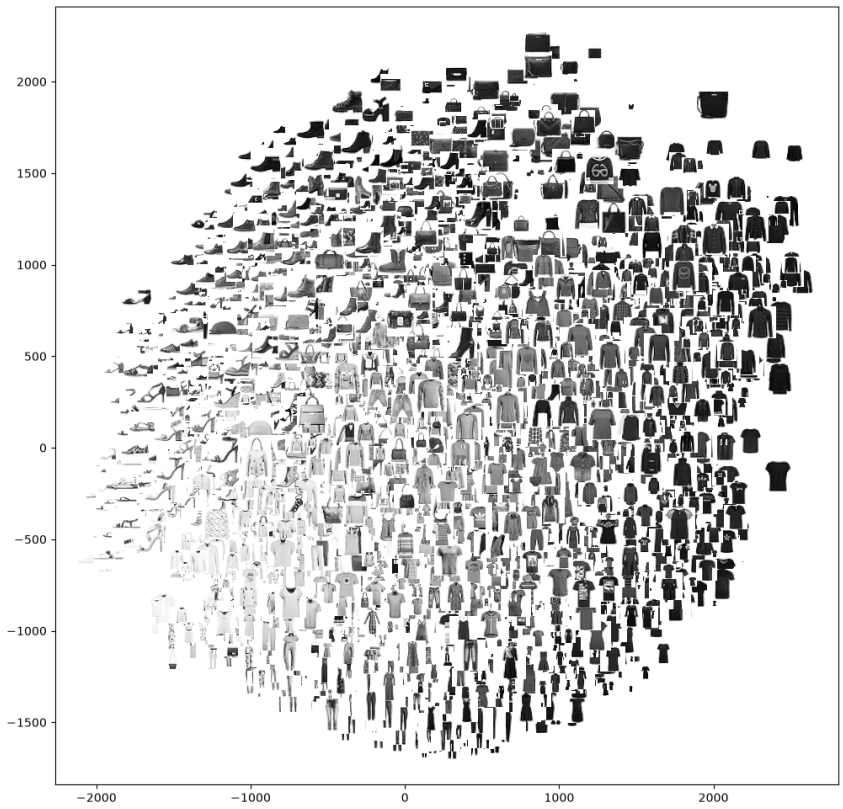
```
precision    recall  f1 -score   support

           0       0.74      0.79      0.76      1000
           1       0.94      0.85      0.90      1000
           2       0.65      0.55      0.60      1000
           3       0.69      0.81      0.75      1000
           4       0.62      0.65      0.63      1000
           5       0.86      0.84      0.85      1000
           6       0.43      0.39      0.41      1000
           7       0.85      0.84      0.84      1000
           8       0.93      0.94      0.93      1000
           9       0.86      0.90      0.88      1000

    accuracy                           0.76     10000
   macro avg       0.76      0.76      0.76     10000
weighted avg       0.76      0.76      0.76     10000

NMI = 0.67
```

```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].scatter(x_test_pc2[:,0], x_test_pc2[:,1], c=y_test, marker='.')
axs[0].set_title("Correct labels")
axs[1].scatter(x_test_pc2[:,0], x_test_pc2[:,1], c=y_pred, marker='.')
axs[1].set_title("SVM result")
for ax in axs:
    ax.set_xticks([])
    ax.set_yticks([])
plt.show()
```



We can also visualise what the individual data points stand for:

```
from matplotlib.offsetbox import OffsetImage, AnnotationBbox

def imscatter(x, y, image, ax, zoom=1):
    im = OffsetImage(image, zoom=zoom, norm=plt.Normalize(0,255), cmap="Greys")
    x, y = np.atleast_1d(x, y)
    artists = []
    for x0, y0 in zip(x, y):
        ab = AnnotationBbox(im, (x0, y0), xycoords='data', frameon=False)
        artists.append(ax.add_artist(ab))
    ax.update_datalim(np.column_stack([x, y]))
    ax.autoscale()
    return artists

fig, ax = plt.subplots(figsize=(12,12))
# plot every 2nd:
for i in range(0, len(x_test_pc2), 2):
    imscatter(x_test_pc2[i,0], x_test_pc2[i,1], x_test[i].reshape(28,28), ax, zoom=np.random.
    rand())
plt.show()
```



Now we look for good values of the hyperparameters C, γ of the SVC. We go about it exactly as in the randomised search above:

- A subsample is enough for the search; training on all 60,000 images takes too long to be repeated dozens of times.
- C and γ can be sensible across many orders of magnitude, so we sample both log-uniformly.
- We scale the principal components to unit variance beforehand. Otherwise γ has to make up for the enormous scale of the pixel sums and ends up at values around 10^{-8}, which are hard to interpret. We put the scaling and the classifier into a `Pipeline`, so that in every fold of the cross-validation the scaling is estimated from that fold's training data alone.

```
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from scipy.stats import loguniform

dims = 12
small_x, _, small_y, _ = train_test_split(x_train_pc[:, :dims], y_train,
                                          train_size=2000, random_state=23)

pipe = make_pipeline(StandardScaler(), SVC())
param_dist = {
    "svc__C": loguniform(1e -1, 1e3),
    "svc__gamma": loguniform(1e -3, 1e0),
}
random_search = RandomizedSearchCV(pipe, param_distributions=param_dist,
                                   n_iter=40, cv=3, scoring='f1_macro',
                                   random_state=23, n_jobs= -1)
random_search.fit(small_x, small_y)
print("best parameters:", random_search.best_params_)
print("f1_macro (cross -validation on the subsample) =", random_search.best_score_)

best parameters: {'svc__C': np.float64(29.662611861688664), 'svc__gamma': np.float64
    (0.017210373067083652)}
```

```
f1_macro (cross -validation on the subsample) = 0.7899375300764434
```

How sensitive is the result to the parameters? A look at the best and the worst candidates:

```
import pandas as pd

results = pd.DataFrame(random_search.cv_results_)
results = results[["param_svc__C", "param_svc__gamma", "mean_test_score"]]
results = results.sort_values("mean_test_score", ascending=False)
print(results.head(5))
print(results.tail(3))
```

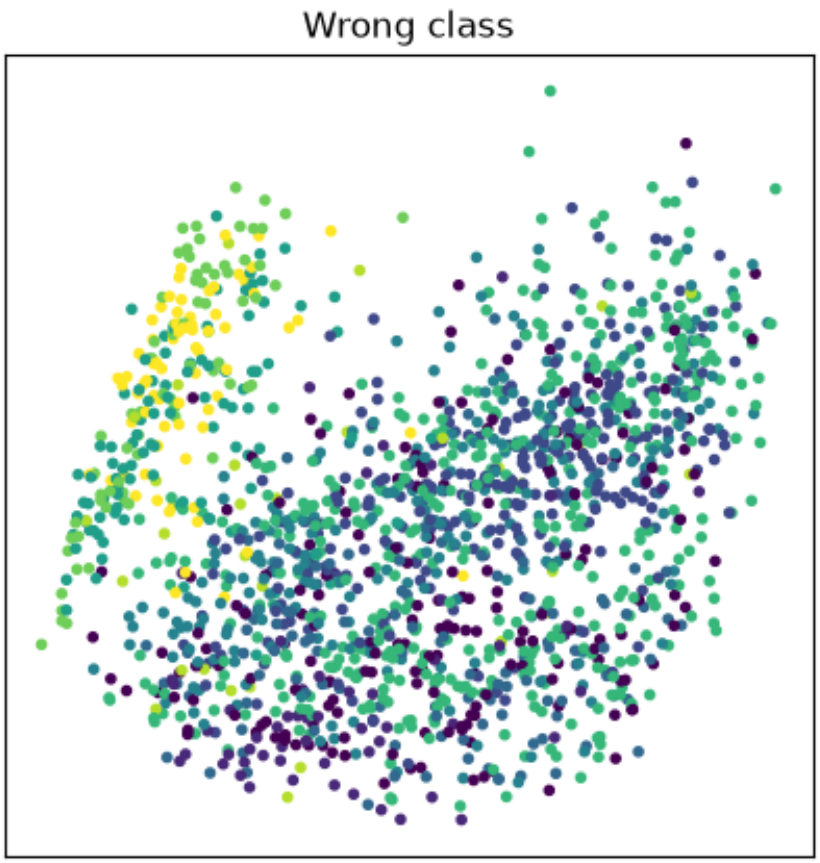
```
param_svc__C  param_svc__gamma  mean_test_score
4      29.662612          0.017210         0.789938
1     115.302583          0.007034         0.787693
31    463.053198          0.005063         0.785714
12    587.036265          0.002658         0.784648
30    469.659620          0.002439         0.784420
    param_svc__C  param_svc__gamma  mean_test_score
5       0.102296          0.448845         0.539789
38      0.215727          0.001445         0.308637
11      0.276637          0.001004         0.242741
```

With the parameters found, we now train on all the samples. The pipeline re-estimates the scaling as it does so, this time from all the training data:

```
from sklearn.metrics import f1_score

best = random_search.best_estimator_     # pipeline with the best parameters
best.fit(x_train_pc[:, :dims], y_train)
y_pred = best.predict(x_test_pc[:, :dims])
print(f1_score(y_test, y_pred, average="macro"))
```

```
0.8362604024819902
```

```
y_wrong_indices = (y_test != y_pred)
y_wrong = y_test[y_wrong_indices]
x_wrong = x_test_pc2[y_wrong_indices]

fig, ax = plt.subplots(figsize=(5, 5))
ax.scatter(x_wrong[:,0], x_wrong[:,1], c=y_wrong, marker='.')
ax.set_title("Wrong class")
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```



Can you manage a better classification (with an F1 score > 0.85)? Some ideas: more principal components, a larger subsample for the search, a different kernel, or an altogether different classifier.

7.8 Expectation maximization: Gaussian mixture models from scratch

7.8.1 Motivation

In supervised learning we saw that linear regression (possibly with a kernel) might help us classify data. In unsupervised learning we used k-means and kernel k-means for classification, each supplemented by scaling the data and a principal component analysis. The results were good whenever the clusters in feature space were convex and well approximated by a normal distribution without covariance (that is, by solid balls). But when the clusters were normally distributed with differing covariances and could not be separated linearly from one another, k-means gave no good results.

Example
Formulated as an expectation-maximization procedure, the k-means algorithm reads:

1. Initialisation (fix the cluster centres)
2. Iterate until the cluster centres converge:
   a. (E-step): assign every point to the cluster centre nearest to it and so arrive at a division into classes
   b. (M-step): set the new cluster centres to the mean of all elements of a class

The E-step (expectation) is so called because we update our expectation about the value of the latent variable.

The M-step (maximization) is so called because we maximise a fitness function – here, we minimise the sum of the squared distances from the points to the cluster centres they have been assigned to.

It is natural to want a procedure that takes the shape of the clusters into account.

Gaussian mixture models, supervised If, for data in $\mathbb{R}^d$, we are given a division into k classes (labels), then for the subset of the data consisting of all points of one class we can compute the sample covariance and the sample expected value, and from them set up a d-variate normal distribution that models this subset.

This yields a model for the data as a whole: write p_i for the number of points in class i divided by the total number of points in the sample, and $f_i(x)$ for the estimated density of the normal distribution of class i; then the density is

$$f(x) = \sum_{i=1}^{k} p_i f_i(x). \tag{223}$$

This model, in turn, we can of course use to judge which class a point belongs to: for a point x we compute the densities

$$P_i(x) = p_i f_i(x) \tag{224}$$

and pick the i for which this expression is largest.

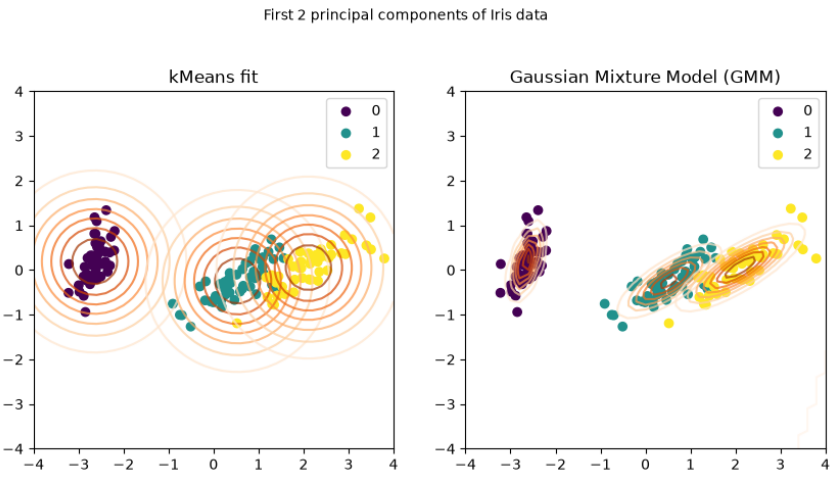
One can picture the generative process like this: the data points were sampled from a (fictitious) Gaussian mixture model by first picking a class j according to the discrete distribution of the p_i and then drawing a sample from the density $f_j(x)$ – once for each data point.

Now if the clusters overlap, it can easily happen that the model classifies some points x differently from their original classes – and in principle there is no remedy for that. Even so, in this way we can find the Gaussian mixture model that best fits the data, and use it to sample data that will look very much like the data at hand (provided the normality assumption was right).

One further observation: the Gaussian mixture model gives us more than just a new division into classes; for every point and every class, it gives us a probability that the point belongs to that class. That is a good deal more information, and it can be considerably more useful. For the points that are not correctly classified, above all, it gives us a way of quantifying how certain we are about the division into classes.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from scipy.stats import multivariate_normal as mvn

x, y = load_iris(return_X_y=True)
# work in principal components, to plot later in the first two:
xpc = PCA().fit_transform(x)

k = 3
assert k == np.max(y) + 1

mvns = list(range(k))
for i in range(k):
    # for simplicty, take only first 2 PCs here:
    mu = np.mean(xpc[y == i][:,:2], axis=0)
    sigma = np.cov(xpc[y == i][:,:2], rowvar=False)
    mvns[i] = mvn(mean=mu, cov=sigma, allow_singular=True)

round_mvns = list(range(k))
for i in range(k):
    mu = np.mean(xpc[y == i][:,:2], axis=0)
    sigma = np.eye(2)
    round_mvns[i] = mvn(mean=mu, cov=sigma, allow_singular=True)

fig = plt.gcf()
fig.set_size_inches(10, 10)
ax1 = plt.subplot(221)
ax2 = plt.subplot(222, sharex=ax1, sharey=ax1)
plt.figtext(0.35, 0.95, "First 2 principal components of Iris data")
ax1.set_title("kMeans fit")
ax2.set_title("Gaussian Mixture Model (GMM)")
xs, ys = np.mgrid[ -4:4.1:.2, -4:4.1:.1]
for i in range(k):
    zs = round_mvns[i].pdf(np.dstack((xs, ys)))
    ax1.contour(xs, ys, zs,
                cmap=plt.cm.Oranges, alpha=.7)
    ax1.scatter(xpc[y == i][:,0], xpc[y == i][:,1],
                c=np.ones_like(y[y == i])*i, vmin=0, vmax=k -1,
                label=str(i))
    zs = mvns[i].pdf(np.dstack((xs, ys)))
    ax2.contour(xs, ys, zs,
                cmap=plt.cm.Oranges, alpha=.7)
    ax2.scatter(xpc[y == i][:,0], xpc[y == i][:,1],
                c=np.ones_like(y[y == i])*i, vmin=0, vmax=k -1,
                label=str(i))
ax1.legend()
ax2.legend()
plt.show()
```



Gaussian mixture models, unsupervised The procedure just described is not available if no division into classes is given to us. In that case the class is a **latent variable**: one we do not know, but which is essential to describing the model. What we can model well are the conditional probabilities given the value of the latent variable – that is, the distribution of a single class. The distribution of the latent variable itself is harder to describe when all we have is unlabelled data.

A two-stage procedure suggests itself: first find a division into classes with k-means (often not a particularly good one, if the homoscedasticity assumption fails) and then, on the basis of that division, set up a Gaussian mixture model – that is, obtain an estimate of the latent variable.

Now, in turn, the division into classes that the mixture model lets us compute will in general differ from the one we computed with k-means beforehand. This time, though, it is plausible that the mixture model achieves the better division, since now the shapes of the clusters are taken into account as well (approximated by ellipsoids, thanks to the normality assumption).

With this improved division into classes – that is, with the improved estimate of the latent variable – we can set up a Gaussian mixture model all over again, re-estimating the covariance matrices and the centres, much as in the iteration step of the k-means algorithm.

One more observation: if, in setting up the covariance matrices, we consider only the quality of the model for a single class (the best fit for that class alone), then we ignore the effect that change of model has on the other classes – and there usually is one. It would therefore be better to optimise all the parameters simultaneously in the iteration step, towards the target quantity – the mixture model with the best fit.

To construct a good estimator for an arbitrary distribution we have used the maximum likelihood method, and that is what we want to try for a mixture of normal distributions too.

7.8.2 Maximum likelihood in models with latent variables

The probability density in the Gaussian mixture model, where θ_j denotes the parameters μ_j, Σ_j of a single mixture component and p_j the distribution of the components, is

$$f(x) = \sum_{j=1}^{k} p_j f_j(x, \theta_j) = P(X = x | M = \theta) = L(\theta), \tag{225}$$

and that is the likelihood (the probability of the data, computed in our model), writing M for our model and X for the data. So the model assumption is now: X is distributed according to f. Here θ is the collection of the parameters p_j together with the parameters μ_j, Σ_j hidden inside the f_j. Reading this quantity as a likelihood means reading it as a function of those parameters.

A tip: if you prefer to picture θ as a list indexed by θ_j, simply put $\theta_0 := (p_1, \ldots, p_k)$.

If we take seriously the modelling decision to express the division of the data points into classes by probabilities, we may as well model the division itself as a random variable Y: $Y_i = j$ if the i-th data point is of class j – only we do not know the variable Y.

Our model, formulated with the latent variable Y, is

$$
\begin{aligned}
P(Y = j \mid \theta) =& p_j \\
P(X = x \mid Y = j, \theta) =& f_j(x,\ \mu_j, \Sigma_j) \\
P(X = x \mid \theta) =& \sum_{j=1}^{k} P(X = x,\ Y = j \mid \theta) \\
=& \sum_{j=1}^{k} P(Y = j \mid \theta)\, P(X = x \mid Y = j, \theta).
\end{aligned}
\tag{226}
$$

Finding the zeros of the derivative? Finding the zeros of the derivative of $L(\theta)$ directly is not easy, and the popular trick of maximising $\ell(\theta) = \log L(\theta)$ instead does not help much here either, because

$$
\begin{aligned}
\ell(\theta) =& \log P(X = x \mid \theta) \\
=& \log \left(\sum_{j=1}^{k} P(X = x, Y = j \mid \theta) \right) \\
=& \log \left(\sum_{j=1}^{k} p_j f_j(x \mid \theta_j) \right)
\end{aligned}
\tag{227}
$$

and the logarithm does not get us out of that sum business so easily.

Jensen to the rescue! We can lower our sights a little and work with Jensen's inequality (the logarithm is concave and $\sum_{j=1}^{k} p_j = 1$):

$$
\ell(\theta) \geq \sum_{j=1}^{k} p_j \log f_j(x \mid \theta_j) \tag{228}
$$

The charm of the expression on the right-hand side is that a θ which maximises the right-hand side cannot have a smaller left-hand side. In formulas: if we are given a $\theta^{(t)}$ and find a $\theta^{(t+1)}$ for which the right-hand side grows, then $\ell(\theta^{(t+1)}) \geq \ell(\theta^{(t)})$ as well.

Note that a fat $1 = \frac{q(j)}{q(j)}$ can also be multiplied into the expression for the contribution of a class, $p_j f_j$ – thus

$$
\begin{aligned}
\ell(\theta) =& \log \left(\sum_{j=1}^{k} P(X = x, Y = j \mid \theta) \right) \\
=& \log \left(\sum_{j=1}^{k} q(j) \frac{P(X = x, Y = j \mid \theta)}{q(j)} \right).
\end{aligned}
\tag{229}
$$

The point of this becomes visible once Jensen's inequality is applied, this time to the convex combination $\sum_{j=1}^{k} q(j) = 1$ – so from now on we assume that q is a probability density:

$$
\begin{aligned}
\ell(\theta) \geq& \sum_{j=1}^{k} q(j) \log \left(\frac{P(X = x, Y = j \mid \theta)}{q(j)} \right) =: L(\theta, q) \\
=& \sum_{j=1}^{k} q(j) \left(\log \left(P(X = x, Y = j \mid \theta) - \log q(j) \right) \right. \\
=& \sum_{j=1}^{k} q(j) \log P\left(X = x, Y = j \mid \theta \right) \qquad - \sum_{j=1}^{k} q(j) \log q(j) \\
=& \sum_{j=1}^{k} q(j) \log P\left(X = x, Y = j \mid \theta \right) \qquad + H(q).
\end{aligned}
\tag{230}
$$

So we have found a lower bound on $\ell(\theta)$ that we can write as the sum of an expression which is easier to compute and the entropy of the distribution with density q. Since the entropy does not depend on θ, $L(\theta, q)$ can be maximised by maximising the other summand.

Remark
In the course *Introduction to Deep Learning* this very lower bound $L(\theta, q)$ is called the *ELBO* (evidence lower bound; the *evidence* is $P(X = x \mid \theta)$). There it is the training objective of *variational autoencoders*, in which the distribution q of the latent variable is output by a neural network.

We shall come back to this observation (which is why we have given it a name, $L(\theta, q)$); for the moment, let us discuss a general approach to the solution.

7.8.3 Auxiliary Functions

If we cannot maximise a function $\ell(\theta)$, we can try a stopgap: pretend that θ is constant in some of the subexpressions of $\ell(\theta)$ and maximise, then hold the other subexpressions constant instead and maximise over the rest. To formalise this we introduce an auxiliary function $Q(\theta, \vartheta)$ that is easier to maximise in θ than ℓ is, but nevertheless satisfies $Q(\theta, \theta) = \ell(\theta)$ and, moreover, $Q(\theta, \vartheta) \leq \ell(\theta)$ for all ϑ.

These assumptions are already enough to show that an iterative process leads to better and better approximations $\theta^{(t)}$ to the θ that maximises $\ell(\theta)$. Here $\theta^{(t+1)}$ denotes the θ that makes $Q(\theta, \theta^{(t)})$ maximal (which by assumption is easy to find).

$$\begin{aligned}\ell(\theta^{(t)}) =& Q(\theta^{(t)}, \theta^{(t)}) \\ \leq& Q(\theta^{(t+1)}, \theta^{(t)}) \text{ (M-step)} \\ \leq& Q(\theta^{(t+1)}, \theta^{(t+1)}) \text{ (E-step)} \\ =& \ell(\theta^{(t+1)}).\end{aligned} \tag{231}$$

All told, $\ell(\theta^{(t)}) \leq \ell(\theta^{(t+1)})$ – which is to say that at least our estimate never gets worse along the way!

If we want to take this approach, what remains is to choose a well-judged auxiliary function $Q(\theta, \vartheta)$.

7.8.4 The complete-data log-likelihood

The last ingredient for developing the EM algorithm for the Gaussian mixture model is the complete-data log-likelihood: the log-likelihood in the fictitious world in which we really do know the complete data – not only the values of X but those of the latent variable Y as well, that is, the actual classes of the data points.

The complete data are therefore the (x_i, y_i), which we model with (X, Y).

It is useful now to look at the variable Y_i in one-hot encoding as well: $Y_{ij} = 1$ if $Y_i = j$, and $Y_{ij} = 0$ if $Y_i \neq j$. Some people write $[Y_i = j]$ for this (the Iverson bracket).

The log-likelihood of the data under the Gaussian mixture model is

$$\begin{aligned}\ell_c(\theta) = \log P((X, Y)|\theta) =& \sum_{i=1}^{n} \log P(X = x_i, Y = y_i \mid \theta) \\ & \sum_{i=1}^{n} \log \prod_{j=1}^{k} P(X = x_i, Y = j \mid \theta)^{Y_{ij}} \\ & \sum_{i=1}^{n} \sum_{j=1}^{k} Y_{ij} \log P(X = x_i, Y = j \mid \theta)\end{aligned} \tag{232}$$

where in the first step we inserted a fat 1: Y_{ij} is 1 for just one factor of the product $\prod_{j=1}^{k}(...)^{Y_{ij}}$ and 0 otherwise, which makes the corresponding factor 1. The second step is just $\log \prod (...)^t = \sum t \log(...)$.

Note also that θ is now really only $\theta_1, \ldots, \theta_k$, since the p_i are encoded in the distribution of Y.

We can simplify further:

$$\log P(X = x_i, Y = y_i) = \log \left(p_{y_i} f_{y_i}(x_i)\right) = \log p_{y_i} + \log f_{y_i}(x_i) \tag{233}$$

The expected value over the latent variable If we assume a distribution $q(y)$ for the latent variable Y, then we can take the expected value over it:

$$\begin{aligned}\mathbb{E}_{Y\sim q}\ell_c(\theta) =& \mathbb{E}_{Y\sim q}\left(\sum_{i=1}^{n}\sum_{j=1}^{k} Y_{ij}\left(\log p_{y_i} + \log f_{y_i}(x_i \mid \theta_{y_i})\right)\right)\\ =& \sum_{i=1}^{n}\sum_{j=1}^{k} \mathbb{E}_{Y\sim q} Y_{ij}\left(\log p_j + \log f_j(x_i \mid \theta_j)\right)\\ =& \sum_{i=1}^{n}\sum_{j=1}^{k} \tau_{ij}\left(\log p_j + \log f_j(x_i \mid \theta_j)\right).\end{aligned} \tag{234}$$

The terms denoted τ_{ij} we call the **responsibilities**, since in our model they are responsible for the division of the points x_i into classes:

$$\tau_{ij} := \mathbb{E}_{Y\sim q} Y_{ij} = P(Y_i = j \mid X = x_i, \theta) = \frac{P(Y_i = j,\ X = x_i \mid \theta)}{P(X = x_i \mid \theta)} \tag{235}$$

If we hold the τ_{ij} fixed, then $\mathbb{E}_{Y\sim q}\ell_c(\theta)$ is a quantity we can maximise (even though the τ_{ij} do of course really depend on θ). We still want to regard the τ_{ij} as a function of θ, but from now on we call that argument ϑ.

So we define

$$Q(\theta, \vartheta) := \sum_{i=1}^{n}\sum_{j=1}^{k} \tau_{ij}(\vartheta)\left(\log p_j + \log f_j(x_i \mid \theta_j)\right) \tag{236}$$

For $\theta = \vartheta$ this is the expected complete-data log-likelihood under our assumption q about the distribution of the latent variable Y. Note, too, that the other term, $\log P(X = x_i, Y = j) = (\log p_j + \log f_j(x_i \mid \theta_j))$, depends on θ, not on ϑ.

The difference between L(θ, q) and Q(θ, θ) Let us recall the observation we reached with the help of Jensen's inequality:

$$\ell(\theta) \geq L(\theta, q) = \sum_{j=1}^{k} q(j)\log P\left(X = x, Y = j \mid \theta\right) \qquad + H(q) \tag{237}$$

and note that the summand other than $H(q)$ is exactly the expected complete-data log-likelihood, that is

$$\ell(\theta) \geq L(\theta, q) = \mathbb{E}_{Y\sim q}\ell_c(\theta) + H(q) = Q(\theta, \theta) + H(q) \tag{238}$$

Since $H(q)$ does not depend on θ, maximising $Q(\theta, \vartheta)$ in θ therefore also maximises $L(\theta, q)$ in θ.

The difference between (θ) and L(θ, q) Let us rearrange the formula for $L(\theta, q)$ once more, in a different way: we express the joint distribution $P(X = x, Y = j)$ through conditional distributions, $P(Y = j, X = x) = P(Y = j|X = x)P(X = x)$, and obtain:

$$\begin{aligned}\ell(\theta) \geq& L(\theta, q) = \sum_{j=1}^{k} q(j)\log\left(\frac{P(X = x, Y = j \mid \theta)}{q(j)}\right)\\ =& \sum_{j=1}^{k} q(j)\log\left(\frac{P(Y = j \mid X = x, \theta)P(X = x \mid \theta)}{q(j)}\right)\\ =& \sum_{j=1}^{k} q(j)\log\left(\frac{P(Y = j \mid X = x, \theta)}{q(j)}\right) \qquad + \sum_{j=1}^{k} q(j)\log\left(P(X = x \mid \theta)\right)\\ =& -H\big(q \,\|\, P(Y \mid X = x, \theta)\big) + \ell(\theta).\end{aligned} \tag{239}$$

The difference between $\ell(\theta)$ and $L(\theta, q)$ is therefore exactly the relative entropy of the estimated distribution q to the actual distribution $P(Y \mid X = x, \theta)$ (in the notation of the chapter Relative entropy):

$$\ell(\theta) - L(\theta, q) = H\big(q \,\|\, P(Y \mid X = x, \theta)\big) \tag{240}$$

With that we also have $\ell(\theta) - Q(\theta, \theta) = H\big(q \,\|\, P(Y \mid X = x, \theta)\big) + H(q)$, so we now understand exactly in what sense $\ell(\theta)$ can be maximised by way of $Q(\theta, \theta)$.

7.8.5 The EM algorithm for fitting a Gaussian mixture model

The algorithm is initialised with a choice of $\theta^{(0)}$ (and, as we have already seen with k-means, the result generally depends very much on this initialisation). One possible choice for the μ_j (the cluster centres) is to run the k-means algorithm (with k-means++ initialisation) on the data and take the converged centroids as the μ_j. For the covariance matrices one can start with identity matrices (which corresponds to a shifted standard normal distribution). The mixing ratio, that is the p_i, can be initialised to $p_i = \frac{1}{k}$.

Then the E-step and the M-step are carried out iteratively, alternating between the two:

- **E-step**: update the parameter ϑ: set $\vartheta = \theta$ and compute the responsibilities $\tau_{ij} := \tau_{ij}(\theta)$, so as to have a correct formula for $Q(\theta, \theta)$ again – or rather a new estimate of q, and with it of $L(\theta, q)$. In this step we estimate the latent variable, aiming to minimise the relative entropy to its actual distribution (that is, to approximate it as well as possible). The better the estimate, the closer $L(\theta, q)$ lies to $\ell(\theta)$.
- **M-step**: update the parameter θ: maximise the expected complete log-likelihood $Q(\theta, \vartheta)$ in θ (with ϑ held fixed, so that the responsibilities τ_{ij} are constant). This maximises $L(\theta, q)$, a lower bound on $\ell(\theta)$.

The procedure converges, since the approximations to a maximum of $\ell(\theta)$ never get worse (because of Jensen's inequality, one might say).

So far we have not needed to know what form the density functions f_j take – in fact everything discussed up to this point works for arbitrary distributions as well! With some care, the latent variable may even be continuous, as long as the sums from $j = 1$ to k are suitably replaced by the corresponding integrals. Seen that way, the EM algorithm is really not one algorithm but a template for developing many concrete EM algorithms.

More generally still, when the aim is not necessarily to maximise a likelihood in the presence of latent variables, one speaks of MM algorithms – for minorise–maximise (or, for minimisation problems, majorise–minimise) – and of optimisation transfer; a good introduction is Hunter & Lange, *A Tutorial on MM Algorithms*, The American Statistician 58 (2004). With these keywords you can track down the literature should you need a generalisation of the EM algorithm.

For the moment, though, we restrict ourselves to a discrete, finite latent variable Y and head towards an implementation in Python.

Concrete formulas for the Gaussian mixture model To implement the algorithm concretely we must of course use concrete density functions too, and that is what we do now – so let us recall that the multivariate normal distribution has this density:

$$f_j(x) = f_j(x, \mu_j, \Sigma_j) = \frac{1}{\sqrt{(2\pi)^n \det \Sigma_j}} e^{-\frac{(x-\mu_j)^T \Sigma_j^{-1} (x-\mu_j)}{2}}. \tag{241}$$

To make life easy for ourselves, we look at the univariate case first, so that

$$f_j(x) = f_j(x, \mu_j, \sigma_j) = \frac{1}{\sqrt{2\pi}\,\sigma_j}\, e^{-\frac{(x-\mu_j)^2}{2\sigma_j^2}}. \tag{242}$$

At step t of the iteration we are given the parameters $\theta^{(t)}$, that is, $\mu_j^{(t)}$ and $\Sigma_j^{(t)}$ together with the mixture parameters $p_j^{(t)}$. The superscript index t counts the iterations; i remains the sample index, as in x_i.

In the E-step we compute the responsibilities

$$\begin{aligned}\tau_{ij}(\theta^{(t)}) =& \frac{P(Y_i = j, X = x_i \mid \theta)}{P(X = x_i \mid \theta)} \\ =& \frac{P(Y_i = j, X = x_i \mid \theta)}{\sum_{m=1}^{k} P(Y_i = m, X = x_i \mid \theta)} \\ =& \frac{p_j^{(t)} f_j\left(x_i, \mu_j^{(t)}, \sigma_j^{(t)}\right)}{\sum_{m=1}^{k} p_m^{(t)} f_m\left(x_i, \mu_m^{(t)}, \sigma_m^{(t)}\right)}.\end{aligned} \tag{243}$$

In the M-step we maximise $Q(\theta, \vartheta)$ in θ by finding the zeros of the derivative with respect to θ, that is, we solve the system of equations

$$\begin{aligned}0 =& \frac{d}{d\theta} Q(\theta, \vartheta) \\ =& \frac{d}{d\theta} \sum_{i=1}^{n} \sum_{j=1}^{k} \tau_{ij}(\vartheta) \left(\log p_j + \log f_j(x_i \mid \theta_j)\right) \\ =& \sum_{i=1}^{n} \sum_{j=1}^{k} \tau_{ij}(\vartheta) \frac{d}{d\theta} \left(\log p_j + \log f_j(x_i \mid \theta_j)\right).\end{aligned} \tag{244}$$

First we look at the derivatives with respect to the μ_j:

$$\begin{aligned}\frac{d}{d\mu_j} \log f_j(x_i) =& -\frac{\frac{d}{d\mu_j}(x_i - \mu_j)^2}{2\sigma_j^2} = \frac{x_i - \mu_j}{\sigma_j^2} \\ \frac{d}{d\mu_j} Q(\theta, \theta^{(t)}) =& \sum_{i=1}^{n} \sum_{j=1}^{k} \tau_{ij}(\vartheta) \frac{d}{d\mu_j} \log f_j(x_i \mid \theta_j) \\ =& \sum_{i=1}^{n} \tau_{ij}(\vartheta) \frac{x_i - \mu_j}{\sigma_j^2}.\end{aligned} \tag{245}$$

Multiplying through by σ_j^2 and solving for μ_j gives the closed formula

$$\mu_j^{(t+1)} = \frac{\sum_{i=1}^{n} \tau_{ij} x_i}{\sum_{i=1}^{n} \tau_{ij}}. \tag{246}$$

Next we look at the derivatives with respect to the σ_j:

$$\begin{aligned}\frac{d}{d\sigma_j} \log f_j(x_i) =& -\frac{1}{\sigma_j} + \frac{(x_i - \mu_j)^2}{\sigma_j^3} \\ \frac{d}{d\sigma_j} Q(\theta, \theta^{(t)}) =& \sum_{i=1}^{n} \tau_{ij} \left(\frac{(x_i - \mu_j)^2}{\sigma_j^3} - \frac{1}{\sigma_j}\right).\end{aligned} \tag{247}$$

Solving the equation $0 = \frac{d}{d\sigma_j} Q(\theta, \theta^{(t)})$ for σ_j (multiplying through by σ_j^3) gives

$$\left(\sigma_j^{(t+1)}\right)^2 = \frac{\sum_{i=1}^{n} \tau_{ij} (x_i - \mu_j)^2}{\sum_{i=1}^{n} \tau_{ij}} \tag{248}$$

Finally, in the same way, the zeros of the derivative with respect to p_j give the formula that represents p_j as the responsibility of x_i for class j, averaged over the x_i:

$$p_j^{(t+1)} = \frac{\sum_{i=1}^{n} \tau_{ij}}{n} \tag{249}$$

With that, the new parameters $\theta^{(t+1)}$ have been found.

Special case: k-means If we hold $p_j = \frac{1}{k}$ and $\Sigma_j = E_d$ fixed, we obtain the formulas

$$\tau_{ij}(\theta^{(t)}) = \frac{f_j\left(x_i, \mu_j^{(t)}, E_d\right)}{\sum_{m=1}^{k} f_m\left(x_i, \mu_m^{(t)}, E_d\right)} \tag{250}$$

A real difference from the classical k-means algorithm is that here we have worked with **soft assignments**: for every possible class of a point x_i we have determined a probability τ_{ij}. Classical k-means, by contrast, commits itself – it makes a **hard assignment**. We can just as well set $\widetilde{\tau_{ij}} := 1$ when τ_{ij} is the largest of all the τ_{im}, and $\widetilde{\tau_{ij}} := 0$ otherwise. With that we get

$$\begin{aligned} \mu_j^{(t+1)} =& \frac{\sum_{i=1}^{n} \widetilde{\tau_{ij}} x_i}{\sum_{i=1}^{n} \widetilde{\tau_{ij}}} \\ =& \frac{1}{n_j} \sum_{i:\ y_i = j} x_i, \qquad n_j := \sum_{i=1}^{n} \widetilde{\tau_{ij}} = \big|\{i : y_i = j\}\big|, \end{aligned} \tag{251}$$

where y_i is the class to which x_i was hard-assigned. That is exactly the arithmetic mean of the n_j points x_i in class j. With the soft assignment instead, it is a weighted mean of the x_i that gives less weight to the points whose assignment is less certain – another obvious improvement on classical k-means.

Implementing more general models If you want to implement models more general than the ones discussed here, you have to adapt the part in which the explicit closed formulas for p_j, μ_j, σ_j are computed. We were able to find closed formulas because the log of a density that can be written as $e^{(\cdots)}$ is especially simple. As a rule of thumb, then, every distribution of that same build can be treated the same way and used in place of the normal distribution. The precise statement is that every distribution from a so-called exponential family is suitable.

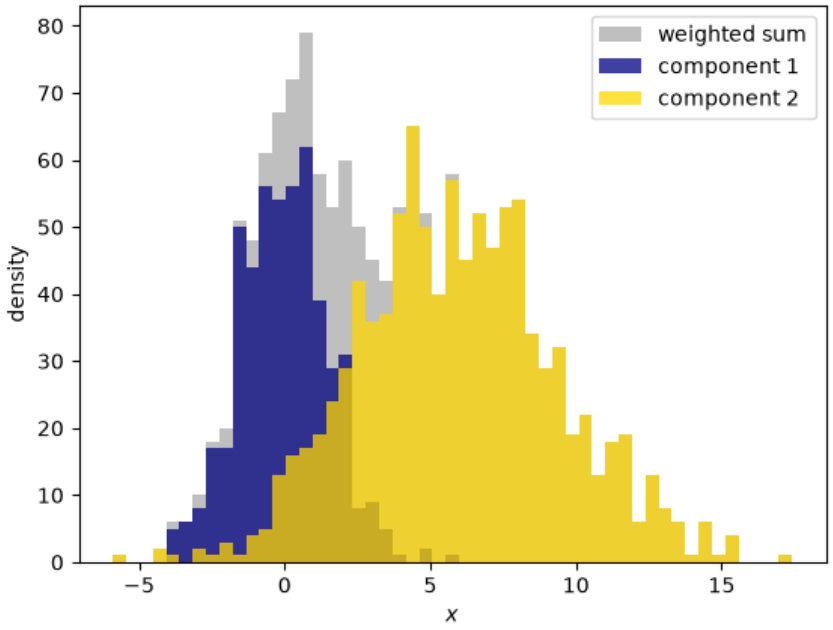
Implementation in Python

```
# version for univariate Gaussian mixtures,
# not vectorised (for the sake of understanding)

from scipy.stats import norm

def EM(xs, pis, mus, sigmas):
    K = len(pis)
    N = xs.shape[0]

    #(E) step
    taus = np.zeros((N,K))
    for i in range(N):
        reg = 0
        for j in range(K):
            taus[i,j] = pis[j] * norm.pdf(xs[i],
                                          loc=mus[j],
                                          scale=sigmas[j])
            reg += taus[i,j]
        for j in range(K):
            taus[i,j] /= reg

    #(M) step
    Npis = np.zeros(K)
    for j in range(K):
        Npis[j] = taus[:,j].sum()
    pis = Npis / N

    mus = np.zeros(K)
    for j in range(K):
        for i in range(N):
            mus[j] += taus[i, j] * xs[i]
```

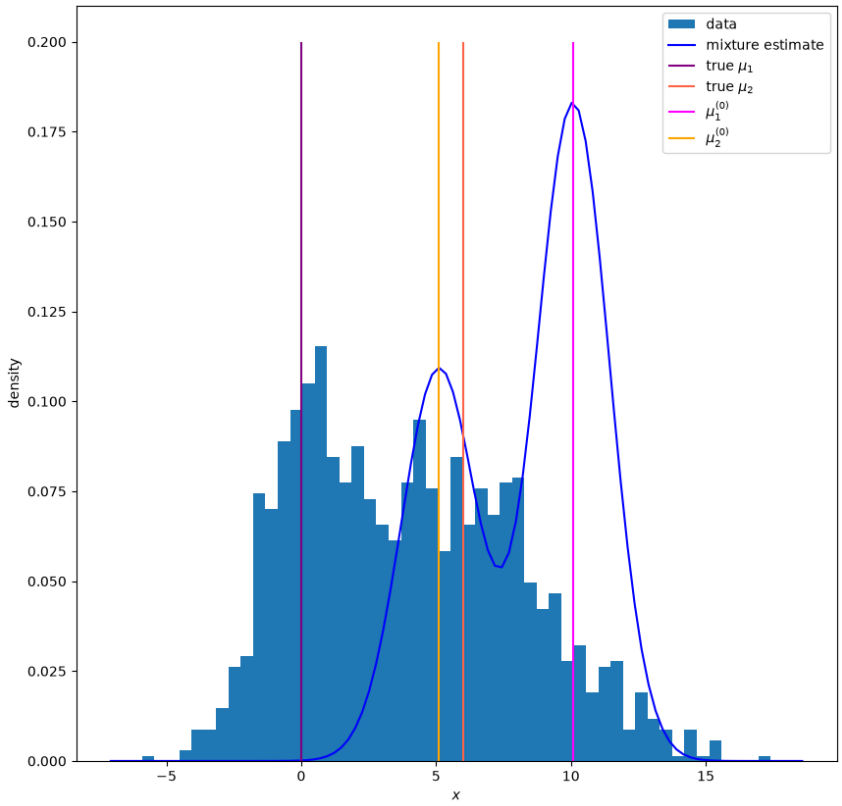
```
        mus[j] /= Npis[j]

    sigmas = np.zeros(K)
    for j in range(K):
        for i in range(N):
            sigmas[j] += taus[i, j] * ((xs[i] - mus[j])**2)
        sigmas[j] /= Npis[j]
        sigmas[j] = np.sqrt(sigmas[j])

    ll_new = 0.0
    for i in range(N):
        s = 0
        for j in range(K):
            s += pis[j] * norm.pdf(xs[i],
                                   loc=mus[j],
                                   scale=sigmas[j])
        # compute new log -likelihood:
        ll_new += np.log(s)

    return pis, mus, sigmas, taus, ll_new
```

As with the k-means algorithm, we now want to watch the fit improve over the course of the iteration. First we knock together some test data:

```
np.random.seed(12345)
pi = 1.0/3
real_mu_1, real_sigma_1 = 0, 1.5
real_mu_2, real_sigma_2 = 6, 3.5
component1_sample = norm.rvs(real_mu_1,
                             real_sigma_1,
                             int(pi*1500))
component2_sample = norm.rvs(real_mu_2,
                             real_sigma_2,
                             int((1 -pi)*1500))
mixture_samples = np.concatenate((component1_sample,
                                  component2_sample))

fig, ax = plt.subplots()
hist = ax.hist(mixture_samples, bins=51,
               color='grey', alpha=.5,
              label="weighted sum")
bins = hist[1]
ax.hist(component1_sample, alpha=.75, bins=bins,
       color='navy', label="component 1")
ax.hist(component2_sample, alpha=.75, bins=bins,
       color='gold', label="component 2")
ax.set_xlabel("$x$")
ax.set_ylabel("density")
ax.legend()
plt.show()
```

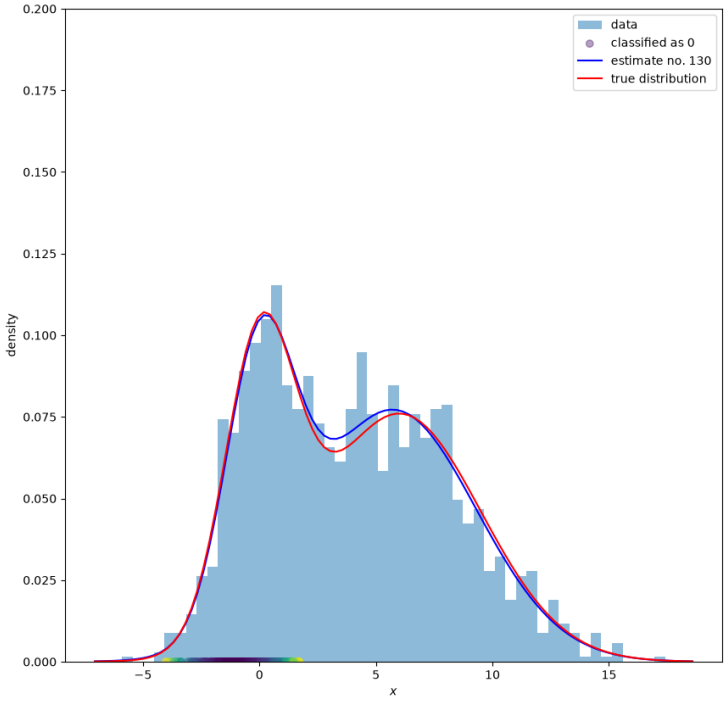



```
# set up the initial theta:
pis =  np.ones(2) + np.random.random(2)
pis /= pis.sum()
mus =  np.random.random(2) * 2*np.std(mixture_samples)\
       + np.mean(mixture_samples)
sigmas =  np.sqrt(np.random.random(2) + 1)

# the true parameters vs. the initial ones:
fig, ax = plt.subplots(figsize=(10,10))
hist = ax.hist(mixture_samples, bins=51,
               density=True,
               label="data")
xmin, xmax = ax.get_xlim()
norm_pdf_domain = np.linspace(xmin, xmax, 100)
ax.plot(norm_pdf_domain,
         pis[0]*norm.pdf(norm_pdf_domain,
                         mus[0], sigmas[0]) +
         pis[1]*norm.pdf(norm_pdf_domain,
                         mus[1], sigmas[1]),
         color='blue',
         label="mixture estimate")
ax.vlines(real_mu_1, 0, .2,
          color='purple', label='true $\\mu_1$')
ax.vlines(real_mu_2, 0, .2,
          color='tomato', label='true $\\mu_2$')
ax.vlines(mus[0], 0, .2,
          color='fuchsia', label='$\\mu_1^{(0)}$')
ax.vlines(mus[1], 0, .2,
          color='orange', label='$\\mu_2^{(0)}$')
ax.set_xlabel("$x$")
ax.set_ylabel("density")
ax.legend()
plt.show()
```




```
import matplotlib.animation as animation
from IPython import display as ipydisplay
import io, base64
from sklearn.cluster import KMeans

def init_theta(samples):
    pis = np.ones(2) / 2
    mus = np.random.random(2) * 2*np.std(samples)\
          + np.mean(samples)
    # a better way would be:
    #mus = KMeans(2).fit(samples.reshape( -1,1)).cluster_centers_
    sigmas =  np.ones(2)
    return pis, mus, sigmas

def em_iteration_draw(run_nr, ax, samples, history):
    pis, mus, sigmas, taus = history[run_nr][0:4]
    ys = np.zeros_like(samples, dtype=bool)
    if(len(taus) == len(samples)):
        for i in range(len(samples)):
            if(taus[i,1] > 0.5):
                ys[i] = 1
    ax.clear()
    ax.set_ylim(0, 0.2)
    ax.hist(samples, bins=51,
            density=True, alpha=0.5,
            label="data")
    if(len(taus) == len(samples)):
        ax.scatter(samples[ys], np.zeros_like(samples[ys]),
                   alpha=0.4, c=taus[ys][:,0],
                   label="classified as 0")
    xmin, xmax = ax.get_xlim()
    norm_pdf_domain = np.linspace(xmin, xmax, 100)
    ax.plot(norm_pdf_domain,
            pis[0]*norm.pdf(norm_pdf_domain,
                            mus[0], sigmas[0]) +
```

```
            pis[1]*norm.pdf(norm_pdf_domain,
                            mus[1], sigmas[1]),
            color='blue',
            label="estimate no. %d" % (run_nr))
    ax.plot(norm_pdf_domain,
            pi    *norm.pdf(norm_pdf_domain,
                            real_mu_1, real_sigma_1) +
            (1 -pi)*norm.pdf(norm_pdf_domain,
                            real_mu_2, real_sigma_2),
            color='red',
            label="true distribution")
    ax.set_xlabel("$x$")
    ax.set_ylabel("density")
    ax.legend()

def plot_ani(samples, history):
    fig, ax = plt.subplots(figsize=(10, 10))
    ani = animation.FuncAnimation(fig, em_iteration_draw,
                                  fargs=(ax, samples, history),
                                  interval=1000,
                                  frames=len(history))
    video = ani.to_html5_video()
    # Output the last frame as a PNG as well: the web export shows the video,
    # the PDF export (which cannot do HTML) uses the still image.
    png = io.BytesIO()
    fig.savefig(png, format='png')
    ipydisplay.display({'text/html': video,
                        'image/png': base64.b64encode(png.getvalue()).decode('ascii')},
                       raw=True)
    plt.close()


max_iter = 210
tol = 0.01
pis, mus, sigmas = init_theta(mixture_samples)
taus = np.zeros((len(mixture_samples), len(pis)))
# record all values in each iteration step for plotting:
history = [(pis, mus, sigmas, taus, 0)]

for i in range(max_iter):
    pis, mus, sigmas = history[ -1][0:3]
    history += [EM(mixture_samples,
                   pis,
                   mus,
                   sigmas)]
    if np.abs(history[ -1][ -1] - history[ -2][ -1]) < tol:
        break

plot_ani(mixture_samples, history)
```



Multivariate data For the more general multivariate case we shall not labour through the derivation again, but go straight to the corresponding code:

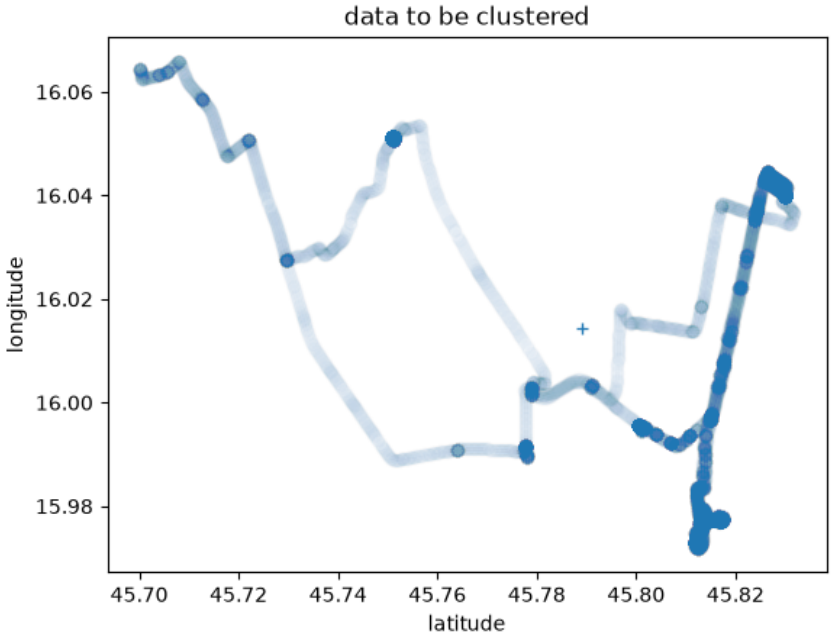
```
from scipy.stats import multivariate_normal as mvn

def EMmulti(xs, pis, mus, sigmas, tol=0.01, max_iter=100):
    """vectorized E -M for multivariate Gaussian mixtures"""

    n, d = xs.shape
    k = len(pis)

    ll_old = 0.0
    for i in range(max_iter):

        # (E) step
        taus = np.zeros((k, n))
        for j in range(k):
            taus[j, :] = pis[j] * mvn(mus[j], sigmas[j]).pdf(xs)
        taus /= taus.sum(axis=0)

        # (M) step
        pis = taus.sum(axis=1)
        pis /= n

        mus = np.dot(taus, xs)
        mus /= taus.sum(axis=1)[:, None]

        sigmas = np.zeros((k, d, d))
        for j in range(k):
            xcs = xs - mus[j, :]
            sigmas[j] = (taus[j,:,None,None]\
                         * np.matmul(xcs[:,:,None],
                                     xcs[:,None,:])).sum(axis=0)
```

```
        sigmas /= taus.sum(axis=1)[:,None,None]

        ll_new = 0.0
        for pi, mu, sigma in zip(pis, mus, sigmas):
            ll_new += pi * mvn(mu, sigma).pdf(xs)
        ll_new = np.log(ll_new).sum()

        if np.abs(ll_new - ll_old) < tol:
            break
        ll_old = ll_new

    return ll_new, pis, mus, sigmas
```

7.8.6 Example applications

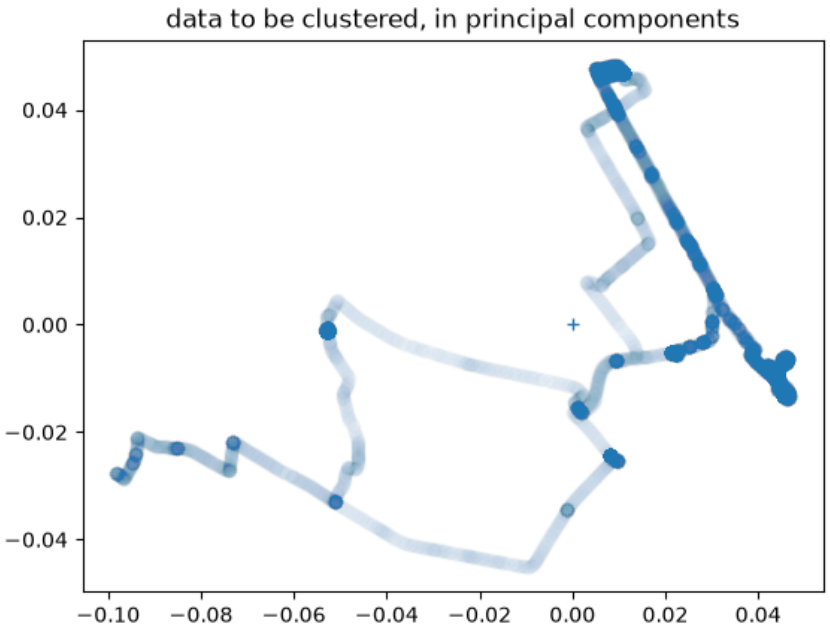
Whereabouts `data.csv` contains a record of whereabouts. The data come, under the MIT licence, from a blog post which explains how to embed OpenStreetMaps – though that is not needed here.

```
data = np.genfromtxt("data.csv", delimiter=',')
fig, ax = plt.subplots()
ax.scatter(data[:,0], data[:,1], alpha=0.01)
ax.set_title("data to be clustered")
ax.set_xlabel("latitude")
ax.set_ylabel("longitude")
ax.plot(np.mean(data[:,0]), np.mean(data[:,1]), marker="+")
plt.show()
```

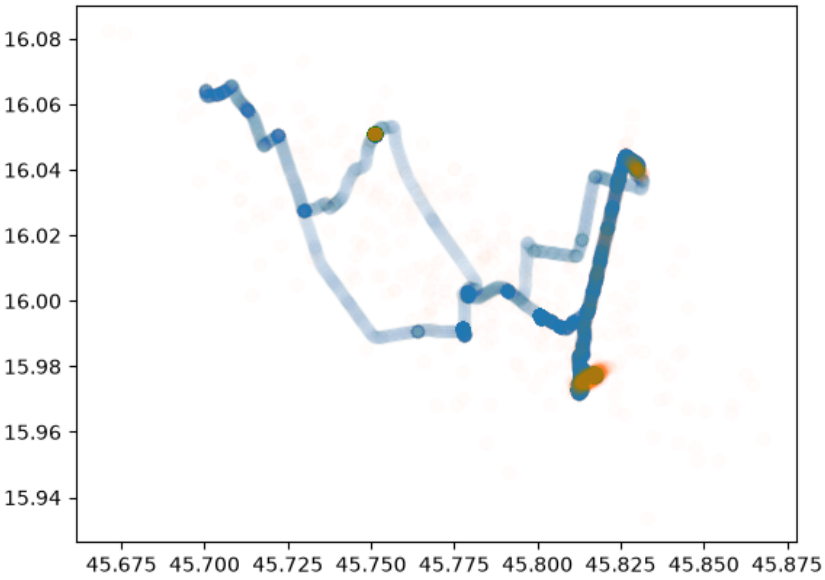


The transformation to principal components is of little use with these data:

```
from sklearn.decomposition import PCA

pc = PCA().fit_transform(data)
fig, ax = plt.subplots()
ax.scatter(pc[:,0], pc[:,1], alpha=0.01)
ax.set_title("data to be clustered, in principal components")
ax.plot(np.mean(pc[:,0]), np.mean(pc[:,1]), marker="+")
plt.show()
```




```
np.random.seed(1234321)
pis = np.array([0.2,0.2,0.2,0.2,0.2])
mus = np.random.random((5,2)) + np.mean(data)
sigmas = np.array([np.eye(2)] * 5)
ll_new, pis, mus, sigmas = EMmulti(data, pis, mus, sigmas)

ys = np.concatenate([mvn(m, s).rvs(int(p*2000))
                     for p, m, s in zip(pis, mus, sigmas)])

fig, ax = plt.subplots()
ax.scatter(data[:,0],data[:,1],alpha=0.01)
ax.scatter(ys[:,0],ys[:,1],alpha=0.01)
plt.show()
```



One interpretation would be that we have now identified the places where the longer stays happened.

Iris classification We now use scikit-learn's GMM (Gaussian mixture model) implementation to model the Iris dataset.

```
from sklearn.mixture import GaussianMixture
```

```
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
x, y = load_iris(return_X_y=True)
k=3

# transform to principal components for plotting:
xpc = PCA().fit_transform(x)

# fit with the EM algorithm:
gm = GaussianMixture(n_components=k,
                     random_state=42,
                     max_iter=100,
                     init_params='k -means++',
                     verbose=1).fit(xpc)
if(gm.converged_):
    print("converged after %d steps" % gm.n_iter_)
else:
    print("could not reach convergence")

Initialization 0
  Iteration 10
Initialization converged.
converged after 19 steps
```

From the Gaussian mixture model we can also obtain the individual probabilities:

```
y_pred = gm.predict(xpc)
y_pred_proba = gm.predict_proba(xpc)
y_pred_maxproba = np.max(y_pred_proba,
                         axis = 1)
```

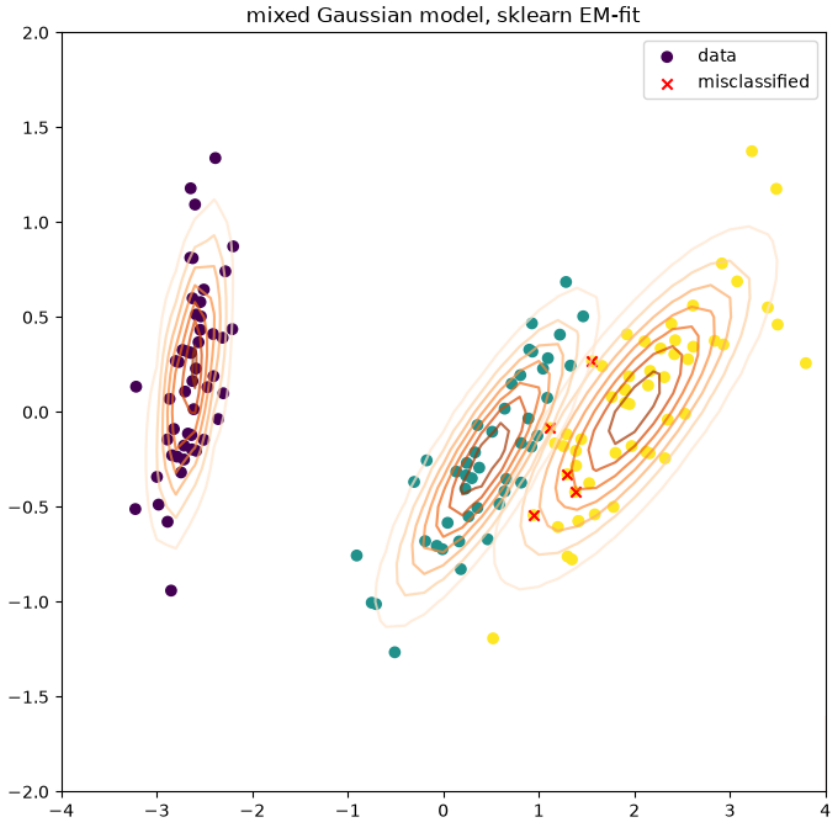
Here we look for the permutation that agrees best with the actual classes:

```
from itertools import permutations
best_accuracy = 0.0
for permutation in permutations(range(k)):
    y_pred_permuted = np.array([permutation[value]
                                for value in y_pred])
    this_accuracy = accuracy_score(y_pred_permuted, y)
    if(this_accuracy > best_accuracy):
        best_accuracy = this_accuracy
        y_pred_bestperm = y_pred_permuted

print("Accuracy: %2.f%%" % (100*best_accuracy))

Accuracy: 97%
```

And finally we plot what the result looks like:

```
# we look directly at the marginal distributions
# in the first 2 principal components:
mvns = [mvn(mean=mu[:2],
            cov=sigma[:2,:2],
            allow_singular=True)
        for mu, sigma in zip(gm.means_,
                             gm.covariances_)]

x_wrongs = xpc[y_pred_bestperm != y]

fig, ax = plt.subplots(figsize=(8,8))
ax.set_title("mixed Gaussian model, sklearn EM -fit")
xs, ys = np.mgrid[ -4:4.1:.2, -2:2.1:.1]
for i in range(k):
    zs = mvns[i].pdf(np.dstack((xs, ys)))
    ax.contour(xs, ys, zs,
               cmap=plt.cm.Oranges, alpha=.7)
```

```
ax.scatter(xpc[:,0], xpc[:,1],
           c=y_pred_bestperm, vmin=0, vmax=k -1,
           label='data')
ax.scatter(x_wrongs[:,0], x_wrongs[:,1],
           c='red', marker='x',
           label='misclassified')
ax.legend()
plt.show()
```



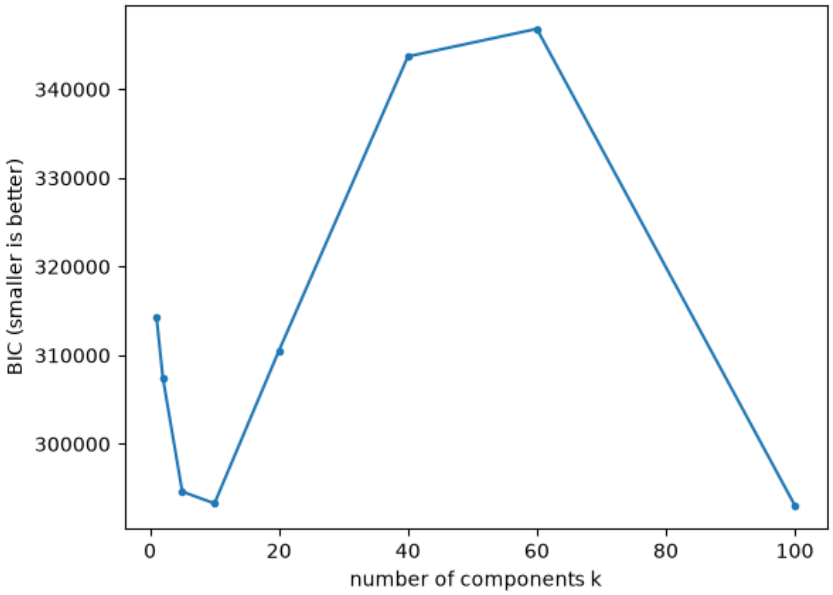
Handwritten digits On the UCI ML hand-written digits dataset we look at how well such a GMM works and how it can be used generatively, to produce new samples of a class.

```
from sklearn.datasets import load_digits
x,y = load_digits(return_X_y=True)
vargoal = .95 # keep so much of variance
pca = PCA(vargoal)
pca.fit(x)
print("kept %d of %d dimensions for %2.f%% of variance "
      % (len(pca.components_), x.shape[1], 100*vargoal))
data = pca.transform(x)

kept 29 of 64 dimensions for 95% of variance
```

> **Note**
> **Looking ahead (not part of the course proper):** the Bayesian information criterion in this section is one of several ways of choosing the number of components, and is not needed in the course.

How many components should the model have? In the chapter on model selection we met tools for questions of this kind. For mixture models the **Bayesian information criterion** (BIC) is in common use: $\mathrm{BIC} = -2\log L + m\log n$, with the log-likelihood $\log L$, the number m of model parameters and the sample size n. Every additional component increases m and has to earn that back first through a better likelihood. One picks the k with the smallest BIC:

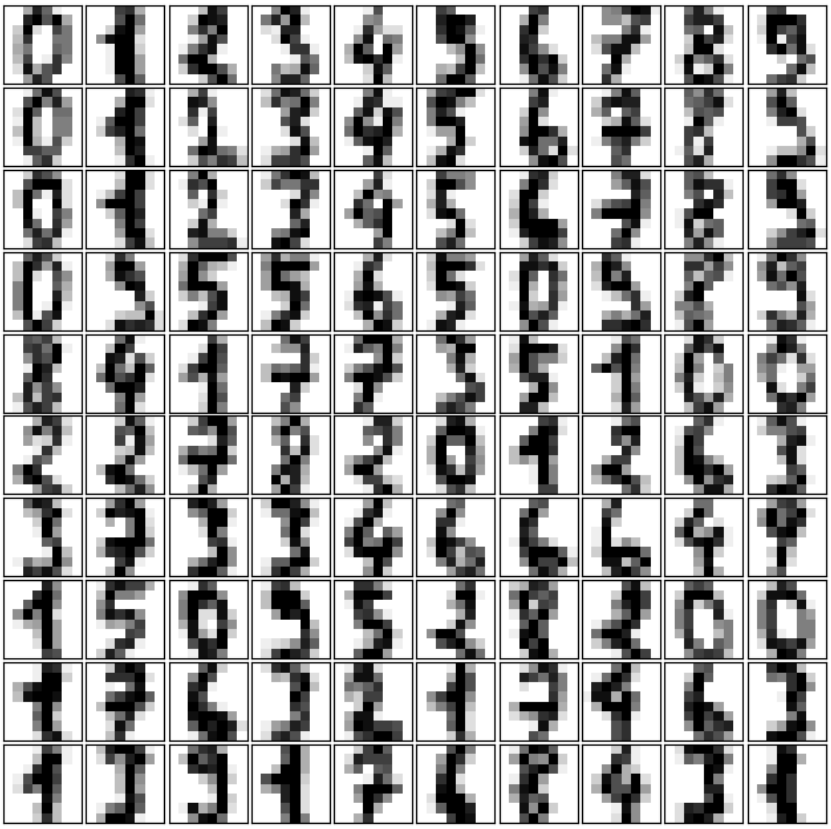
```
ks = [1, 2, 5, 10, 20, 40, 60, 100]
bics = [GaussianMixture(k, random_state=23).fit(data).bic(data) for k in ks]

fig, ax = plt.subplots()
ax.plot(ks, bics, marker=".")
ax.set_xlabel("number of components k")
ax.set_ylabel("BIC (smaller is better)")
plt.show()
print("smallest BIC at k =", ks[np.argmin(bics)])
```



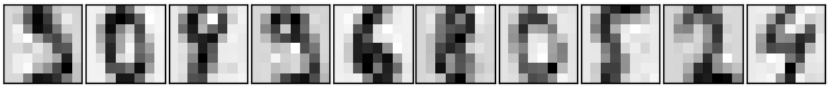

```
smallest BIC at k = 100
```

Here the BIC keeps falling right up to the largest k tested: the digits evidently break up into very many small clusters (different ways of writing the same digit), and anyone who wants to find the minimum has to extend the range. In what follows we nevertheless try out various k by hand, to see what the individual components learn.

```
def plot_digits(x):
    so_many_rows = min(100, len(x)) // 10
    fig, ax = plt.subplots(so_many_rows, 10, figsize=(8, 8),
    subplot_kw=dict(xticks=[], yticks=[]))
    fig.subplots_adjust(hspace=0.05, wspace=0.05)
    for i, axi in enumerate(ax.flat):
        im = axi.imshow(x[i].reshape(8, 8), cmap='binary')
        #im.set_clim(0, 16)
    plt.show()

plot_digits(x)
```



```
gm = GaussianMixture(100, random_state=23)
gm.fit(data)

new_data, new_labels = gm.sample(10)
new_data_in_old_coordinates = pca.inverse_transform(new_data)
plot_digits(new_data_in_old_coordinates)
print(new_labels)
```



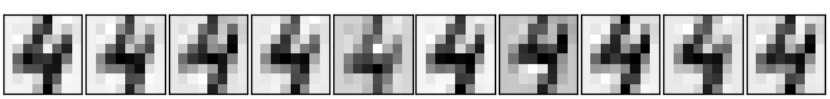
```
[ 1 15 24 49 57 60 62 65 68 99]
```

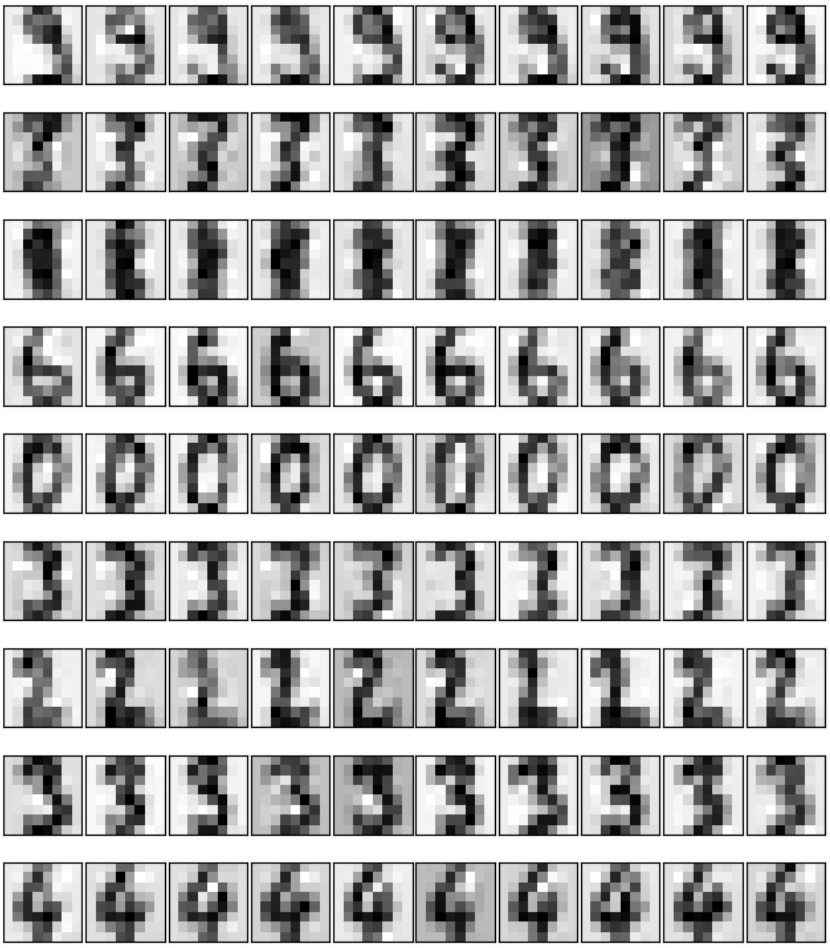
These new digits were generated for us by the GMM. But why are the labels that go with them not the digits 0–9?

Because in training we never tried to have exactly one normal distribution per digit in the mixture. With the 100 individual normal distributions used here, generating new digits works all the same. If we additionally run the classification separately (with SVMs, say – or indeed directly, as a latent variable in the EM algorithm), we have a model that can also generate new digits of a particular class on demand (a 7, for instance).

To gain a little insight into our model, let us look at individual mixture components:

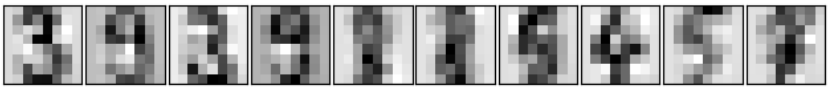
```
for i in range(10):
    mu = gm.means_[i]
    sigma = gm.covariances_[i]
    new_data = mvn(mu, sigma).rvs(10)
    new_data_in_old_coordinates = pca.inverse_transform(new_data)
    plot_digits(new_data_in_old_coordinates)
```





So we do see that each mixture component stays within a single class – there are simply several different "zeros". With only 10 mixture components, the model looks worse:

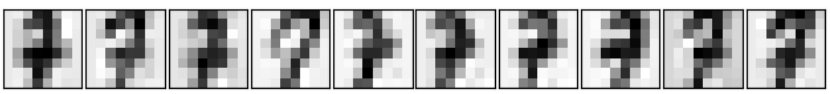
```
gm = GaussianMixture(10, random_state=23)
gm.fit(data)

new_data, new_labels = gm.sample(10)
new_data_in_old_coordinates = pca.inverse_transform(new_data)
plot_digits(new_data_in_old_coordinates)
print(new_labels)
```



```
[1 1 1 1 2 2 5 7 8 9]
```

To generate, say, a 7 within our paradigm and without additional algorithms, we can train directly on the subset of the data consisting of the 7s:

```
gm7 = GaussianMixture(10, random_state=23)
gm7.fit(data[y == 7])

new_data, new_labels = gm7.sample(10)
new_data_in_old_coordinates = pca.inverse_transform(new_data)
plot_digits(new_data_in_old_coordinates)
```



Fashion-MNIST Simply because we can, let us look at exactly the same thing on the Fashion-MNIST dataset:

```
x_train, y_train = load_mnist(kind='train')
#x_test, y_test = load_mnist(kind='t10k')

def show_fashion_image(raw_data, ax):
    im = ax.imshow(raw_data.reshape(28, 28), cmap='binary')
    #im.set_clim(0, 255) # gets computed by matplotlib anyway
```

Let us see how long PCA.fit and EM (gm.fit) take on this somewhat larger dataset:

```
vargoal = .90
pca = PCA(vargoal)
%timeit pca.fit(x_train)
pca.fit(x_train)

4.2 s \pm 1.22 s per loop (mean \pm std. dev. of 7 runs, 1 loop each)

PCA(n_components=0.9)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

[x]PCA

?Documentation for PCAiFitted

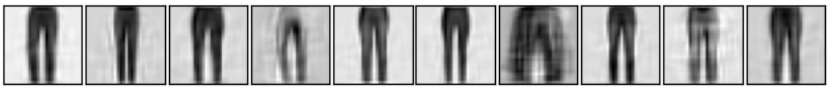
```
print("kept %d of %d dimensions for %2.f%% of variance "
      % (len(pca.components_), x_train.shape[1], 100*vargoal))
fashion_data = pca.transform(x_train)

kept 84 of 784 dimensions for 90% of variance

%%time
gm = GaussianMixture(50, random_state=23)
gm.fit(fashion_data[y_train == 1])

CPU times: user 4min 36s, sys: 260 ms, total: 4min 36s
Wall time: 21.9 s

GaussianMixture(n_components=50, random_state=23)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

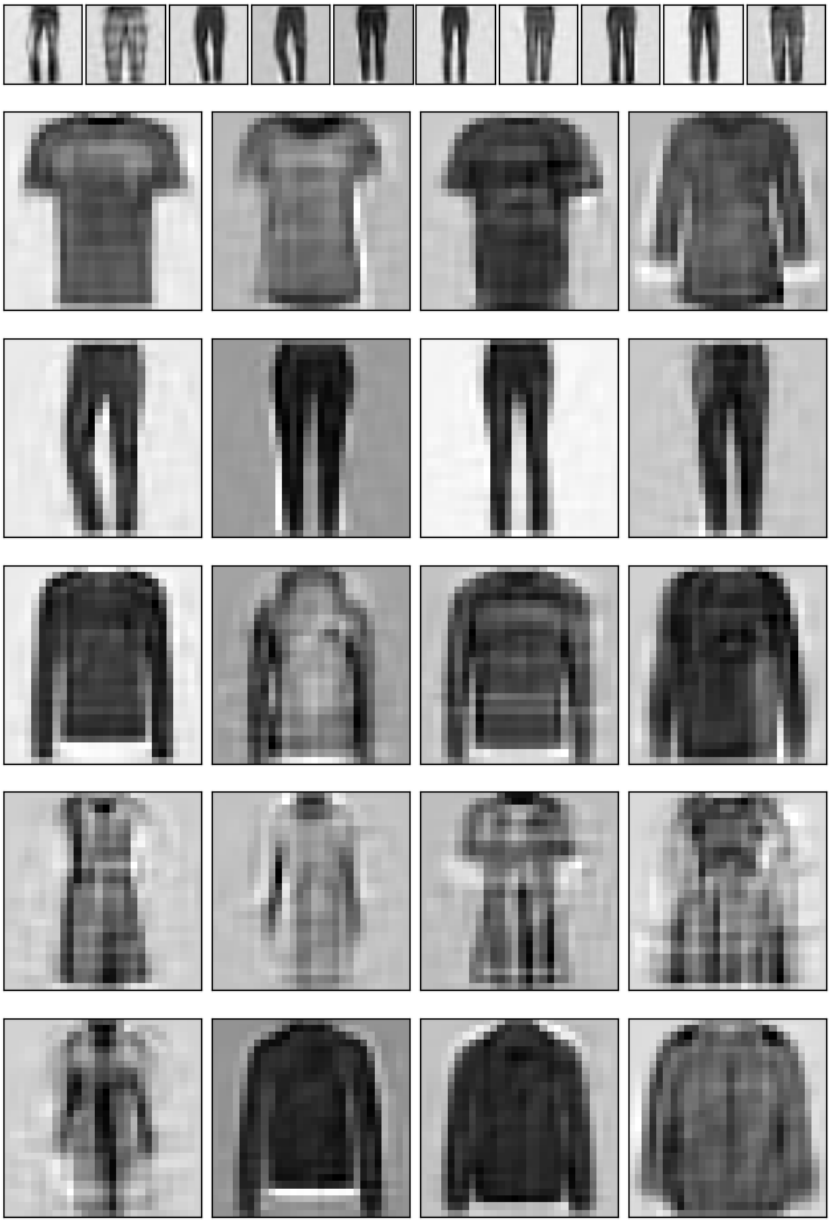
[x]GaussianMixture

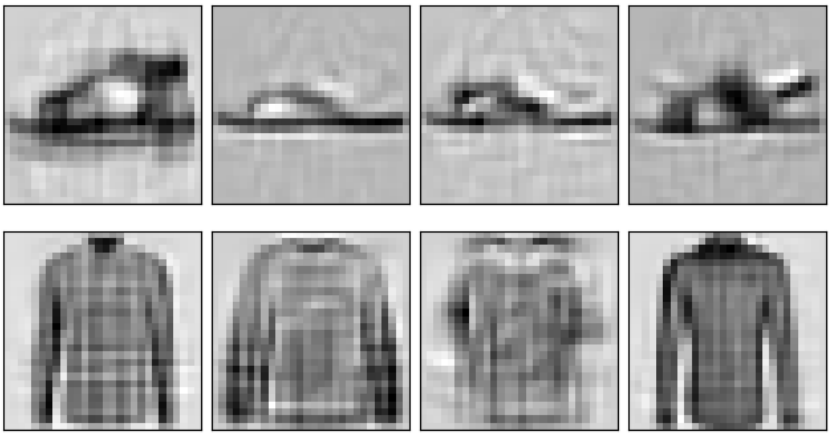
?Documentation for GaussianMixtureiFitted

```
def draw_gm_sample(gm, pca, n=10):
    new_data, new_labels = gm.sample(n)
    new_data_in_old_coordinates = pca.inverse_transform(new_data)

    fig, axs = plt.subplots(1, len(new_data),
                            figsize=(8, 8),
                            subplot_kw=dict(xticks=[],
                                            yticks=[]))
    fig.subplots_adjust(hspace=0.05, wspace=0.05)
    for datapoint, ax in zip(new_data_in_old_coordinates, axs):
        show_fashion_image(datapoint, ax)
    plt.show()

draw_gm_sample(gm, pca, n=10)
```



```
%time highres_pca = PCA(.94).fit(x_train)
highres_data = highres_pca.transform(x_train)

CPU times: user 53.8 s, sys: 78.8 ms, total: 53.9 s
Wall time: 4.27 s

highres_gm = GaussianMixture(90, tol=1e -2, random_state=42)
%time highres_gm.fit(highres_data[y_train == 1])
draw_gm_sample(highres_gm, highres_pca, n=10)
for i in range(7):
    highres_gm.fit(highres_data[y_train == i])
    draw_gm_sample(highres_gm, highres_pca, n=4)

CPU times: user 2min 15s, sys: 169 ms, total: 2min 15s
Wall time: 10.9 s
```





7.8.7 Looking back and looking ahead

With that, these notes come to an end. The road through the machine learning part of the course was a chain of ideas, each built on the last:

- **k-means** was our first algorithm: measure distances, assign points to centroids, recompute the centroids.
- In the chapter on **statistical models** we recognised k-means as a maximum likelihood estimator and introduced the notions of estimator, bias and likelihood.
- **Principal component analysis** and **linear regression** were the first methods to rest on the covariance matrix and on minimising squared deviations, respectively, with regularisation as the remedy for overfitting.
- The **kernel trick** showed that many of these methods need only distances or inner products and can therefore be carried over to nonlinear structures.
- **Statistical tests** and **model selection** dealt with how to assess results and models without letting the sample fool you: significance, cross-validation, systematic hyperparameter search.
- **Expectation Maximization** closes the circle: a Gaussian mixture model is a 'soft' k-means, one that follows from the maximum likelihood method and Jensen's inequality, and with criteria such as the BIC even the number of components can be chosen on good grounds.

We have deliberately left out, among other things, neural networks and deep learning, time series analysis, reinforcement learning, and everything that comes after the model: deployment, monitoring and the maintenance of models in production. The foundations laid in these notes – probability, expected value, covariance, likelihood, regularisation and validation – turn up again in all of those subjects. The course *Introduction to Deep Learning* follows on directly from these notes; in particular it uses random variables and the expected value, cross-entropy as a loss function, linear regression as the simplest model, Numpy shapes and broadcasting, the train-test split with validation, and the lower bound $L(\theta, q)$ from this chapter, there called the ELBO. Anyone who would like to read further will find the mathematical depth in Friedman et al. [2001] and Deisenroth et al. [2020], and the practitioner's perspective in Schutt and O'Neil [2013].

References

M. Allen, M. Babiker, Y. Chen, and H. C. de Coninck. *IPCC SR15: Summary for Policymakers*. Intergovernmental Panel on Climate Change, 10 2018.

C. R. Charig, D. R. Webb, S. R. Payne, and J. E. A. Wickham. Comparison of treatment of renal calculi by open surgery, percutaneous nephrolithotomy, and extracorporeal shockwave lithotripsy. *British Medical Journal (Clinical Research Edition)*, 292(6524):879–882, 1986. doi: 10.1136/bmj.292.6524.879.

M. P. Deisenroth, A. A. Faisal, and C. S. Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.

D. Fox. Entropie. *Datenschutz und Datensicherheit (DuD)*, 32:543, 2008.

J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 3 edition, 2006.

J. Friedman, T. Hastie, and R. Tibshirani. *The Elements Of Statistical Learning*, volume 1 of *series in statistics*. Springer, 2001.

H. Haller and S. Krauss. Misinterpretations of Significance: A problem students share with their teachers? *Methods of Psychological Research Online*, 7(1):1–20, 2002.

T. Herndon, M. Ash, and R. Pollin. Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. *Cambridge Journal of Economics*, 38(2):257–279, 2014. doi: 10.1093/cje/bet075.

J. P. A. Ioannidis. Why Most Published Research Findings Are False. *PLoS Medicine*, 2(8):e124, 8 2005. ISSN 1549-1676. doi: 10.1371/journal.pmed.0020124. URL http://dx.doi.org/10.1371/journal.pmed.0020124.

J. Matejka and G. Fitzmaurice. Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1290–1294, 2017. doi: 10.1145/3025453.3025912.

C. Molnar. *Modeling Mindsets: The Many Cultures Of Learning From Data*. Independently published, nov 15 2022. ISBN 979-8358729339.

A. Pöge. Persönliche Codes bei Längsschnittuntersuchungen III. *Methoden, Daten, Analysen (mda)*, 5 (1):109–134, 2011. URL https://nbn-resolving.org/urn:nbn:de:0168-ssoar-255044.

C. M. Reinhart and K. S. Rogoff. Growth in a Time of Debt. *American Economic Review*, 100(2): 573–578, 2010. doi: 10.1257/aer.100.2.573.

E. Schubert. Stop using the elbow criterion for k-means and how to choose the number of clusters instead. *ACM SIGKDD Explorations Newsletter*, 25(1):36–42, 6 2023. ISSN 1931-0153. doi: 10.1145/3606274.3606278. URL http://dx.doi.org/10.1145/3606274.3606278.

R. Schutt and C. O'Neil. Doing Data Science: Straight Talk from the Frontline, 2013.

C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27: 379–423, 623–656, 1948.

A. Z. Spector, P. Norvig, C. Wiggins, and J. M. Wing. *Data Science in Context: Foundations, Challenges, Opportunities*. Cambridge University Press, 2022.