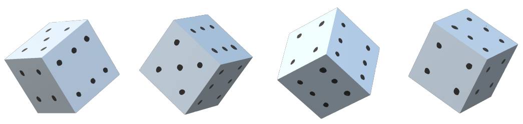
IB Foundations of Data Science

Damon Wischik, Computer Science, Cambridge University



Contents

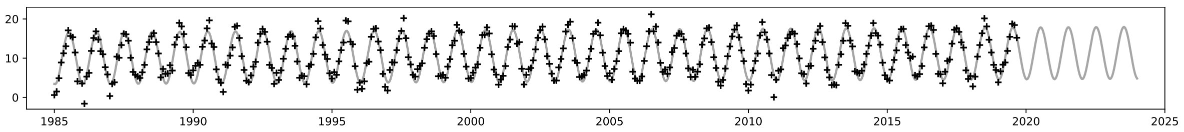
ACKNOWLEDGEMENTS

Thanks to Richard Gibbens, Jakub Perlin, Thomas Sauerwald, Alicja Chaszczewicz, and L.A. Mlodzian for spotting bugs in earlier versions of these notes.

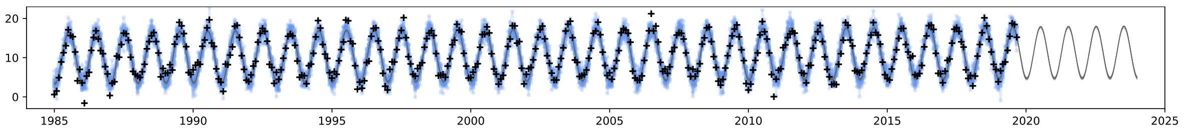
October 11, 2019 at 21:13:00

Part I
Learning with probability models

Many tools in machine learning, from simple linear regression to neural network classifiers, come down in the end to writing out a probability model and then estimating the model's parameters by using data. This is called *fitting the model*. Here's an example of a fitted model, for Cambridge temperatures. The crosses mark the datapoints, and the grey line shows the fitted model. and the datapoints are coloured by whether they are above or below the line.



Why a *probability* model? A probability model specifies everything that we think might have happened and how likely it is. For example "temperatures are cyclical, and on top of this there are fluctuations, but a fluctuation of more than a few °C is highly unlikely". We need a model that can describe noisy data, in order to be able to learn from noisy data.



When a data scientist looks at a dataset, she doesn't just see the datapoints, she also sees a cloud of all the counterfactual possibilities of what the datapoints might have been. Only if we know what might plausibly have happened can we judge the significance of what actually did happen.[1]

A large part of this course, and of machine learning in general, is knowing enough building blocks to come up with useful probability models. The building blocks are random variables, which we look at from a programming perspective in section 1.3 and a mathematical perspective in section 1.4.

We'll start with some simple probability models, and build up to more advanced models for clustering (Gaussian mixture models in section 1.5) and for learning from labelled data (image classification in section 1.6).

Section 2 dives deep into one particular class of probability model, namely linear regression. This is a flexible and interpretable class of model, and has fast routines for fitting to data. It should be your go-to method for all sorts of data science and machine learning problems, the second thing you try to get a sense of the data you're working with. (The first thing you try should be simple tabulation!)

Fitting a model, for anything beyond the most basic toy examples, involves a bit of maths to figure out an expression to optimize, and then numerical optimization by computer. The workhorse of machine learning is computational optimization, for example by the so-called gradient descent method. Andrej Karpathy, director of AI and vision at Tesla, writes[2] that "Gradient descent can write code better than you. I'm sorry." He elaborates

> *Software 1.0 is code we write. Software 2.0 is code written by the optimization based on an evaluation criterion (such as "classify this training data correctly"). It's likely that any setting where the program is not obvious but one can repeatedly evaluate the performance of it (e.g.—did you classify some images correctly? do you win games of Go?) will be subject to this transition, because the optimization can find much better code than what a human can write.*

Software 2.0 still needs a human to decide out what to optimize—which comes from probability models. Sections 1.1 and 1.2 describe how to take a given probability model and fit it to data using numerical optimization, first with maths and then with Python.

[1]The counterfactual climate plot actually shows many grey lines superimposed, each corresponding to a different counterfactual possibility. The grey lines are almost perfectly aligned, which gives us confidence in the fit.

[2]Andrej Karpathy. *Software 2.0*. Blog. Nov. 11, 2017. URL: `https://medium.com/@karpathy/software-2-0-a64152b37c35` (visited on 11/07/2018).

1. Specifying and fitting models

1.1. Maximum likelihood estimation

tl;dr. Assume we have observed data, and we're told the probability model behind the data. Assume also that this probability model has an unknown parameter, which we wish to estimate.

The *likelihood* is the probability of the observed data, viewed as a function of the unknown parameter. The *maximum likelihood estimator*, or *mle*, is the parameter value that maximizes the likelihood.

Here are some worked example of maximum likelihood parameter estimation for very simple probability models, so simple that we can find the maximum likelihood estimate using basic calculus and we don't need a computer.

Exercise 1.1 (Coin tosses).
Suppose we take a biased coin, and tossed it $n = 10$ times, and observe $x = 6$ heads. Let's use the probability model

$$\mathbb{P}(\text{num. heads} = x) = \binom{n}{x} p^x (1-p)^{n-x}, \quad x \in \{0, 1, \dots, n\}.$$

where p is the probability of heads and $1 - p$ is the probability of tails. What is p?

$\binom{n}{x}$ is the binomial coefficient, equal to $n! \,/\, x!(n-x)!$

The likelihood is the probability of the observed data[3] $x = 6$*, viewed as a function of the unknown parameter* p*: it is*

$$\text{lik}(p) = \binom{n}{x} p^x (1-p)^{n-x}.$$

A sensible estimate for p *is the mle, i.e. the value of* p *that maximizes the likelihood. To find it, solve*

$$\frac{d}{dp}\text{lik}(p) = \binom{n}{x}\Big(xp^{x-1}(1-p)^{n-x} - (n-x)p^x(1-p)^{n-x-1} \Big) = 0$$

which has the solution

$$\hat{p} = \frac{x}{n}.$$

It's often easier to maximize $\log(\text{lik}(\cdot))$ *rather than* $\text{lik}(\cdot)$*: it must give us the same solution, because* log *is an increasing function. In this case,*

$$\log \text{lik}(p) = \kappa + x \log p + (n-x)\log(1-p)$$

where $\kappa = \binom{n}{x}$*. Note that* κ *doesn't depend on* p*—so as far as likelihood is concerned,* κ *is a constant. When we solve*

$$\frac{d}{dp}\log \text{lik}(p) = \frac{x}{p} - \frac{n-x}{1-p} = 0$$

we also get the solution $\hat{p} = x/n$*.*

■

Exercise 1.2 (The plug-in principle).
In the coin toss example, estimate the odds of heads, i.e. estimate $p/(1-p)$.

[3] Why did we take the observed data to be $x = 6$, rather than the pair $(x, n) = (6, 10)$? In general, we take the observed data to be *whatever might be different if we reran the experiment*. The probability model specified in the question describes a coin-tosser who decided on $n = 10$ in advance, and then tossed the coin, and happened to see $x = 6$, i.e. it treats x as the observed data and $n = 10$ as a fixed parameter. But other interpretations of the first sentence are possible—for example, perhaps the coin-tosser kept tossing coins until she saw $x = 6$ heads, and the observed data is n, the number of tosses needed. If we believed this to be the mechanism, we'd have chosen a different probability model.

In the real world, no one ever tells you the probability model or the mechanism behind a dataset, and you have to invent one yourself.

Write h for the odds of heads, $h = p/(1-p)$. One way to estimate h is by rewriting the entire model in terms of h, via the substitution $p = h/(1+h)$:

$$\operatorname{lik}(h) = \binom{n}{x}\Big(\frac{h}{1+h}\Big)^x\Big(\frac{1}{1+h}\Big)^{n-x}.$$

When we maximize this, we find the mle is $\hat{h} = x/(n-x)$.

There is a simpler way to find the mle for h. We've already found the mle for p, $\hat{p} = x/n$. We can just plug this in to the formula for h, to get the mle for h:

$$\hat{h} = \frac{\hat{p}}{1-\hat{p}} = \frac{x/n}{1-x/n} = \frac{x}{n-x}.$$

This so-called plug-in method is derived from the chain rule in calculus. It means that no matter how we happen to have parameterized the model, we'll draw the same conclusions.

∎

Exercise 1.3 (Estimating multiple parameters).

Suppose we ask $n = 100$ people their views on Brexit, and 37 say Leave, 35 say Remain, and the other 28 don't care. Using the probability model

$$\mathbb{P}\big(\text{leavers} = x_L,\ \text{remainers} = x_R\big) = \frac{n!}{x_L!x_R!(n-x_L-x_R)!}p_L^{x_L}p_R^{x_R}(1-p_L-p_R)^{n-x_L-x_R}$$

estimate the parameters p_L and p_R.

The log likelihood is

$$\log\operatorname{lik}(p_L, p_R) = \kappa + x_L\log p_L + x_R\log p_R + (n-x_L-x_R)\log(1-p_L-p_R).$$

This is a function of two variables. To find the maximum, we need to solve two equations simultaneously:

$$\frac{\partial}{\partial p_L}\log\operatorname{lik}(p_L, p_R) = 0 \qquad \textit{and} \qquad \frac{\partial}{\partial p_R}\log\operatorname{lik}(p_L, p_R) = 0.$$

Doing the differentiation,

$$\frac{x_L}{\hat{p}_L} - \frac{n-x_L-x_R}{1-\hat{p}_L-\hat{p}_R} = 0 \qquad \textit{and} \qquad \frac{x_R}{\hat{p}_R} - \frac{n-x_L-x_R}{1-\hat{p}_L-\hat{p}_R} = 0$$

which after some algebra gives

$$\hat{p}_L = \frac{x_L}{n} \qquad \textit{and} \qquad \hat{p}_R = \frac{x_R}{n}.$$

∎

THINGS TO WATCH OUT FOR

It's common to write $\mathbb{P}(\text{num. heads} = x \mid p)$ or $\operatorname{lik}(p \mid x)$ to emphasize that the formula involves both the unknown parameter p and the observed data x. Note that this is NOT a conditional probability, it just happens to use the same vertical bar symbol.

What's the difference between p and $\hat{p}$? We denote by p the unknown parameter, and we denote by $\hat{p}$ the estimate we found for p from the data. Think of the ^ as a mountain top, reminding us that we found a maximum!

Another word for $\hat{p}$ is estimat*or*. This emphasizes that $\hat{p}$ is a function, which takes the observed data as its input and returns an estimated value as its output. When you derive an estimator, scan through your formula and double-check that it doesn't have any unknown parameters. For example, suppose we had tried to solve exercise 1.3 by considering only one of the parameters, e.g. by differentiating $\log\operatorname{lik}(p_L, p_R)$ with respect to p_L and finding where the derivative is zero; we'd have ended up with the answer

$$\hat{p}_L = \big(1-p_R\big)\frac{x_L}{n-x_R}.$$

This is NOT a valid estimator for p_L because the right hand side depends on p_R which is an unknown parameter.

* ✻ *

What does likelihood actually measure? The maximum likelihood procedure is intuitively sensible, but what guarantees do we have about its accuracy? In part III we'll explore some of the paths that statisticians and philosophers have taken in thinking about likelihood and what it can be used for. Here's a question, to start you thinking:

If I toss 3 coins and get 3 heads, the maximum likelihood estimator is $p = 1$. If I toss 1 million coins and get 1 million heads, it's still 1. I should be more confident in the latter case, but how do we measure confidence? And what would it take to persuade me I'll *never* see a tail?

1.2. Numerical optimization

The workhorse of machine learning is numerical optimization, especially the algorithm known as gradient descent. There is much advice to be found about numerical optimization algorithms, but not much that sheds light on the concepts behind data science and machine learning, so we won't say much here. We will simply give a simple general-purpose routine that will be adequate for most of this course.

Because numerical optimization is so important, there are many specialized algorithms. For any useful branch of machine learning, chances are there's some specialized library for efficient numerical optimization within that branch. The practical handout introduces Keras, commonly used in deep learning.

tl;dr. To find the minimum of a function $f : \mathbb{R}^K \to \mathbb{R}$,

```
import scipy.optimize

# The function to minimize. Input x is a length-K list, output is a real number
def f(x):
    return ....

x₀ = [...]  # where to start the search, a length-K list
x̂ = scipy.optimize.fmin(f, x₀)
```

There is no scipy.optimize.fmax, so to maximize $f(x)$ we should find the minimum of $-f(x)$.

The optimization routine isn't omniscient. It will find a local minimum, not necessarily a global minimum. It might fail to find even a local minimum, if the function isn't well-behaved. To make it happy, pick a sensible x_0, based on your understanding of roughly what the answer is likely to be.

To find a local minimum over a constrained domain, it's best to transform parameters into an unconstrained domain. For example,

- Instead of minimizing over $x > 0$, minimize over $y \in \mathbb{R}$ and let $x = e^y$
- Instead of minimizing over $x \in [0, 1]$, minimize over $y \in \mathbb{R}$ and let $x = e^y/(1 + e^y)$
- Instead of minimizing over $(x, y, z) \in \mathbb{R}^3$ such that $x+y+z = 1$, minimize over $(x, y) \in \mathbb{R}^2$ and set $z = 1 - x - y$.

If these sorts of tricks don't work, then you need a specialist optimizer.

Exercise 1.4. Find the maximum over $\sigma > 0$ of

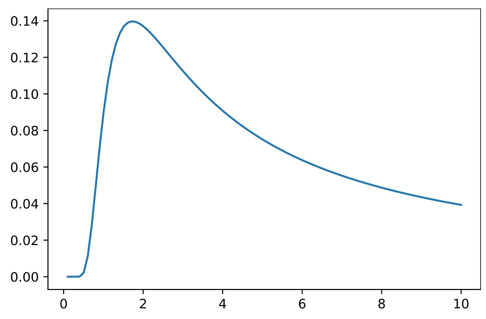
$$f(\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-3/2\sigma^2}$$

Let's optimize over $\tau \in \mathbb{R}$, with the transform $\sigma = e^\tau$. This ensures $\sigma > 0$.

```
import scipy.optimize
import numpy
import matplotlib.pyplot as plt

def f(σ):
    return numpy.exp(−3*0.5/numpy.power(σ,2)) / numpy.sqrt(2*numpy.pi*numpy.power(σ,2))

# From the plot, the maximum is somewhere around σ = 2
# I implemented f using numpy maths functions, so that it's automatically vectorized,
# which is handy for plotting.
fig,ax = plt.subplots()
σ = numpy.linspace(0,10,100)[1:]  # remove σ = 0, where the function doesn't work
ax.plot(σ, f(σ))
plt.show()
```



```
# Optimize in terms of τ = log σ, τ ∈ ℝ
# We want to maximize f, i.e. minimize -f
# fmin returns a list of length 1; unpack it with (τ̂,)=...
(τ̂,) = scipy.optimize.fmin(lambda τ: -f(numpy.exp(τ)), numpy.log(2))
σ̂ = numpy.exp(τ̂)
```

∎

Exercise 1.5 (Softmax transformation).
Find the maximum of

$$f(x_1, x_2, x_3) = 0.2 \log x_1 + 0.5 \log x_2 + 0.3 \log x_3$$

over $x_1, x_2, x_3 \in [0, 1]$ such that $x_1 + x_2 + x_3 = 1$.

Let's optimize over $(\xi_1, \xi_2) \in \mathbb{R}^2$, with the transform

$$x_1 = \frac{e^{\xi_1}}{e^{\xi_1} + e^{\xi_2} + 1}, \quad x_2 = \frac{e^{\xi_2}}{e^{\xi_1} + e^{\xi_2} + 1}, \quad x_3 = \frac{1}{e^{\xi_1} + e^{\xi_2} + 1}.$$

The exponentiation makes sure we get positive values, even for negative ξ_i. The normalization by $e^{\xi_1} + e^{\xi_2} + 1$ ensures we get $x_1 + x_2 + x_3 = 1$. Since the x_i are positive and sum to one, they must be in the range $[0, 1]$. We don't need a third parameter ξ_3, because the constraint $x_1 + x_2 + x_3 = 1$ means there are only two free variables.

```
def f(ξ):  # input: a vector of length 2
    ξ1,ξ2 = ξ
    x = numpy.exp([ξ1,ξ2,0])
    x1,x2,x3 = x / numpy.sum(x)
    return 0.2*numpy.log(x1) + 0.5*numpy.log(x2) + 0.3*numpy.log(x3)

# Initial guess (ξ1, ξ2) = 0, i.e. (x1, x2, x3) = (1/3, 1/3, 1/3)
ξ1,ξ2 = scipy.optimize.fmin(lambda ξ: -f(ξ), [0,0])
x = numpy.exp([ξ1,ξ2,0])
x / numpy.sum(x)
```

∎

There's nothing special about the transform we used; we could use any transform we like, as long as it produces (x_1, x_2, x_3) that satisfy the constraint. This particular choice is a version of the so-called *softmax* transform, widely used in machine learning whenever we want a neural network to output a probability vector.

1.3. Random variables in code

tl;dr. A *random variable* is a function that can give different answers, e.g. a function that calls a random number generator. There are no restrictions on the return type of the function, but for most of the models we study it will be either integer or real.

We can specify a random variable either with source code, or by giving its probability distribution, i.e. by giving the value of

$$\mathbb{P}(X \in A) \qquad \text{for every set } A.$$

Two random variables X and Y are said to be *independent* if

$$\mathbb{P}(X \in A \text{ and } Y \in B) = \mathbb{P}(X \in A)\;\mathbb{P}(Y \in B) \qquad \text{for all } A \text{ and } B.$$

Informally, it means "knowing the value of one gives no information about the other."

Two random variables X and Y are said to be *identically distributed*, written $X \sim Y$, if they have the same probability distribution.

To explain the notation for random variables, we'll be working in this section with standard uniform random variables in the range [0,1], generated in Python by random.random(), and referred to mathematically as $U[0,1]$. The next section discusses several other more interesting random variables.

When we write a maths expression like $X \sim U[0,1]$ we mean "generate X by calling random.random()". Another way of saying this is "X is drawn from the $U[0,1]$ distribution."

It's common to use a hybrid maths / code notation to indicate transformations and combinations of random variables. For example, if we see the maths

$$X \sim U[0,1], \quad Y = aX + b$$

it tells us to generate Y like this:

```
def ry(a,b):
    x = random.random()
    y = a * x + b
    return y
```

In maths expressions we often use capital letters to denote random variables and lower case for constants. This is how we can tell from the maths that X and Y denote random variables while a and b denote constants, which we might as well pass in as parameters. It's good practice to state this explicitly, for example "Let $Y = aX + b$, where a and b are parameters."

Remember that a random variable doesn't *have* any particular value, it's a mechanism for *generating* values. This is clear enough when it's written as a function like ry. But also, if we're discussing a naked script e.g.

```
x = ry(2, 3)
```

then (from the point of view of the person running the script) x is a random variable because it gets a different value every time the script is called.

INDEPEDENT RANDOM VARIABLES

When a maths expression involving two or more random variables, then the way it is written tells you whether they are meant to be generated independently. If we see the maths

$$X_1 \sim U[0,1]\,, \quad X_2 \sim U[0,1]\,, \quad Y = X_1 \log X_2$$

or

$$X_1\,, X_2 \sim U[0,1]\,, \quad Y = X_1 \log X_2$$

it tells us to generate them independently:

```
def ry():
    x₁ = random.random()
    x2 = random.random()
    return x1 * math.log(x2)
```

If we see the maths

$$X \sim U[0,1], \quad Y = X \log X$$

it tells us to use the same X value twice:

```
def ry():
    x = random.random()
    return x * math.log(x)
```

Conversely, if the code makes multiple calls to a random number generator, e.g.

```
(x1, x2) = [random.random() for _ in range(2)]
```

it's telling us that X_1 and X_2 are independent.

EQUAL VALUES OR IDENTICAL DISTRIBUTIONS?

We use a special symbol $\sim$, as in $X \sim Y$, to denote that the two random variables have identical distributions. This is different from writing $X = Y$, which denotes that they are equal in value.

```
# Generate two indepedent random variables X,Y ~ U[0,1].
# They are identically distributed, because they are generated in the same way.
x = random.random()
y = random.random()

# Are they equal in value? No, of course not, they almost always have different values.
assert x != y
```

Two random variables might be identically distributed, even if they come from different source code. Here's an example:

Example 1.6 (Equality of distributions).
Here are two different random variables:

```
def rgeom(p):
    x = 1
    while random.random() > p:
        x = x + 1
    return x

def rgeom2(p):
    def rexp(λ):
        u = random.random()
        return - math.log(u) / λ
    λ = - math.log(1-p)
    z = rexp(λ)
    return math.ceil(z)
```

It can be shown that

$$\mathbb{P}\big(\text{rgeom}(p) = k\big) = \mathbb{P}\big(\text{rgeom2}(p) = k\big) = (1-p)^{k-1}p.$$

Hence the two random variables are identically distributed.

Probability calculations in exercise 3.5 page 42

The maths for this example is left to section 3.3, which includes many other examples and techniques for calculating probabilities.

1.4. Random variables in maths

tl;dr. A random variable is a function that can give different answers. We say that a random variable *takes values* in $\mathcal{S}$, or equivalently that it is $\mathcal{S}$-valued, if the return value of the function is an element of the set $\mathcal{S}$.

Numerical random variables (i.e. that return an integer or real) are so useful that we often write 'random variable' to mean 'numerical random variable', and use other wording when it's not numerical. The most common building blocks for probability models are

- *discrete random variables*, taking integer values;
- *continuous random variables*, taking real values.

A continuous random variable is specified by its density function, while a discrete random variable is specified by its probability mass function. We use the same notation, $\text{Pr}(x)$, for both cases.

Two random variables X and Y are independent if their joint density $\text{Pr}_{X,Y}(x, y)$ satisfies

$$\text{Pr}_{X,Y}(x, y) = \text{Pr}_X(x)\,\text{Pr}_Y(y).$$

This extends to collections of random variables: $(X_1, \ldots, X_n)$ are independent if

$$\text{Pr}(x_1, \ldots, x_n) = \text{Pr}_{X_1}(x_1) \times \cdots \times \text{Pr}_{X_n}(x_n).$$

A collection of independent random variables is called a *random sample*. If all the X_i are drawn from the same distribution, it's called an *independent identically-distributed* or *i.i.d.* random sample.

The goal of this section is to introduce the mathematical notation used for describing random variables, especially $\text{Pr}(\cdot)$ and random samples. This is all that's needed for basic model fitting, as described in the next two sections. Most tools from machine learning and statistics are oriented around random samples, and $\text{Pr}(\cdot)$ is a fundamental quantity for nearly all learning methods.

To design more advanced probability models, and for Bayesian learning, we'll need some more advanced tools for manipulating random variables. This is left to section 3.

1.4.1. STANDARD RANDOM VARIABLES

As a data scientist you should be familiar with a repertoire of standard random variables. Many of the standard random variables take one or more parameters. For example, let X be the number of heads seen from n tosses of a biased coin, for which the probability of heads is p. This is called the Binomial distribution, written $X \sim \text{Bin}(n, p)$. In Python, numpy.random.binomial(n,p).

Here are some of the standard random variables that you will use over and over again. There's a longer list in the appendix, including Python calls.

Common random variables: appendix, page 61

DISCRETE RANDOM VARIABLES

Uniform	$X \sim U\{a,\dots,b\}$	An integer, uniformly distributed in $\{a,\dots,b\}$.
Geometric	$X \sim \text{Geom}(p)$	Counts the number of failures before a success, where success happens with probability p
Binomial	$X \sim \text{Bin}(n,p)$	Counts the number of heads in n tosses of a biased coin, where p is the probability of heads
Poisson	$X \sim \text{Pois}(\lambda)$	Used for modelling counts such as the number of buses that arrive in an interval of time.

CONTINUOUS RANDOM VARIABLES

Uniform	$X \sim U[a,b]$	A floating point value, uniformly distributed in the interval $[a,b]$.
Exponential	$X \sim \text{Exp}(\lambda)$	Used for modelling lifetimes, e.g. the time until the next bus arrives.
Normal / Gaussian	$X \sim N(\mu,\sigma^2)$	Used to model magnitudes, e.g. the height of a person.
Beta	$X \sim \text{Beta}(a,b)$	Arises in Bayesian inference.

Looking up random variables. The first place to look up a random variable is Wikipedia. Below are two examples, the entries for the Poisson distribution and the Normal distribution. Some terminology: "support" means "X takes values in ..."; PMF is the probability mass function and PDF is the probability density function, both of which we're writing as $\Pr_X(\cdot)$; and CDF is the cumulative distribution function $\mathbb{P}(X \leq \cdot)$.

The second place to look up a random variable is the documentation for numpy.random, which has routines for sampling from random variables, and for scipy.stats, which has routines for densities and cumulative distribution functions among others.

Poisson distribution

Notation	$\text{Pois}(\lambda)$
Parameters	$\lambda > 0$ (rate)
Support	$k \in \{0,1,\dots\}$
PMF	$\dfrac{\lambda^k e^{-\lambda}}{k!}$
CDF	$e^{-\lambda}\sum_{i=0}^{\lfloor k\rfloor}\dfrac{\lambda^i}{i!}$
Mean	λ
Variance	λ

Normal distribution

Notation	$N(\mu,\sigma^2)$
Parameters	mean $\mu \in \mathbb{R}$, scale $\sigma > 0$
Support	$x \in \mathbb{R}$
PDF	$\dfrac{1}{\sqrt{2\pi\sigma^2}}e^{-(x-\mu)^2/2\sigma^2}$
CDF	$\Phi\left(\dfrac{x-\mu}{\sigma}\right)$
Mean	μ
Variance	σ^2

The Normal distribution The Normal distribution is so widely used in data science and machine learning that it's worth mentioning some extra properties here.

- If $X \sim \text{Normal}(\mu,\sigma^2)$ and a and b are constants, then $aX + b \sim \text{Normal}(a\mu + b, a^2\sigma^2)$. Section 3.3 page 41 describes tools for reasoning about this and other transformations of a random variable.
- An engineer's rule of thumb: an arbitrary random variable can be approximated reasonably well by a Normal. This is so useful and simple that it can't possibly always be true—but what's remarkable is that it's often nearly true. In Part III we'll see the Central Limit Theorem, which is circumstantial evidence for the rule of thumb.

1.4.2. SPECIFYING NUMERICAL RANDOM VARIABLES

Any random variable can be specified by giving its *probability distribution*, i.e. by giving the value of

$$\mathbb{P}(X \in A)$$

for every set $A \subseteq \mathcal{S}$. Numerical random variables (either continuous or discrete and integer-valued) can alternatively be specified by giving the *cumulative distribution function*, often shortened to *distribution function* or even *distribution*,

$$F(x) = \mathbb{P}(X \leq x).$$

If X is a discrete random variable (assumed to take integer values, unless we explicitly say otherwise), then

$$F(x) = \mathbb{P}(X \leq x) = \sum_{y=-\infty}^{x} \mathbb{P}(X = y)$$

and F is a step function, increasing at every y where $\mathbb{P}(X = y) > 0$.

If on the other hand $F(x)$ is differentiable, then X is said to be a continuous random variable. Its *density function* is defined to be $f(x) = F'(x)$, and

$$F(x) = \mathbb{P}(X \leq x) = \int_{y=-\infty}^{x} f(y)\, dy\,.$$

For a continuous random variable, $\mathbb{P}(X = x) = 0$ for every x, and so these four expressions are all equal:

$$\left.\begin{matrix} \mathbb{P}(a \leq X \leq b) \\ \mathbb{P}(a < X \leq b) \\ \mathbb{P}(a \leq X < b) \\ \mathbb{P}(a < X < b) \end{matrix}\right\} = \int_{x=a}^{b} f(x)\, dx\,.$$

Unified density notation. In this course we'll write $\text{Pr}_X(x)$, and refer to it as "density", for both discrete and continuous random variables:

$$\text{Pr}_X(x) = \begin{cases} \mathbb{P}(X = x) & \text{when } X \text{ is a discrete random variable} \\ & \text{(}\text{Pr}_X \text{ is called the } \textit{probability mass function}\text{)} \\ f(x) & \text{when } X \text{ is a continuous random variable with density } f \\ & \text{(}\text{Pr}_X \text{ is called the } \textit{probability density function}\text{)} \end{cases}$$

For both discrete and continuous random variables, for any[4] set A,

$$\mathbb{P}(X \in A) = \begin{cases} \sum_{x \in A} \text{Pr}_X(x) & \text{if } X \text{ is discrete} \\ \int_{x \in A} \text{Pr}_X(x)\, dx & \text{if } X \text{ is continuous.} \end{cases}$$

Many results in data science and machine learning apply to both discrete and continuous random variables, but with slightly different meanings, and this notation helps us write formulae that work for both cases (and we'll typically only write one version of the formula, and say "use sums or integrals as appropriate").

[4]Any reasonable set A. For a discrete random variable, A should be a set of integers. For a continuous random variable, A should be a finite collection of open or closed intervals. If we let A be the set of all rational numbers, for example, then integration is problematic!

1.4.3. INDEPENDENCE

Discrete. For a pair of discrete random variables X and Y, their *joint density* is

$$\Pr_{X,Y}(x,y) = \mathbb{P}(X = x \text{ and } Y = y).$$

In section 1.3 we gave a definition of independence in terms of the joint distribution: we said X and Y are independent if

$$\mathbb{P}(X \in A \text{ and } Y \in B) = \mathbb{P}(X \in A)\,\mathbb{P}(Y \in B) \quad \text{for all } A \text{ and } B.$$

It's not hard to show that

$$X, Y \text{ independent} \iff \Pr_{X,Y}(x,y) = \Pr_X(x)\Pr_Y(y).$$

The $\Rightarrow$ direction is trivial, and the $\Leftarrow$ direction isn't hard.

Continuous. For continuous random variables X and Y, or when one is discrete and the other continuous, it's trickier to define the joint density. A formal definition is left to section 3.4. The set-based definition from section 1.3 is the technically correct way to define independence, but for most practical purposes it's best to use the density-based formula, $\Pr_{X,Y}(x,y) = \Pr_X(x)\Pr_Y(y)$.

Random samples. A collection of random variables $(X_1, \dots, X_n)$ are independent if

$$\Pr(x_1, \dots, x_n) = \Pr_{X_1}(x_1) \times \cdots \times \Pr_{X_n}(x_n).$$

This is especially useful when we're using maximum likelihood estimation to estimate an unknown parameter, and our dataset is a random sample, as we'll see in the next section.

1.5. Learning generative models

tl;dr. In generative modelling, we have a dataset $x_1, \ldots, x_n$, and we want to find a distribution that might have generated it.

1. First, choose a distribution with one or more tunable parameters. Call the corresponding random variable X, and write its distribution as $\Pr_X(x \mid \theta)$ where θ is the parameter (which may be a real number, an integer, or a tuple). We want to view the dataset $x_1, \ldots, x_n$ as independent samples generated from X.
2. Next, write out the likelihood of θ given the dataset. This is just the probability of observing the dataset, which by independence is
$$\operatorname{lik}(\theta \mid x_1, \ldots, x_n) = \Pr_X(x_1 \mid \theta) \times \cdots \times \Pr_X(x_n \mid \theta).$$
3. Estimate θ using maximum likelihood estimation, i.e. by solving
$$\hat{\theta} = \arg\max_{\theta} \log \operatorname{lik}(\theta \mid x_1, \ldots, x_n).$$
This is called *fitting the model*.

In machine learning, there are many tools for finding patterns in a dataset that doesn't have pre-existing labels; these tools come under the heading "unsupervised learning". Generative modelling falls into this category: there are no labels attached to the datapoints, they're just considered to be identically distributed samples from the same distribution.

Unsupervised learning also includes all sorts of descriptive tools, with an emphasis on clustering algorithms (though note that generative models can also be used to find clusters, as in example 1.9 below.) The advantage of generative modelling over descriptive algorithms is that it comes with a general-purpose principled way to evaluate how good a model is, namely the likelihood function, whereas descriptive algorithms are more ad hoc. Modern approaches to unsupervised learning such as VAE (variational auto-encoders) and GANs (generative adversarial networks) are based on generative modelling.

Exercise 1.7 (Coin tosses).
Suppose we take a biased coin and toss it $n = 10$ times, and observe the outcomes
$$(x_1, \ldots, x_{10}) = (H, H, t, t, H, H, t, t, H, H).$$
Fit the probability model
$$X = \begin{cases} H & \text{with probability } p \\ t & \text{with probability } 1-p. \end{cases}$$

The density of X is
$$\Pr_X(x \mid p) = \begin{cases} p & \text{if } x = H \\ 1-p & \text{if } x = t \end{cases}$$
so the likelihood is
$$\begin{aligned}\operatorname{lik}(p \mid x_1, \ldots, x_n) &= \prod_{i=1}^{n} \begin{cases} p & \text{if } x_i = H \\ 1-p & \text{if } x_i = t \end{cases} \\ &= p^y (1-p)^{n-y} \quad \text{where } y = 6 \text{ is the number of heads}\end{aligned}$$
and the log likelihood is
$$\log \operatorname{lik}(p \mid x_1, \ldots, x_n) = y \log p + (n-y)\log(1-p).$$
Maximizing this with respect to p is exactly what we did in exercise 1.1 on page 2, and the answer is $\hat{p} = y/n = 0.6$.

∎

Exercise 1.8 (Fitting a Normal distribution).
Let the dataset be $x_1, \dots, x_n$. Fit a $\text{Normal}(\mu, \sigma^2)$ distribution, where μ and σ are unknown.

The Normal distribution: page 63 in the appendix

If $X \sim \text{Normal}(\mu, \sigma^2)$ then X is a continuous random variable with probability density function

$$\Pr_X(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$$

where $-\infty < \mu < \infty$ and $0 < \sigma < \infty$. The log likelihood function given a dataset $x_1, \dots, x_n$ is

$$\begin{aligned}\log \text{lik}(\mu, \sigma \mid x_1, \dots, x_n) &= \sum_{i=1}^{n} \log \Pr_X(x_i \mid \mu, \sigma) \\ &= \sum_i \Big(-\frac{1}{2}\log(2\pi\sigma^2) - \frac{(x_i-\mu)^2}{2\sigma^2} \\ &= -\frac{n}{2}\log(2\pi) - n\log\sigma - \frac{\sum_i (x_i-\mu)^2}{2\sigma^2}.\end{aligned}$$

To find the maximum likelihood estimator, differentiate with respect to μ and σ and find where the derivative is equal to zero. There are two parameters, so we have a pair of simultaneous equations to solve:

$$\begin{aligned}\frac{\partial}{\partial\mu}\log \text{lik} &= -\frac{\sum_i 2(x_i-\mu)}{2\sigma^2} = 0 \\ \frac{\partial}{\partial\sigma}\log \text{lik} &= -\frac{n}{\sigma} + \frac{\sum_i (x_i-\mu)^2}{\sigma^3} = 0.\end{aligned}$$

The solution is

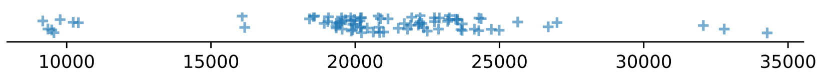
$$\hat\mu = \frac{\sum_i x_i}{n}, \qquad \hat\sigma = \sqrt{\frac{1}{n}\sum_i (x_i - \hat\mu)^2}.$$

∎

Exercise 1.9 (Gaussian mixture model).
The galaxies dataset[5] consists of velocities in km/sec of 82 galaxies in the Corona Borealis region. Clusters in such a dataset is evidence for voids and superclusters in the far universe.

```python
galaxies = [
  9172, 9350, 9483, 9558, 9775, 10227, 10406, 16084, 16170, 18419, 18552, 18600, 18927,
  19052, 19070, 19330, 19343, 19349, 19440, 19473, 19529, 19541, 19547, 19663, 19846,
  19856, 19863, 19914, 19918, 19973, 19989, 20166, 20175, 20179, 20196, 20215, 20221,
  20415, 20629, 20795, 20821, 20846, 20875, 20986, 21137, 21492, 21701, 21814, 21921,
  21960, 22185, 22209, 22242, 22249, 22314, 22374, 22495, 22746, 22747, 22888, 22914,
  23206, 23241, 23263, 23484, 23538, 23542, 23666, 23706, 23711, 24129, 24285, 24289,
  24366, 24717, 24990, 25633, 26690, 26995, 32065, 32789, 34279]
```

```python
# A rug plot, with the y coordinates jittered to see more clearly
fig,ax = plt.subplots(figsize=(8,.3))
jitter_y = numpy.random.uniform(low=-1, high=1, size=len(galaxies))
ax.scatter(galaxies, jitter_y, marker='+', alpha=.6)
for f in ['left','top','right']: ax.spines[f].set_visible(False)
ax.set_ylim([-2,1.3])
ax.set_yticks([])
plt.show()
```



Fit a Gaussian mixture model with three clusters. In other words, fit the random variable

$$X = \begin{cases} \text{Normal}(\mu_1, \sigma_1^2) & \text{with probability } p_1 \\ \text{Normal}(\mu_2, \sigma_2^2) & \text{with probability } p_2 \\ \text{Normal}(\mu_3, \sigma_3^2) & \text{with probability } p_3 \end{cases}$$

where $p_1 + p_2 + p3 = 1$, and all nine parameters are unknown.

According to an exercise on the example sheet, the density is

$$\Pr_X(x) = p_1 \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-(x-\mu_1)^2/2\sigma_1^2} + p_2 \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-(x-\mu_2)^2/2\sigma_2^2} + p_3 \frac{1}{\sqrt{2\pi\sigma_3^2}} e^{-(x-\mu_3)^2/2\sigma_3^2}$$

We'll use numerical optimization. First, as described in section 1.2, we'll rewrite in terms of unconstrained parameters. In this case, the constraints are $p_i \in [0,1]$, $p_1 + p_2 + p_3 = 1$, $\sigma_i > 0$, so let's optimize the parameter

$$\theta = (q_1, q_2,\ \mu_1, \mu_2, \mu_3,\ \tau_1, \tau_2, \tau_3) \in \mathbb{R}^8$$

transformed into the parameters we want by

$$p_1 = \frac{e^{q_1}}{e^{q_1} + e^{q_2} + 1}, \qquad p_2 = \frac{e^{q_2}}{e^{q_1} + e^{q_2} + 1}, \qquad p_3 = \frac{1}{e^{q_1} + e^{q_2} + 1}$$
$$\sigma_1 = e^{\tau_1}, \qquad \sigma_2 = e^{\tau_2}, \qquad \sigma_3 = e^{\tau_3}.$$

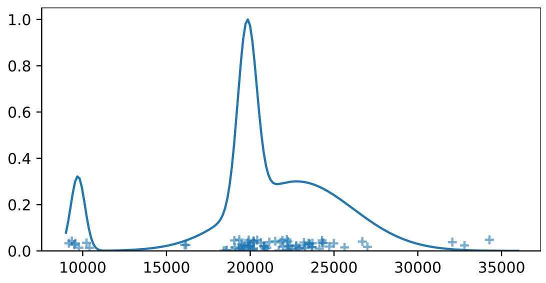
We can now define the likelihood function, and maximize the log likelihood.

```
import scipy.stats
Pr = scipy.stats.norm.pdf   # density of the Normal distribution

def loglik(θ, x): # x is a vector of observations; returns a vector of loglik
    q₁,q₂, μ₁,μ₂,μ₃, τ₁,τ₂,τ₃ = θ
    p = numpy.exp([q₁,q₂,1])
    [p₁,p₂,p₃] = p / sum(p)
    σ₁,σ₂,σ₃ = numpy.exp([τ₁,τ₂,τ₃])
    lik = p₁*Pr(x,loc=μ₁,scale=σ₁) + p₂*Pr(x,loc=μ₂,scale=σ₂) + p₃*Pr(x,loc=μ₃,scale=σ₃)
    return numpy.log(lik)

# Initial guess inspired by the rug plot in the question
initial_guess = [0,0, 10000,20000,24000, math.log(1000),math.log(5000),math.log(8000)]
θ̂ = scipy.optimize.fmin(lambda θ: -numpy.sum(loglik(θ,galaxies)),
                        initial_guess, maxiter=5000)

# Plot the fitted density, with a rug plot of the actual data
fig,ax = plt.subplots()
x = numpy.linspace(9000,36000,200)
f = numpy.exp(loglik(θ̂, x))
ax.plot(x, f/numpy.max(f))  # we don't really care what the y-axis is
jitter_y = numpy.random.uniform(low=0, high=0.05, size=len(galaxies))
ax.scatter(galaxies, jitter_y, marker='+', alpha=.6)
ax.set_ylim(ymin=0)
plt.show()
```



∎

[5]W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S (MASS)*. galaxies dataset. New York: Springer, 2002. Source: M. Postman, J. P. Huchra, and M. J. Geller. "Probes of large-scale structures in the Corona Borealis region". In: *Astronomical Journal* (1986)

1.6. Supervised learning

tl;dr. Consider a dataset in which each datapoint is a tuple of values. Think of it as a spreadsheet or database table: each row/record is a datapoint, and the columns are fields in the tuple. Often we want to understand how one item in the tuple depends on the others. The item we want to understand is called the *response variable* or *label* and the others are called *covariates* or *predictors*. Write y_i for the label for record $i \in \{1, \dots, n\}$, and write x_i for the predictor variable or variables.

The goal is to find a probability model which, given the predictors, might have generated the labels. This is called *probabilistic supervised learning* or *regression modelling*.

1. First, choose a probabilistic model for the label, where the distribution depends on one or more unknown parameters as well as on the predictors. We want to view the y_i as independent samples, y_i sampled from a random variable Y_i with density $\Pr(y \mid \theta, x_i)$, where θ is the unknown parameter.
2. Next, write out the likelihood of θ given the dataset. This is just the probability of observing the dataset, which by independence is
$$\text{lik}(\theta \mid y_1, \dots, y_n) = \Pr(y_1 \mid \theta, x_1) \times \cdots \times \Pr(y_n \mid \theta, x_n).$$
3. Estimate θ using maximum likelihood estimation, i.e. by solving
$$\hat{\theta} = \operatorname*{arg\,max}_{\theta} \log \text{lik}(\theta \mid y_1, \dots, y_n).$$
This is called *fitting the model*.

This is all there is to much of machine learning, especially Kaggle-style competitions—the art is inventing models that fit the data well, and for which the parameters give insight. In section 2.4 we'll study linear models, a flexible and interpretable class of regression models based on the Normal distribution.

Exercise 1.10 (Simple linear regression).
Given a labelled dataset $[(y_1, x_1), \dots, (y_n, x_n)]$ consisting of pairs of real numbers, fit the model
$$Y_i \sim \text{Normal}\big(a + bx_i, \sigma^2\big)$$
which we could alternatively write as
$$Y_i = a + bx_i + \text{Normal}(0, \sigma^2),$$
where σ is given, and a and b are parameters to be estimated.

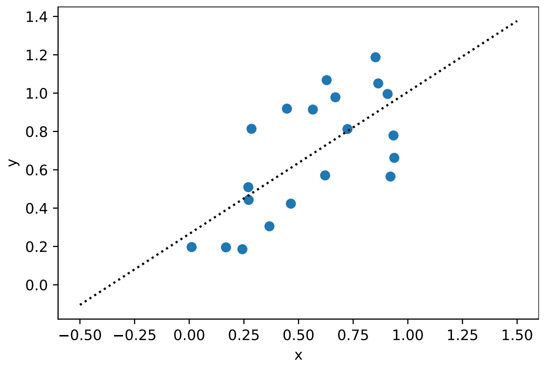
```
import scipy.stats
Pr = scipy.stats.norm.pdf

def loglik(θ, x, y):
    a,b = θ
    lik = Pr(y, loc=a+b*x, scale=σ) # σ is a constant
    return numpy.log(lik)

initial_guess = [0,1]
â,b̂ = scipy.optimize.fmin(lambda θ: -numpy.sum(loglik(θ,x,y)),
                          initial_guess, maxiter=5000)

# Plot the line y = â + b̂x, and superimpose the dataset
fig,ax = plt.subplots(figsize=(6,4))
xnew = numpy.linspace(-.5,1.5,100)
```

```
ax.plot(xnew, ahat+bhat*xnew, linestyle='dotted', color='black')
ax.scatter(x, y)
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```



∎

Alternatively, in this case, the equations are simple enough that we can solve them with maths rather than computation.

The density of Y_i is

$$\Pr(y_i \mid a, b, x_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y_i - a - bx_i)^2/2\sigma^2}$$

so the log likelihood of the dataset is

$$\begin{aligned}\log \operatorname{lik}(a, b \mid y_1, \dots, y_n) &= \sum_{i=1}^n \log \Pr(y_i \mid a, b, x_i) \\ &= \sum_{i=1}^n \Big\{ -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(y_i - a - bx_i)^2}{2\sigma^2} \Big\} \\ &= -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^n (y_i - a - bx_i)^2.\end{aligned}$$

To find the maximum likelihood estimator, differentiate with respect to a and b and find where the derivative is equal to zero. There are two parameters, so we have a pair of simultaneous equations to solve:

$$\begin{aligned}\frac{\partial}{\partial a}\log \operatorname{lik} &= \frac{1}{\sigma^2}\sum_{i=1}^n (y_i - a - bx_i) = 0 \\ \frac{\partial}{\partial b}\log \operatorname{lik} &= \frac{1}{\sigma^2}\sum_{i=1}^n (y_i - a - bx_i)x_i = 0.\end{aligned}$$

The solution is

$$\hat{b} = \frac{n\bar{x}\bar{y} - \sum_i x_i y_i}{n\bar{x}^2 - \sum_i x_i^2}, \qquad \hat{a} = \bar{y} - \hat{b}\bar{x}$$

where $\bar{x} = \sum_i x_i/n$ and $\bar{y} = \sum_i y_i/n$.

∎

Exercise 1.11 (Binomial regression).

The UK Home Office makes available several datasets of police records, at `data.police.uk`. The stop-and-search dataset has been preprocessed to list the number of stops and the number of those that led to the police finding something suspicious, for each police force and each year.

police_force	year	stops	find
bedfordshire	2017	786	231
cambridgeshire	2016	1691	621
cambridgeshire	2017	581	264

Fit the model $Y_i \sim \text{Binom}(x_i, p)$ where Y_i is the number of ‘find’ incidents in a given police force and year, x_i is the number of stops, and p is the parameter to estimate.

Binom is for the Binomial random variable: appendix page 62

The binomial distribution is a discrete random variable commonly used for counting the number of successes in a sequence of yes-no trials. If $X \sim \text{Binom}(n,p)$ then n is the number of trials, $0 \le p \le 1$ is the success probability, and the probability mass function is

$$\text{Pr}_X(r \mid n,p) = \binom{n}{r} p^r (1-p)^{n-r}, \quad r \in \{0,\dots,n\}.$$

We'll assume the records in the dataset are independent, since we're not told otherwise. In maths notation,

$$\text{Pr}(y_1,\dots,y_n \mid p) = \prod_{i=1}^{n} \binom{x_i}{y_i} p^{y_i} (1-p)^{x_i - y_i}.$$

(Covariates are fixed and known so we're treating them as constants in this equation, not as parameters.) The log likelihood is

$$\begin{aligned}\log \text{lik}(p \mid y_1,\dots,y_n) &= \sum_i \Bigl(\log \binom{x_i}{y_i} + y_i \log p + (x_i - y_i)\log(1-p) \Bigr) \\ &= \kappa + \Bigl(\sum_i y_i\Bigr) \log p + \Bigl(\sum_i x_i - \sum_i y_i\Bigr) \log(1-p)\end{aligned}$$

where κ is a constant i.e. doesn't depend on p. The maximum likelihood estimator for p solves

$$\frac{d}{dp} \log \text{lik}(p \mid y_1,\dots,y_n) = 0$$

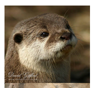
and the solution is

$$\hat{p} = \frac{\sum_i y_i}{\sum_i x_i}.$$

∎

This is not a very interesting model. The only slightly non-obvious thing it's told us is "don't estimate p separately for each police force and year, then average these estimates; instead estimate p from the whole aggregated data". The modelling exercise becomes much more interesting when we use it to investigate the influence of multiple covariates, e.g. how gender and race interact.

Example 1.12 (Neural network classification).
The ImageNet dataset consists of over 14 million images, each hand-annotated with a label. Here are some sample rows:

image	category
	"otter"
	"otter"
	"cello"

Suppose we've built a black-box function

$$f(\theta, x) = (s_1,\dots,s_K) \in \mathbb{R}^K$$

which takes in a vector of parameters θ, and an image x expressed as a vector of pixels, and which returns a list of scores s_k, one score for each category $k \in \{1,\dots,K\}$ where K is the number of different categories. These scores are real values, perhaps not in the range $[0,1]$, so we can't use them directly in a probability model. Here's a handy trick:[6] define probabilities

$$p_k = p_k(\theta, x) = \frac{e^{s_k}}{e^{s_1} + \cdots + e^{s_K}},$$

and consider the probability model

$$\mathbb{P}(Y_i = y \mid \theta, x_i) = p_y, \quad y \in \{1,\dots,K\}.$$

The log likelihood is

$$\log \operatorname{lik}(\theta \mid y_1, \ldots, y_n) = \sum_{i=1}^{n} \log p_{y_i}(\theta, x_i).$$

To fit the model, we pick θ to maximize the log likelihood.

In deep learning, we use a function f implemented as a neural network (a convolutional neural network, for image classification) and θ is the vector of the network's connection weights. Finding the maximum likelihood estimator for θ is referred to as "training the neural network".

[6]This transformation from scores to probabilities is just an algebraic gimmick, a trick to map a vector in $\mathbb{R}^K$ to a probability vector, and it doesn't have any deeper meaning. It's called *softmax* in machine learning and *multinomial logit* in statistics.

1.7. Supervised learning and prediction loss *

Any number of deep learning tutorials refer to prediction and loss functions, but might not even mention probability modelling at all. Yet section 1.6 explained supervised learning entirely in terms of probability modelling and in particular maximum likelihood estimation. What's the connection?

Let's look more closely at the log likelihood that we seek to maximize. In all the three examples from the last section, the goal "maximize the log likelihood of the unknown parameters given the dataset" can, after some algebraic wrangling, be rewritten as

$$\text{minimize} \quad \sum_{i=1}^{n} \mathsf{Loss}\big(\mathsf{prep}(y_i), \mathsf{pred}_\theta(x_i)\big) \quad \text{over} \quad \theta$$

for appropriate functions Loss, prep, and pred_θ. The interpretation is that $\mathsf{pred}_\theta(x)$ is the prediction of our model when given input x, $\mathsf{prep}(y)$ is some housekeeping needed to put the label y into a form suitable for prediction, and $\mathsf{Loss}(\mathsf{prep}(y), p)$ is the penalty when the true label is y and we predicted p.

The loss-function interpretation doesn't involve any explicit probability modelling. Someone who doesn't believe in probability theory could perfectly well formulate a task as a problem of minimizing prediction loss; they might even claim that deep learning is entirely about prediction and loss functions, and doesn't need any modelling at all. However, you're much better off starting with a probability model:

- Without a probability model, different loss functions are just formulae that you have to memorize. With a probability model, you still have to design a model, but the loss functions don't look like a laundry list of mystery.
- If you face a new type of dataset, it's fairly intuitive to design a probability model for it, perhaps in the form of simulation code. You can then derive the corresponding loss function, and since it comes from your intuitive probability model, it should be well-behaved. On the other hand, if you only think in terms of prediction loss, you might design a loss function that makes the learning go haywire. Arguably, any sane loss function has a corresponding probability model.
- There are some probability models which don't have a natural interpretation as minimizing prediction loss. If you start with probability modelling, you allow yourself a wider class of models.
- Unsupervised learning (in the form of generative modelling, section 1.5) and supervised learning are almost exactly the same thing, if we think in terms of probability models. If we think in terms of prediction loss, it's hard to even formulate what unsupervised learning is meant to achieve.

Algebraic wrangling. Here are the details of how to turn "maximize log likelihood" into "minimize prediction loss", for two of the three problems. For exercise 1.10, we sought

$$\operatorname*{arg\,max}_{a,b} \Big\{ -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - a - bx_i)^2 \Big\}$$

which is equivalent to

$$\operatorname*{arg\,min}_{a,b} \sum_{i=1} L(y_i, a + bx_i) \quad \text{where} \quad L(y,x) = (y-x)^2.$$

In example 1.12 we sought

$$\operatorname*{arg\,max}_{\theta} \sum_{i=1}^{n} \log p_{y_i}(\theta, x_i) = \operatorname*{arg\,max}_{\theta} \sum_{i=1}^{n}\sum_{k=1}^{K} 1_{y_i=k} \log p_k(\theta, x_i).$$

The last expression involves $1_{\{\cdot\}}$ which denotes the *indicator function*, $1_{\text{true}} = 1$ and $1_{\text{false}} = 0$. This is an algebraic trick to say "only keep the p_k term where $y_i = k$". Equivalently, we sought

$$\operatorname*{arg\,min}_{\theta} \sum_{i=1}^{n} L\big(\text{onehot}(y_i), \text{softmax}(f(\theta, x_i))\big)$$

$$\text{where} \quad L(q,p) = -\big(q_1 \log p_1 + \cdots + q_K \log p_K\big)$$

and

$$\text{onehot}(y) = \big(1_{y=1}, \dots, 1_{y=K}\big), \quad \text{softmax}(s) = \Big(\frac{e^{s_1}}{e^{s_1} + \cdots + e^{s_K}}, \dots, \frac{e^{s_K}}{e^{s_1} + \cdots + e^{s_K}}\Big)$$

In machine learning, this L is referred to as "cross-entropy loss".

2. Feature spaces / linear regression

In data science, a *feature* is any measurable property of the objects being studied. A *linear model* is a model with unknown parameters in which the parameters are weighted by features and combined linearly.

Section 2.1 starts with a very simple linear model example, to flesh out the uselessly abstract definitions above, and to show how to implement linear models in Python. In the following sections we'll elaborate:

- Linear models are expressive and interpretable, and we can use them to ask all sorts of questions about a dataset by choosing appropriate features. We'll look at examples in section 2.2. Linear models should be your go-to models for all sorts of data science and machine learning problems, the second thing you try (after simple tabulations) to get a sense of the data you're working with.

- A simple way to estimate the parameters is using *least squares estimation*. There are fast algorithms for doing this, which come from the mathematics of linear algebra. The mathematics also gives insight into how linear models work, especially questions of parameter identifiability. Section 2.3 contains a review of the relevant linear algebra.

- There is a probabilistic interpretation of least squares estimation: it is maximum likelihood estimation, for a supervised-learning probability model in which the labels have a Gaussian distribution. Section 2.4 describes this link.

  Because linear modelling comes from a probability model, we can use all sorts of probability-based inference techniques (which will be studied in Part III of this course) to compute confidence intervals etc.

- Linear models are the building block for many other machine learning techniques such as logistic regression and deep neural networks (from the Part II course "Data science principles and practice") and support vector machines and perceptrons (from the Part II course "Machine learning and Bayesian inference").

2.1. Fitting a linear model

tl;dr. A *linear model* can be written as

$$y = \beta_1 e_1 + \cdots + \beta_K e_K + \varepsilon$$

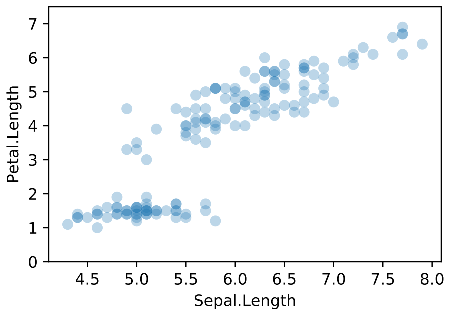
where $y = (y_1, y_2, \dots)$ is the vector of responses with y_i the value for record i in the dataset, $e_1, \dots, e_K$ are feature vectors with $e_k = (e_{k,1}, e_{k,2}, \dots)$ where $e_{k,i}$ is the value of the kth feature for record i, β_k is the parameter that weights the kth feature, and $\varepsilon = (\varepsilon_1, \varepsilon_2, \dots)$ is a vector of *residuals*, also called error or noise.

Least squares estimation means picking the parameters β to minimize the *mean square error* $\sum_i \varepsilon_i^2/n$. Use sklearn.linear_model.LinearRegression() to do this.

Example 2.1.
The Iris dataset was collected by the botanist Edgar Anderson and popularized[7] by Ronald Fisher in 1936. Fisher has been described as a "genius who almost single-handedly created the foundations for modern statistical science". The dataset consists of 50 samples from each of three species of iris, each with four measurements. The full dataset is `https://teachingfiles.blob.core.windows.net/datasets/iris.csv`.

Petal length	Petal width	Sepal length	Sepal width	species
1.0	0.2	4.6	3.6	setosa
5.0	1.9	6.3	2.5	virginica
5.8	1.6	7.2	3.0	virginica
1.7	0.5	5.1	3.3	setosa
4.2	1.2	5.7	3.0	versicolor
...				

Let's investigate how petal length depends on sepal length. Here is a plot:



It suggests a curve. Let's fit a quadratic curve, using the linear model

$$\text{Petal.Length} \approx \alpha + \beta\,\text{Sepal.Length} + \gamma\,(\text{Sepal.Length})^2. \qquad (1)$$

Linear does NOT mean 'straight line'. It refers to linear algebra—adding vectors, and multiplying vectors by scalars. In vector form, the model says

$$\begin{bmatrix} \text{Petal.Length}_1 \\ \text{Petal.Length}_2 \\ \vdots \end{bmatrix} \approx \alpha \begin{bmatrix} 1 \\ 1 \\ \vdots \end{bmatrix} + \beta \begin{bmatrix} \text{Sepal.Length}_1 \\ \text{Sepal.Length}_2 \\ \vdots \end{bmatrix} + \gamma \begin{bmatrix} (\text{Sepal.Length}_1)^2 \\ (\text{Sepal.Length}_2)^2 \\ \vdots \end{bmatrix}.$$

[7] It's tempting for computer scientists and mathematicians to think that data science is about algorithms and calculating with distributions and so on, but shared datasets are arguably more important. C.P. Scott, the former editor of *The Guardian*, said "Comment is free, but facts are sacred".

Modern advances in neural networks and deep learning were propelled by two shared datasets: the MNIST database of handwritten digits, and the ImageNet database of labelled photos. The story of ImageNet and of Fei-Fei Li, the researcher who collected it, is told in *The data that transformed AI research—and possibly the world*, `https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/`.

In addition to shared datasets, it's also useful to have a shared challenge, what David Donoho calls a *common task framework*. See David Donoho. *50 years of Data Science*. Presentation at the Tukey centennial workshop. 2015. URL: `http://courses.csail.mit.edu/18.337/2015/docs/50YearsDataScience.pdf`

In scientific computing, the coding style is also in terms of vectors:

```
iris = pandas.read_csv('https://teachingfiles.blob.core.windows.net/datasets/iris.csv')

# A linear model with three feature vectors [one, x, x**2] and the response vector y
one, x, y = numpy.ones(len(iris)), iris['Sepal.Length'], iris['Petal.Length']
model = sklearn.linear_model.LinearRegression(fit_intercept=False)
model.fit(numpy.column_stack([one, x, x**2]), y)
(α,β,γ) = model.coef_
```

```
(-17.447,  5.392,  -0.296)
```

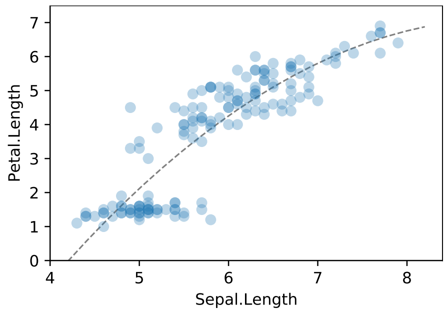
In fact the sklearn model fitting function always includes a one vector, unless we explicitly tell it otherwise with fit_intercept=False. Another way to write this code is

```
model2 = sklearn.linear_model.LinearRegression()
model2.fit(numpy.column_stack([x, x**2]), y)
α,(β,γ) = model2.intercept_, model2.coef_
```

What does this fit look like? We could explicitly evaluate $\alpha + \beta x + \gamma x^2$ for a range of x values and plot. Or use model.predict(), to relieve us from re-typing the model formula.

```
newx = numpy.linspace(4.2, 8.2, 20)
predy = model2.predict(numpy.column_stack([newx, newx**2]))

fig,ax = plt.subplots(figsize=(4.5,3))
ax.plot(newx, predy, color='0.5', zorder=-1, linewidth=1, linestyle='dashed')
ax.scatter(iris['Sepal.Length'], iris['Petal.Length'], alpha=.3)
ax.set_ylim(0,7.5)
ax.set_ylabel('Petal.Length')
ax.set_xlabel('Sepal.Length')
plt.show()
```



Terminology. We'd describe model (1) as having two features, Sepal.Length, and (Sepal.Length)^2. The rows in this dataset have other attributes, and they can be transformed to create an infinite variety of features, but we'll only use the word *feature* for data attributes that are being used in a model. We call Petal.Length the *response* or *label* in this model, not a feature.

Why two features, and not one, or three? From the perspective of the person preparing the dataset, there is only one feature, Sepal.Length. From the perspective of the person computing α, β, and γ, there are two data features that have to be accounted for, and it's irrelevant that they came from the same column in the dataset. From the perspective of a stickler for definitions, the definition of 'linear model' says that parameters are weighted by features, so there is really a third feature, the constant feature one with parameter α. Don't get uptight about defining the word 'feature', just write out your models explicitly, and there will be no confusion.

* ✻ *

The model is linear because it combines the unknown parameters α, β and γ in a linear formula. There's no reason to think this is in any way a 'true' model, and we could equally well have proposed a non-linear model e.g.

$$\text{Petal.Length} \approx \alpha - \beta e^{-\gamma \text{Sepal.Length}}.$$

Linear models are just easier to work with, so they're a better place to start.

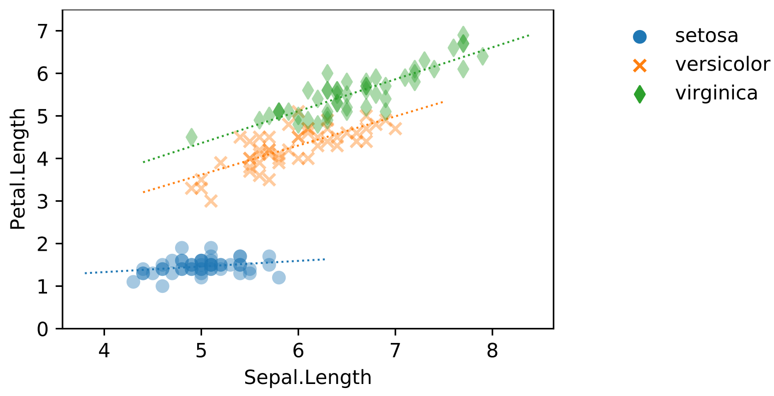
2.2. Features

Here is a gallery of cunning ways to use features to ask questions about a dataset.

2.2.1. ONE-HOT CODING

One-hot coding is used to turn an enum feature (also called *categorical* or *factor*) into a collection of binary features, so it can be used in a linear model. Here's an example.

The Iris data is made up of three species. Maybe there's a straight-line fit between petal length and sepal length, but with different slopes and intercepts for each species.



One way to write this is

$$\text{Petal.Length} \approx \alpha_{\text{species}} + \beta_{\text{species}}\text{Sepal.Length}.$$

Here's the same equation, but written as vectors, and abbreviating Petal.Length as PL, and Sepal.Length as SL:

$$\begin{array}{l} \text{seto} \\ \text{virg} \\ \text{virg} \\ \text{seto} \\ \text{vers} \\ \\ \end{array} \begin{bmatrix} \text{PL}_1 \\ \text{PL}_2 \\ \text{PL}_3 \\ \text{PL}_4 \\ \text{PL}_5 \\ \vdots \end{bmatrix} \approx \alpha_{\text{seto}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix} + \alpha_{\text{virg}} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} + \alpha_{\text{vers}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ \vdots \end{bmatrix} + \beta_{\text{seto}} \begin{bmatrix} \text{SL}_1 \\ 0 \\ 0 \\ \text{SL}_4 \\ 0 \\ \vdots \end{bmatrix} + \beta_{\text{virg}} \begin{bmatrix} 0 \\ \text{SL}_2 \\ \text{SL}_3 \\ 0 \\ 0 \\ \vdots \end{bmatrix} + \beta_{\text{vers}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \text{SL}_5 \\ \vdots \end{bmatrix}$$

Or writing symbols for the vectors:

$$\begin{aligned} \text{PL} \approx\ & \alpha_{\text{seto}}\mathsf{s}_{\text{seto}} + \alpha_{\text{virg}}\mathsf{s}_{\text{virg}} + \alpha_{\text{vers}}\mathsf{s}_{\text{vers}} \\ & + \beta_{\text{seto}}(\mathsf{s}_{\text{seto}} \otimes \text{SL}) + \beta_{\text{virg}}(\mathsf{s}_{\text{virg}} \otimes \text{SL}) + \beta_{\text{vers}}(\mathsf{s}_{\text{vers}} \otimes \text{SL}) \end{aligned}$$

In this equation, each s_k is a binary vector marking out which rows belong to species k, for example $\mathsf{s}_{\text{seto}} = 1[\text{Species=setosa}]$. This is called *one-hot coding* of the Species vector. Also, $\otimes$ means elementwise multiplication.

1_x also written $1[x]$ is the indicator function, $1_{\text{true}} = 1$ and $1_{\text{false}} = 0$

```
species_levels = numpy.unique(iris['Species'])
x, y = iris['Sepal.Length'], iris['Petal.Length']
s1,s2,s3 = (iris['Species']==s for s in species_levels)
model = sklearn.linear_model.LinearRegression(fit_intercept=False)
model.fit(numpy.column_stack([s1,s2,s3,s1*x,s2*x,s3*x]), y)
```

We've seen one-hot coding before, in section 1.7, where we used it to encode labels in an image classification task.

Notation warning. The model equation

$$\text{Petal.Length} \approx \alpha_{\text{species}} + \beta_{\text{species}}\text{Sepal.Length}$$

should be interpreted as a vector equation. In Python, the α_{species} vector comes from

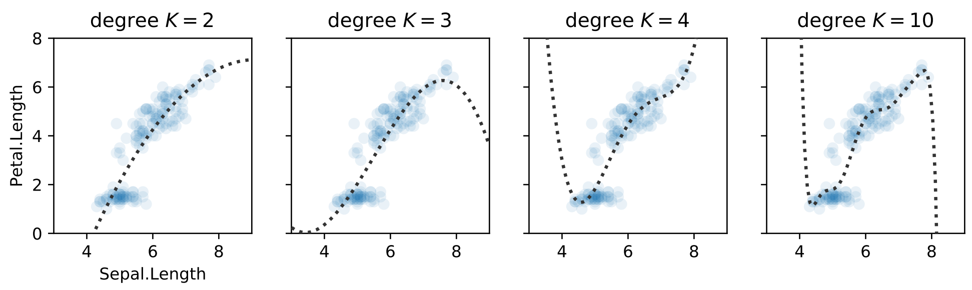
```
α = {'seto': ..., 'virg': ..., 'vers': ...}        # a dictionary, one key per species
species = ['seto', 'virg', 'virg', 'seto', vers', ...]  # a list , one entry per datapoint
α_species = [α[s] for s in species] # a list , one entry per datapoint
```

It's also possible to treat the equation as a 'scientific' equation, referring to a hypothetical individual. The neat (or confusing) thing about this notation is that we write the same equation whether we're working on a dataset or reporting our findings.

2.2.2. NON-LINEAR RESPONSE

We've already seen that we can use the quadratic feature $(\text{Sepal.Length})^2$ to capture smooth curves. Higher degree polynomials have more parameters to estimate, so they're more expressive and can fit the data better, but it's unwise to rely on them especially outside the range where we have data. In the iris dataset from page 22,

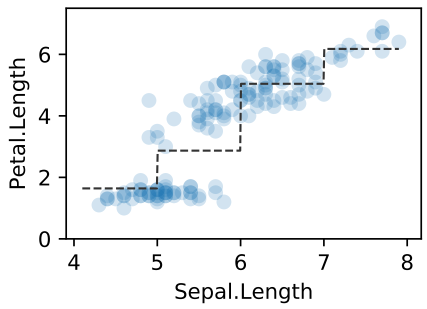
$$\text{Petal.Length} \approx \alpha + \beta_1\,\text{Sepal.Length} + \beta_2\,(\text{Sepal.Length})^2 + \cdots + \beta_K\,(\text{Sepal.Length})^K$$



A different approach is to use parameters for anchor points in an arbitrary curve. In this next model the arbitrary curve is a step function with fixed x-axis breaks, and least squares estimation finds the height at each step.

$\lfloor x \rfloor$ is x rounded down to the nearest integer

$$\text{Petal.Length} \approx \beta_4\,1\big[\lfloor\text{Sepal.Length}\rfloor == 4\big] + \cdots + \beta_7\,1\big[\lfloor\text{Sepal.Length}\rfloor == 7\big]\,.$$



This model is more honest because it is upfront about being an arbitrary fit to the data, incapable of extrapolating outside the data range. This example isn't interesting (we could just as well have fitted each integer bin separately), but it's very useful when combined with other features. More guidance on curve fitting on page 29.

2.2.3. PERIODIC PATTERNS

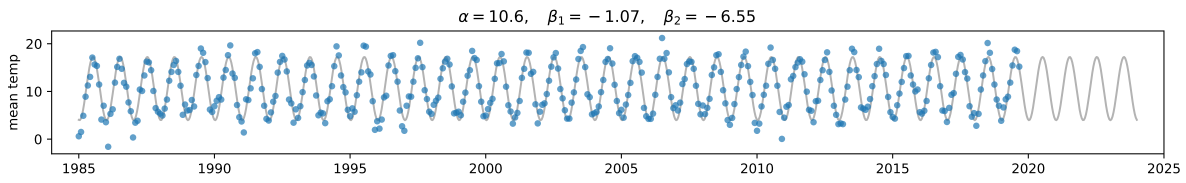
Example 2.2.
The UK Met Office makes available historic data[8] from 37 stations around the UK. Each station has monthly records for mean daily maximum temperature `tmax`, mean daily minimum temperature `tmin`, days of air frost `af`, total rainfall `rain`, and total sunshine duration `sun`. Coverage varies; the longest records are from Oxford and from Armagh, going back to 1853. A snapshot is available at `https://teachingfiles.blob.core.windows.net/datasets/climate.csv`.

month	tmax	tmin	af	rain	sun	station	lat	lng	alt_m
1963 Sep	14.7	5.9	0	126.4	127.7	Eskdalemuir	55.311	-3.206	242
1955 Aug	–	–	–	35.1	194.7	Shawbury	52.794	-2.663	72
1937 May	15.3	8.4	0	59.8	184.8	Lowestoft	52.483	1.727	18
2007 Aug	20.6	11.8	0	40.3	204.6	Waddington	53.175	-0.522	68
1925 July	21.8	12.6	0	23.2	–	Sheffield	53.381	-1.490	131
...									

The annual cycle makes it hard to compare the datapoints, for example to look for evidence of increasing temperatures. We could simply average over the 12 months of each year, and plot this average over time. This isn't ideal, because averaging is lossy i.e. we'd be throwing away data; and because a missing value for one month will cause the entire year to be missing. A cleverer solution is to use features to model the effects we're trying to capture. Let's consider the model

$$\mathsf{temp} \approx \alpha + \beta \sin(2\pi \mathsf{t} + \theta)$$

where t is the date in years, and α, β, and θ are unknown parameters. The plot below shows the data and the fitted model for Cambridge station (measured at the National Institute of Agricultural Botany, near the building for Artificial Intelligence and Environmental Risk). The plot shows the mean temperature $\mathsf{temp} = (\mathsf{tmin} + \mathsf{tmax})/2$.



The model is linear in α and β and not in θ—but there is a cunning trick from A-level trigonometry that lets us rewrite it as a linear model. The trick is

$$\sin(A + B) = \sin A \cos B + \cos A \sin B$$

and so our model can be rewritten

$$\mathsf{temp} \approx \alpha + \beta_1 \sin(2\pi \mathsf{t}) + \beta_2 \cos(2\pi \mathsf{t}).$$

```
climate = pandas.read_csv('https://teachingfiles.blob.core.windows.net/datasets/climate.cs
df = climate.loc[(climate.station=='Cambridge') & (climate.yyyy>=1985)]
t = df.yyyy + (df.mm-1)/12
temp = (df.tmin + df.tmax)/2

X = numpy.column_stack([numpy.sin(2*numpy.pi*t), numpy.cos(2*numpy.pi*t)])
model = sklearn.linear_model.LinearRegression()
model.fit(X, temp)
α,(β1,β2) = (model.intercept_, model.coef_)
```

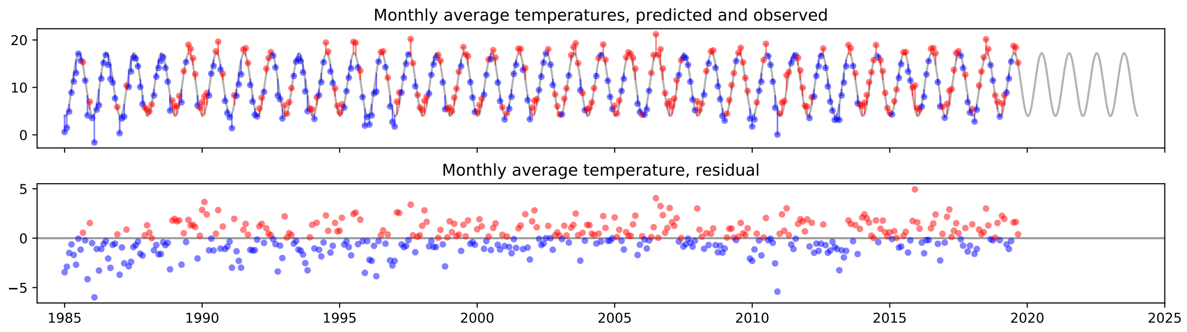
2.2.4. DISCOVERING FEATURES

It's often illuminating to plot the residual vector, to find out if we have missed any features worth incorporating. Here's an example. For the climate data above, we fitted the model

$$\mathsf{temp} = \alpha + \beta_1 \sin(2\pi \mathsf{t}) + \beta_2 \cos(2\pi \mathsf{t}) + \varepsilon$$

(this time we're writing the equation to make the residuals explicit). Here are two plots of ε, and they show clearly that there's a systematic trend over time—we see more positive ε in later years. If ε varies systematically with some feature, it's a sign that we could improve the model (make it fit better) by incorporating that feature into the model.

[8] https://www.metoffice.gov.uk/public/weather/climate-historic



```
10 pred = model.predict(X) # using model and X from lines 6–9 above
11 resid = temp − pred
12
13 with plt.rc_context({'figure.figsize': (15, 1.7*2.2), 'figure.subplot.hspace': 0.3}):
14     fig,(ax1,ax2) = plt.subplots(nrows=2,ncols=1, sharex=True)
15
16 # Top plot: arrows and points
17 for (t1,pred1,resid1) in zip(t, pred, resid):
18     ax1.arrow(t1, pred1, 0, resid1, alpha=0.5, color='blue' if resid1<=0 else 'red')
19 ax1.scatter(t, temp, s=15, alpha=0.5, c=numpy.where(resid<=0,'blue','red'))
20 # and a smooth line to show the fit
21 newt = numpy.linspace(1985, 2024, 1000)
22 newtemp = model.predict(numpy.column_stack([numpy.sin(2*numpy.pi*newt), numpy.cos(2*numpy.pi*newt)]))
23 ax1.plot(newt, newtemp, color='0.7', zorder=−1)
24
25 # Bottom plot: just the points
26 ax2.scatter(t, resid, s=15, alpha=0.5, c=numpy.where(resid<=0,'blue','red'))
27 ax2.axhline(0, color='0.6', zorder=−1)
28
29 ax1.set_xlim([1984, 2025])
30 ax1.set_title('Monthly average temperatures, predicted and observed')
31 ax2.set_title('Monthly average temperature, residual')
32 plt.show()
```

2.2.5. SECULAR TREND

We learnt from the residual plot that

$$\text{temp} \approx \alpha + \beta_1 \sin(2\pi t) + \beta_2 \cos(2\pi t)$$

isn't a great fit to the data, because this equation says that temperatures are purely periodic, whereas in fact there's a systematic increase over time. Systematic changes over time are called "secular", as opposed to periodic patterns (which, like the divine, are eternal and unchanging). We can incorporate a secular trend with a $+\gamma t$ term:

$$\text{temp} \approx \alpha + \beta_1 \sin(2\pi t) + \beta_2 \cos(2\pi t) + \gamma t$$

```
33 model = sklearn.linear_model.LinearRegression()
34 X = numpy.column_stack([numpy.sin(2*numpy.pi*t), numpy.cos(2*numpy.pi*t), t])
35 model.fit(X, temp)
36 α,(β_1,β_2,γ) = model.intercept_, model.coef_
```

```
(-60.458,  (-1.069, -6.5452, 0.0355))
```

Intercepts. In the purely periodic fit, line 9 in the code, we got $\alpha = 10.6$°C, a reasonable figure for average annual temperatures; whereas in the model with a secular trend we get $\alpha = -60.458$°C. Why is this α so extreme?

In either model, α is what we get from evaluating the model formula at $\mathsf{t} = 0$. In other words, α is the fitted temperature for January in 1 BC (there being no year 0 AD on the calendar). For the secular model, we're extrapolating a linear trend backwards from the present day to 1 BC, and it's daft to trust such a wild extrapolation! If we rewrite the model as

$$\mathsf{temp} \approx \alpha + \beta_1 \sin(2\pi\mathsf{t}) + \beta_2 \cos(2\pi\mathsf{t}) + \gamma(\mathsf{t\text{-}2000})$$

then α will report the temperature for January 2000. The two forms of the model give exactly the same predictions, but the second form gives more interpretable parameters.

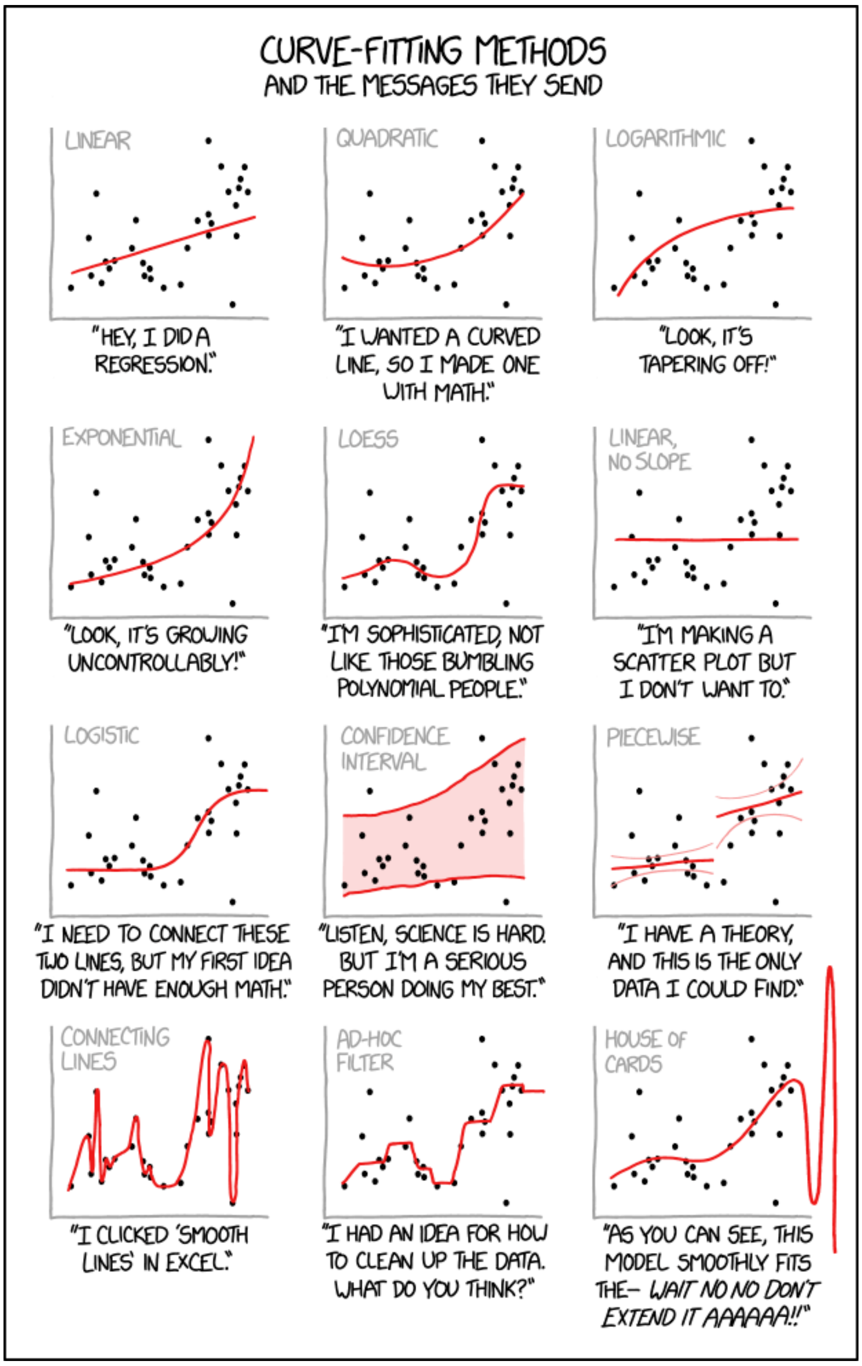
* ✻ *

We design features for several purposes:

- Features to extract a particular summary from the data, e.g. the linear trend in the climate data
- 'Black box' features that capture enough detail for us to be able to make good predictions or extrapolations—we don't have to understand such features, we just want them to work well
- Features that turn arbitrary objects like tweets or sentence fragments into numbers that can be put into quantitative models, e.g. distributional semantics which you will study in Part II *Natural Language Processing*, and term frequency models for documents which you will study in Part II *Information Retrieval*.

The more features we add, the better the fit i.e. the smaller the residual we can achieve. But models with too many features tend to be bad at generalizing to new data (see the polynomial fits in section 2.2.2). It's an art to design sets of features that are expressive enough to capture the meaningul variation in the data, while being parsimonious enough to generalize well.

xkcd by Randall Munroe, `https://xkcd.com/2048/`


CURVE-FITTING METHODS
AND THE MESSAGES THEY SEND
LINEAR
"HEY, I DID A REGRESSION."
QUADRATIC
"I WANTED A CURVED LINE, SO I MADE ONE WITH MATH."
LOGARITHMIC
"LOOK, IT'S TAPERING OFF!"
EXPONENTIAL
"LOOK, IT'S GROWING UNCONTROLLABLY!"
LOESS
"I'M SOPHISTICATED, NOT LIKE THOSE BUMBLING POLYNOMIAL PEOPLE."
LINEAR, NO SLOPE
"I'M MAKING A SCATTER PLOT BUT I DON'T WANT TO."
LOGISTIC
"I NEED TO CONNECT THESE TWO LINES, BUT MY FIRST IDEA DIDN'T HAVE ENOUGH MATH."
CONFIDENCE INTERVAL
"LISTEN, SCIENCE IS HARD. BUT I'M A SERIOUS PERSON DOING MY BEST."
PIECEWISE
"I HAVE A THEORY, AND THIS IS THE ONLY DATA I COULD FIND."
CONNECTING LINES
"I CLICKED 'SMOOTH LINES' IN EXCEL."
AD-HOC FILTER
"I HAD AN IDEA FOR HOW TO CLEAN UP THE DATA. WHAT DO YOU THINK?"
HOUSE OF CARDS
"AS YOU CAN SEE, THIS MODEL SMOOTHLY FITS THE- WAIT NO NO DON'T EXTEND IT AAAAAA!!"

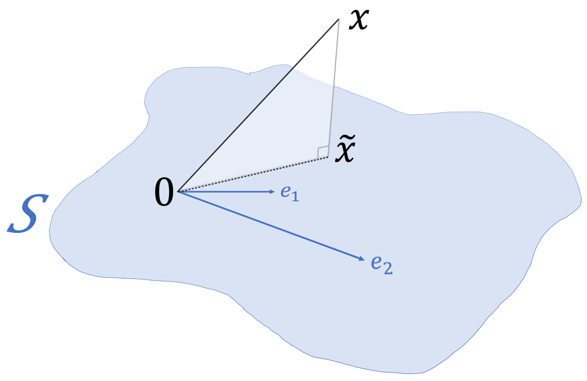
2.3. Linear mathematics

A linear model like the Iris model on page 22 is a vector equation,

$$\begin{bmatrix} \text{Petal.Length}_1 \\ \text{Petal.Length}_2 \\ \vdots \end{bmatrix} \approx \alpha \begin{bmatrix} 1 \\ 1 \\ \vdots \end{bmatrix} + \beta \begin{bmatrix} \text{Sepal.Length}_1 \\ \text{Sepal.Length}_2 \\ \vdots \end{bmatrix} + \gamma \begin{bmatrix} (\text{Sepal.Length}_1)^2 \\ (\text{Sepal.Length}_2)^2 \\ \vdots \end{bmatrix}.$$

In mathematics, vector equations like this come under the heading of *linear mathematics*. For data science all we need is vectors in simple Euclidean space, $\mathbb{R}^n$ where n is the number of records in the dataset—but it's good for the soul to define linear algebra abstractly, so that the concepts can be applied to other settings.[9] The appendix on page 65 presents the abstract maths.

For the purposes of this course, all the concepts from linear mathematics that we'll need are shown in this picture:



Linearly independent vectors e_1 and e_2. A collection of vectors $\{e_1, \dots, e_K\}$ is *linearly dependent* if there is some set of real numbers $(\lambda_1, \dots, \lambda_K)$, not all equal to zero, such that

$$\lambda_1 e_1 + \dots + \lambda_K e_K = 0$$

If so, at least one of the e_i can be written as a linear combination of the others. Otherwise, they are *linearly independent*, and

$$\lambda_1 e_1 + \dots + \lambda_K e_K = 0 \quad \Rightarrow \quad \lambda_1 = \dots = \lambda_K = 0.$$

If they are linearly independent then the rank of the matrix $[e_1, \dots, e_K]$ is K, otherwise the rank is $< K$.

The subspace S spanned by e_1 and e_2. The subspace spanned by a collection of vectors $\{e_1, \dots, e_K\}$, also called their span, is the set of all linear combinations:

$$S = \Big\{ \lambda_1 v_1 + \dots + \lambda_K v_K \ : \ \lambda_k \in \mathbb{R} \ \text{ for all } k \Big\}.$$

The projection of x onto $\tilde{x} \in S$. Given a subspace S spanned by $\{e_1, \dots, e_K\}$, and any other vector x, there is a unique vector $\tilde{x}$ that solves

$$\tilde{x} = \operatorname*{arg\,min}_{y \in S} \|x - y\|^2.$$

This $\tilde{x}$ is called the projection of x onto S, because the residual $x - \tilde{x}$ is orthogonal to S i.e. $(x - \tilde{x}) \cdot y = 0$ for all $y \in S$.

Since $\tilde{x} \in S$, it can be written as a linear combination of the e_k,

$$\tilde{x} = \hat{\lambda}_1 e_1 + \dots + \hat{\lambda}_K e_K.$$

Finding the $\hat{\lambda}_k$ is called least squares estimation, because the error term being minimized is a sum of squares. If the e_k are linearly independent then there is a unique solution for the $\hat{\lambda}_k$. Otherwise, there are multiple ways to write the linear combination.

[9] See Part II lecture courses on *Digital Signal Processing* and *Computer Vision* (Fourier transforms and wavelets, where vectors represent functions) and *Quantum Computing* (where vectors represent quantum states).

2.4. Linear regression and least squares

tl;dr. A *linear regression* is a probabilistic model of the form

$$Y_i \sim \beta_1 e_{1,i} + \cdots + \beta_K e_{K,i} + \text{Normal}(0, \sigma^2)$$

where $e_1, \ldots, e_K$ are covariates, Y is the random response, and σ and $\beta_1, \ldots, \beta_K$ are unknown parameters. It is implicit that the Y_i are independent. This is a supervised learning problem; the Y_i are the labels, and the $e_{k,i}$ are the predictors.

Fitting this model to a vector of observed values y is equivalent to least squares estimation for linear model

$$y = \beta_1 e_1 + \cdots + \beta_K e_K + \varepsilon.$$

Furthermore, the maximum likelihood estimator for σ^2 is the mean square error $\sum_i \varepsilon_i^2/n$.

To demonstrate the link between linear regression and linear models, it's easier to work through an illustration rather than to write out abstract equations.

For the Iris dataset on page 22, we investigated how petal length depends on sepal length. Consider the linear regression model

$$\mathsf{Petal.Length}_i \sim \alpha + \beta\,\mathsf{Sepal.Length}_i + \gamma\,(\mathsf{Sepal.Length}_i)^2 + \text{Normal}(0, \sigma^2)$$

where $i \in \{1, \ldots, n\}$ indexes the rows of the dataset, and each $\mathsf{Petal.Length}_i$ is an independent random variable, and $\mathsf{Sepal.Length}_i$ is being treated as a covariate i.e. a non-random value. For brevity, let $Y_i = \mathsf{Petal.Length}_i$, let $e_i = \mathsf{Sepal.Length}_i$, and let $f_i = (\mathsf{Sepal.Length}_i)^2$, giving

$$Y_i \sim \alpha + \beta e_i + \gamma f_i + \text{Normal}(0, \sigma^2)$$

which (following the remark on page 10) can be rewritten

$$Y_i \sim \text{Normal}\big(\alpha + \beta e_i + \gamma f_i\,,\ \sigma^2\big).$$

Then the density function for a single observation y_i is

$$\Pr(y_i \mid \alpha, \beta, \gamma, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\left(y_i - (\alpha + \beta e_i + \gamma f_i)\right)^2 / 2\sigma^2}$$

and the log likelihood of the entire dataset is

$$\log \text{lik}(\alpha, \beta, \gamma, \sigma \mid y) = -\frac{n}{2} \log\big(2\pi\sigma^2\big) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} \Big(y_i - (\alpha + \beta e_i + \gamma f_i)\Big)^2.$$

Let's find the maximum likelihood estimators for α, β, γ, and σ. We'll do this in two steps. The first step is to maximize the last term, i.e. find $\hat\alpha$, $\hat\beta$, and $\hat\gamma$ that solve

$$\min_{\alpha,\beta,\gamma} \big\|y - (\alpha 1 + \beta e + \gamma f)\big\|^2.$$

In this equation we have switched to vector notation, and 1 means the vector $[1, 1, \ldots, 1]$. This is nothing other than least squares estimation for the linear model

$$\mathsf{Petal.Length} \approx \alpha 1 + \beta \mathsf{Sepal.Length} + \gamma(\mathsf{Sepal.Length}^2).$$

The second step is to find σ to maximize what's left, i.e. to solve

$$\max_{\sigma > 0} \left\{ -\frac{n}{2} \log\big(2\pi\sigma^2\big) - \frac{1}{2\sigma^2} \big\|y - (\hat\alpha 1 + \hat\beta e + \hat\gamma f)\big\|^2 \right\}.$$

This is a simple one-parameter optimization problem, once we know $\hat\alpha$, $\hat\beta$, and $\hat\gamma$, and the solution is

$$\hat\sigma = \sqrt{\frac{1}{n} \big\|y - (\hat\alpha 1 + \hat\beta e + \hat\gamma f)\big\|^2}.$$

2.5. Confounded features

tl;dr. When we fit a linear regression

$$Y_i \sim \beta_1 e_{1,i} + \cdots + \beta_K e_{K,i} + N(0, \sigma^2)$$

to a dataset y, then finding maximum likelihood estimators is equivalent to finding least squares estimates for the corresponding linear model

$$y \approx \beta_1 e_1 + \cdots + \beta_K e_K.$$

- If the feature vectors $\{e_1, \ldots, e_K\}$ are linearly independent, then there is a unique solution for the least squares estimates and hence for the maximum likelihood estimates.
- If the feature vectors are *not* linearly independent, then the estimates are not unique; there are multiple ways to write any given linear combination of the feature vectors. We say the parameters are *non-identifiable* or *confounded*.

Identifiability is about understanding whether or not a dataset is capable of answering the questions we want answered.

The linear space spanned by the feature vectors is called the *feature space*.

To work out if the features are linearly independent, use either the definition of linear independence or the matrix rank property on page 30. A warning: sklearn.linear_model.LinearRegression always returns coefficients for *some* linear combination, and in the non-identifiable case it will make an arbitrary choice.

If the feature vectors are confounded, then there is some feature vector that can be written in terms of the others (by definition of linear independence). We can simply discard this feature; doing so won't change the feature space. For example, if there are three features $\{e, f, g\}$ and $e = 0.2f - 0.5g$, then any linear combination

$$y = \alpha e + \beta f + \gamma g$$

can be rewritten

$$y = (\beta + 0.2\alpha) f + (\gamma - 0.5\alpha) g.$$

Thus the model with only two features $\{f, g\}$ can express anything that can be expressed with $\{e, f, g\}$.

Exercise 2.3. Consider the simple straight-line linear regression

$$y \approx \alpha + \beta x$$

for the dataset $y = [5, 2, 1, 3]$, $x = [1, 2, 4, 5]$. Are the feature vectors linearly independent? If x or y were different, would your answer change?

The feature vectors are

$$\mathsf{one} = \begin{bmatrix}1\\1\\1\\1\end{bmatrix} \quad \textit{and} \quad \mathsf{x} = \begin{bmatrix}1\\2\\4\\5\end{bmatrix}.$$

We can just look at these and see that there is no way to write one *in terms of* x*, or* x *in terms of* one*, so they are linearly independent. Alternatively, use Python to check the rank of the matrix* [one,x] *is 2:*

```
one = [1,1,1,1]
x = [1,2,4,5]
numpy.linalg.matrix_rank(numpy.column_stack([one,x]))
```

To test linear independence for an arbitrary $x = [x_1, \ldots, x_n]$, we have to ask if it's possible to solve

$$\alpha\,\mathsf{one} + \beta\,\mathsf{x} = 0$$

with non-zero coefficients. It's reasonably easy to see, in this case, that if $x_1 = \cdots = x_n$ then they are linearly dependent: either all the x_i are equal to zero in which case $(\alpha, \beta) = (0, 1)$ works, or they *are nonzero in which case $(\alpha,\beta)=(x_1,1)$ works. In other words, if all the x coordinates are equal, then we can't fit a straight line.*

Alternatively, if we wanted to be formal about it, we'd write the vector equation as simultaneous equations,

$$\begin{aligned}\alpha+\beta x_1&=0\\ \alpha+\beta x_2&=0\\ &\vdots\end{aligned}$$

and solve them with pure algebra.

The question "are the feature vectors linearly independent?" is only about the feature vectors, so it doesn't depend on y, only on x.

∎

Example 2.4.

The UK Home Office makes available several datasets of police records, at data.police.uk. The dataset police is a log of stop-and-search incidents, available as `https://teachingfiles.blob.core.windows.net/datasets/stop-and-search.csv`. Here is a sample of rows.

police force	operation	date-time	lat	lng	gender	age	ethnicity
		object of search			outcome		
Hampshire	NA	2014-07-31T23:20:00	50.93	-1.38	Male	25–34	Asian
		controlled drugs			nothing found		
Hampshire	NA	2014-07-31T23:30:00	50.91	-1.43	Male	34+	White
		controlled drugs			suspect summonsed		
Hampshire	NA	2014-07-31T23:45:00	51.00	-1.49	Male	10–17	White
		controlled drugs			nothing found		
Hampshire	NA	2014-08-01T00:40:00	59.91	-1.40	Male	34+	White
		stolen goods			nothing found		
Hampshire	NA	2014-08-01T02:05:00	50.88	-1.32	Male	10–17	White
		article for use in theft			nothing found		

We wish to investigate whether there is racial bias in police decisions to stop-and-search. Consider the linear model

$$1[\mathsf{outcome}=\mathsf{find}]\approx\alpha+\beta_{\mathsf{eth}}$$

where eth is the vector of ethnicities. If the police are biased against ethnicity Black for example, then we'd expect relatively more fruitless stops of individuals of that ethnicity, i.e. more stops for which $1[\mathsf{outcome}=\mathsf{find}]=0$, hence β_{Black} would be smaller than the other β coefficients.

Write this model as a linear equation using one-hot coding. Are the parameters identifiable? If not, rewrite the model so that they are.

(It's a hack to treat a binary response as a real number with an implied Normal distribution. However, (i) the question only asks about parameter identifiability, which is a question about the feature vectors not the response, and (ii) the hack can still give us interesting answers about the distribution of response.)

With one-hot coding, the model is

$$1[\mathsf{outcome{=}find}]\approx\alpha\,\mathsf{one}+\sum_{k\in\textit{ethnicities}}\beta_k 1[\mathsf{eth}=k].$$

We want to know whether the parameters are identifiable, i.e. whether the feature vectors are linearly independent. We can test this by looking at the matrix rank. There are 6 features, but the matrix rank is only 5, therefore they're not linearly independent.

```
# It's a big file, so retrieve it and store locally for future use
if os.path.exists('stop-and-search.csv'):
    print("file already downloaded")
else:
    !wget "https://teachingfiles.blob.core.windows.net/datasets/stop-and-search.csv"
police = pandas.read_csv('stop-and-search.csv')

```

```
# Discard rows with missing ethnicity
ethnicity_levels = ['Asian','Black','Mixed','Other','White']
ok = police['officer_defined_ethnicity'].isin(ethnicity_levels)
eth = police.loc[ok, 'officer_defined_ethnicity']

# Assemble the feature matrix, one column per feature, and check its rank
eth_onehot = [(eth==i).astype(int) for i in ethnicity_levels]
X = numpy.column_stack([numpy.ones(len(eth))] + eth_onehot)
X.shape, numpy.linalg.matrix_rank(X)
```

```
((940998, 6), 5)
```

But this doesn't give us any insight into what's wrong with the model. For that, maths is better. First, note that the feature matrix has lots of duplicate rows, which (for the purposes of understanding identifiability) are redundant. In fact, there are only as many distinct rows as there are distinct ethnicity levels in the dataset, namely 5.

```
police.groupby('officer_defined_ethnicity').apply(len)
```

```
Asian    125646
Black    253315
Mixed      1644
Other     27809
White    532584
```

So we're essentially only interested in the vectors

$$\begin{array}{cccccc} \textsf{one} & 1[\textsf{eth=Asian}] & 1[\textsf{eth=Black}] & 1[\textsf{eth=Mixed}] & 1[\textsf{eth=Other}] & 1[\textsf{eth=White}] \\ \begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix} & \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix} & \begin{bmatrix}0\\1\\0\\0\\0\end{bmatrix} & \begin{bmatrix}0\\0\\1\\0\\0\end{bmatrix} & \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix} & \begin{bmatrix}0\\0\\0\\0\\1\end{bmatrix} \end{array}$$

Clearly the five one-hot coded vectors sum up to one*, so the* one *vector is redundant. Also, those five vectors are clearly linearly independent. So if we write the model as*

$$1[\textsf{outcome=find}] \approx \sum_{k \in \textit{ethnicities}} \beta_k 1[\textsf{eth} = k]$$

then the parameters are identifiable.
∎

* ✻ *

Data science is all about noise and uncertainty, whereas linear independence is a strict clean mathematical definition, so we shouldn't pay too much attention to it. Even when feature vectors are linearly independent, if they are closely correlated then the parameters will be hard to identify—in a sense which will be made precise when we study confidence intervals in part III. Features that are linearly independent but closely correlated are also described as confounded.

2.6. Gauss's invention of least squares *

There is a link between linear regression and least squares estimation, but it's not just "Oh, how nice, after we've done least squares estimation we can express our answer as a probability model." Arguably, the probability model has primacy. (i) In many situations, random quantities can be approximated by a Normal distribution. (ii) Likelihood is a fundamental measure of evidence for all sorts of inference procedures. (iii) Maximum likelihood estimation for Normal random variables is equivalent to least squares estimation. (iv) Therefore, least squares estimation is a reasonable thing to do, and not just a totally heuristic kludge.

Part III will study inference procedures. Part IV will explain the use of the Normal as an approximation.

Least squares estimation was invented by Carl Friedrich Gauss, the 'prince of mathematicians', who also invented the Gaussian distribution—referred to in these notes as the Normal distribution. Here is Gauss's account[10] of how the idea of least squares came to him. Before Gauss, . . .

> *. . . in every case in which it was necessary to deduce the orbits of heavenly bodies from observations, there existed advantages not to be despised, suggesting, or at any rate permitting, the application of special methods; of which advantages the chief one was, that by means of hypothetical assumptions an approximate knowledge of some elements could be obtained before the computation of the elliptic elements was commenced. Notwithstanding this, it seems somewhat strange that the general problem—To determine the orbit of a heavenly body, without any hypothetical assumption, from observations not embracing a great period of time, and not allowing the selection with a view to the application of special methods,—was almost wholly neglected up to the beginning of the present century; or at least, not treated by any one in a manner worthy its importance; since it assuredly commended itself to mathematicians by its difficulty and elegance, even if its great utility in practice were not apparent. An opinion had universally prevailed that a complete determination from observations embracing a short interval of time was impossible—an ill-founded opinion—for it is now clearly shown that the orbit of a heavenly body may be determined quite nearly from good observations embracing only a few days; and this without any hypothetical assumption.*
>
> *Some idea occurred to me in the month of September of the year 1801, engaged at the time on a very different subject, which seemed to point to the solution of the great problem of which I have spoken. Under such circumstances we not unfrequently, for fear of being too much led away by an attractive investigation, suffer the associations of ideas, which more attentively considered, might have proved most fruitful in results, to be lost from neglect. And the same fate might have befallen these conceptions, had they not happily occurred at the most propitious moment for their preservation and encouragement that could have been selected. For just about this time the report of the new planet, discovered on the first day of January of that year with the telescope at Palermo, was the subject of universal conversation; and soon afterwards the observations made by the distinguished astronomer Piazzi from the above date to the eleventh of February were published. Nowhere in the annals of astronomy do we meet with so great an opportunity, and a greater one could hardly be imagined, for showing most strikingly, the value of this problem, than in this crisis and urgent necessity, when all hope of discovering in the heavens this planetary atom, among innumerable small stars after the lapse of nearly a year, rested solely upon a sufficiently approximate knowledge of its orbit to be based upon these very few observations. Could I ever have found a more seasonable opportunity to test the practical value of my conceptions, than now in employing them for the determination of the orbit of the*

[10]Carl Friedrich Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientum.* 1809. English translation: Charles Henry Davis. *Theory of the motion of the heavenly bodies moving about the sun in conic sections.* 1857. URL: `https://quod.lib.umich.edu/m/moa/AGG8895.0001.001/15?rgn=full+text;view=image`.

planet Ceres, which during the forty-one days had described a geocentric arc of only three degrees, and after the lapse of a year must be looked for in a region of the heavens very remote from that in which it was last seen? This first application of the method was made in the month of October, 1801, and the first clear night, when the planet was sought for (by de Zach, December 7, 1801) as directed by the numbers deduced from it, restored the fugitive to observation. Three other new planets, subsequently discovered, furnished new opportunities for examining and verifying the efficiency and generality of the method.

Several astronomers wished me to publish the methods employed in these calculations immediately after the second discovery of Ceres; but many things—other occupations, the desire of treating the subject more fully at some subsequent period, and, especially, the hope that a further prosecution of this investigation would raise various parts of the solution to a greater degree of generality, simplicity, and elegance,—prevented my complying at the time with these friendly solicitations. I was not disappointed in this expectation, and I have no cause to regret the delay. For the methods first employed have undergone so many and such great changes, that scarcely any trace of resemblance remain between the method in which the orbit of Ceres was first computed, and the form given in this work. Although it would be foreign to my purpose, to narrate in detail all the steps by which these investigations have been gradually perfected, still, in several instances, particularly when the problem was one of more importance than usual, I have thought that the earlier methods ought not to be wholly suppressed. But in this work, besides the solution of the principal problems, I have given many things which, during the long time I have been engaged upon the motions of the heavenly bodies in conic sections, struck me as worthy of attention, either on account of their analytical elegance, or more especially on account of their practical utility.

Part II
Handling probability models

3. Simulations and calculations

Computer scientists are used to reasoning about algorithms, for example to prove correctness or to analyse running time. Dijkstra is associated with this school of mathematical reasoning about code:[11]

> *Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.*

Suppose we have a simulator that uses random number generators—in other words, suppose we have a probability model expressed in code. It's natural to want to analyse it mathematically, Dijsktra-style. For example, we might have implemented a climate simulator that uses random variables, and we might want to calculate the distribution of its output, e.g. the frequency of extreme events. This will typically involve integrals, for calculating probabilities and expectations.

There is another stance, diametrically opposed to Dijkstra's. Suppose we have a probability model for the climate, expressed as equations with random variables, and we want to calculate the distribution of the output. If the maths is too hard (as it usually is), we can just implement the model in a simulator, and run it, and we'll *see* its output directly. (Or, more precisely, if we run it many times, we'll see a random sample drawn from the model's output distribution.) This is called Monte Carlo simulation—so perhaps the mathematical approach ought to be called the "Trinity College method" in honour of Newton, its most famous son.

The middle way, a bit of computation and a bit of maths, is best. There are some situations where the maths is too hard and computation is the only tool available. There are other situations, such "inverse problems" in which we know the model's output and want to deduce the likely input, where computation is too slow without some mathematical help. It's easy to get 1000-fold speedup by replacing an inner simulation loop with a deft equation.

The question of how accurate Monte Carlo simulation is, we defer to the study of limit theorems in part IV.

* ✻ *

In this section we will study various mathematical and simulation tools for reasoning about probability models. From the point of view of this course, this section is really a build-up to Bayesian inference.

- Bayes's rule is based on conditional probability, so you need understand the idea of a conditional random variable (section 3.5) which is built on the idea of random tuples and marginal densities (section 3.4)
- Bayesian calculations generally involve the "densities sum to one" rule (section 3.2)
- Bayesian computation can be done with weighted Monte Carlo integration, which builds on standard Monte Carlo integration (section 3.1).

[11] Edsger W. Dijkstra. "How do we tell truths that might hurt?" personal note EWD498. URL: `http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF`.

3.1. Monte Carlo integration

tl;dr. The *mean* or *expectation* of a random variable X is

$$\mathbb{E}\,X = \begin{cases} \sum_x x\,\Pr_X(x) & \text{for a discrete random variable} \\ \int_x x\,\Pr_X(x)\,dx & \text{for a continuous random variable.} \end{cases}$$

More generally, if h is some real-valued function, then $\mathbb{E}\,h(X)$ is

$$\mathbb{E}\,h(X) = \begin{cases} \sum_x h(x)\,\Pr_X(x) & \text{for a discrete random variable} \\ \int_x h(x)\,\Pr_X(x)\,dx & \text{for a continuous random variable.} \end{cases}$$

This can be approximated by

$$\mathbb{E}\,h(X) \approx \frac{1}{n}\sum_{i=1}^{n} h(x_i)$$

where $x_1, \dots, x_n$ is a sample drawn from distribution X. This approximation is called *Monte Carlo integration*.

The formal statement of Monte Carlo integration is useful when we come in part IV to analysing how accurate the approximation is, but it obscures the simplicity of the idea. To better convey the idea, here are some different contexts in which it's used. Afterwards, on page 39, a note on efficient implementation in Python.

ESTIMATING PROBABILITIES

Suppose we want to estimate $\mathbb{P}(X \in A)$. The obvious strategy is to simulate X many times, and count how often it lies in A.

```
# Let X ~ N(μ = 1, σ = 3). What is P(X > 5)?
x = numpy.random.normal(loc=1, scale=3, size=10000)  # simulate the r.v.
i = x > 5        # i is a Boolean vector, same length as x
numpy.mean(i)    # returns the average of i, treating True as 1 and False as 0
```

To connect this to the abstract definition, we're defining

$$h(x) = \begin{cases} 1 & \text{if } x > 5 \\ 0 & \text{if } x \leq 5, \end{cases}$$

setting $I = h(X)$, and computing $\mathbb{E}\,I$. This function h is called an *indicator function*, also written $h(x) = 1_{x>5}$ or $h(x) = 1[x > 5]$, and so another way of writing the approximation is

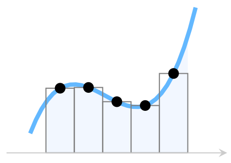
$$\mathbb{P}(X \in A) = \mathbb{E}\,1_{X\in A} \approx \frac{1}{n}\sum_{i=1}^{n} 1_{X_i \in A}.$$

ESTIMATING AN INTEGRAL

Suppose we've been asked to find

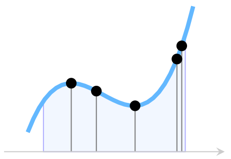
$$\int_{x=a}^{b} h(x)\,dx\,.$$

The method you might have learnt at school is to split the x range into n equally sized pieces, and approximate the function by a series of rectangles, taking the height of the rectangle to be the value of h at the midpoint.



$$\approx \sum_{i=1}^{n} h(x_i)w, \quad \text{where } x_i = a + w\big(i - 1/2\big)\,, \quad w = \frac{b-a}{n}.$$

The sum is just h evaluated at n grid points, times a constant w. There's nothing special about those grid points. Why not just pick the sampling points at random? In other words, pick n independent Uniform$[a, b]$ random variables $X_1, \ldots, X_n$, and approximate



$$\approx \frac{b-a}{n} \sum_{i=1}^{n} h(X_i).$$

To connect this to the abstract definition, we're using $X \sim U[a, b]$, so

$$\begin{aligned}
\mathbb{E}\, h(X) &= \int h(x) \Pr_X(x)\, dx && \text{by definition of expectation} \\
&= \int_a^b h(x) \frac{1}{b-a}\, dx && \text{since } \Pr_X(x) = \frac{1}{b-a} \text{ for } x \in [a, b] \\
&\approx \frac{1}{n} \sum_{i=1}^{n} h(X_i) && \text{by the Monte Carlo approximation}
\end{aligned}$$

and so, rearranging,

$$\int_a^b h(x)\, dx \approx \frac{b-a}{n} \sum_{i=1}^{n} h(X_i)\,.$$

VECTORIZED COMPUTATION *

To compute the Monte Carlo approximation $\sum_i h(x_i)/n$, we need to generate a large sample $(x_1, \ldots, x_n)$ and then apply h to each item. In Python, the best coding style for this is vectorized.

Vectorized thinking is great for conciseness. Surely no one would prefer the iterative style

```
tot = 0
for _ in range(n):
    x = rng()
    tot = tot + h(x)
tot/n
```

or even the list comprehension style

```
xs = [rng() for _ in range(n)]
sum(h(x) for x in xs) / n
```

when they can just write

```
x = rng(size=n)
numpy.mean(h(x))
```

But vectorized coding is perhaps more important from the point of view of performance. Every time the Python interpreter has to evaluate a Python expression there's a performance hit; the first two versions take this hit on every sample, whereas in the vectorized version the iteration is all done in numpy's C code. On a larger scale, if n is so large that the computation should be split across multiple cores or machines, then it's hard for a compiler to see how to achieve parallelization when the function is written out as iteration, much easier when it is vectorized. Vectorized thinking means avoiding for loops and instead writing our computations in a way that shows our intention more clearly, to give the compiler a chance to figure out what can be distributed and parallelized.

It's worth getting familiar with the vectorized routines in numpy. See the course material for IA Scientific Computing, chapter 2.

3.2. Probability densities sum to one

There are two properties that arises again and again in Bayesian calculations. For any random variable X, and for any event A,

$$\sum_x \Pr_X(x) = 1 \quad \text{and} \quad \mathbb{P}(A) = \sum_x \mathbb{P}(A \mid X = x) \Pr_X(x).$$

The first is called "densities sum to one", and the second is called "the law of total probability". These equations as written are for discrete random variables; if X is a continuous random variable then replace the sum by an integral.

Exercise 3.1.
Let X be a random variable taking values $\{0, 1, \dots\}$ in with $\Pr_X(x) = \kappa r^x$ where $0 < r < 1$ is given and κ is a constant. Find κ.

Standard maths formulae: $1 + r + r^2 + \cdots = 1/(1-r)$ for $|r| < 1$, and $1 + r + \cdots + r^n = (1 - r^{k+1})/(1-r)$.

By the "densities sum to one" rule, $\sum_{x=0}^{\infty} \Pr_X(x) = 1$, *hence*

$$\sum_{x=0}^{\infty} \kappa r^x = \frac{\kappa}{1-r} = 1$$

hence $\kappa = (1-r)$. *Furthermore,*

$$F(x) = \sum_{y=0}^{x} (1-r) r^x = (1-r) \frac{1 - r^{x+1}}{1-r} = 1 - r^{x+1}.$$

∎

Exercise 3.2 (The Beta distribution).
The Beta distribution describes a continous $[0, 1]$-valued random variable, with density function

$$\Pr(x) = \binom{a+b-1}{a-1} x^{a-1} (1-x)^{b-1}$$

(but with a generalized form of the binomial coefficient when a or b is non-integer).

1. Suppose Y is a $[0, 1]$-valued continuous random variable with density
$$\Pr_Y(y) = \kappa y^2 (1-y)^3 .$$
Find κ.
2. Without using any calculus, find
$$\int_{y=0}^{1} y \Pr_Y(y)\, dx .$$

Part (1). *By the "densities sum to one" rule applied to* $X \sim \text{Beta}(a, b)$,

$$\int_{x=0}^{1} \binom{a+b-1}{a-1} x^{a-1} (1-x)^{b-1}\, dx = 1,$$

and if we let $\tilde{a} = a - 1$ *and* $\tilde{b} = b - 1$ *and rearrange, we get*

$$\int_{x=0}^{1} x^{\tilde{a}} (1-x)^{\tilde{b}}\, dx = 1 \Big/ \binom{\tilde{a} + \tilde{b} + 1}{\tilde{a}}.$$

Applying this to Y,

$$\int_{y=0}^{1} \Pr_Y(y)\, dy = \kappa \Big/ \binom{6}{2}.$$

By the "densities sum to one" rule, $\kappa = \binom{6}{2} = 15$.

Part (2). *The integral we want is*

$$\int_{y=0}^{1} \kappa y^3 (1-y)^3\, dy = \kappa \Big/ \binom{7}{3} = 15/35 = 3/7.$$

∎

3.3. Handling numerical random variables *

Calculating with probabilities, like any area of mathematics, is a skill that can only be aquired through practice. There is no universal method (thanks, Gödel!), but nevertheless here's a tip:

- Try to express your problem in equations that use random variables.
- If that doesn't work, try working with the cumulative distribution function $\mathbb{P}(X \leq x)$, or the tail distribution function $\mathbb{P}(X > x)$.
- If everything fails, your last resort is the density function $\Pr_X(x)$ and integration.

To understand the concepts behind data science and machine learning, you don't actually need to *do* any probability calculations, you can just use Monte Carlo integration instead. But if your competitor is good at probability, they'll be able to write faster code. Also, when it comes to the exam you won't have a computer, so you'll have to be able to do *some* calculation …

TRANSFORMING RANDOM VARIABLES

When trying to find the distribution of a continuous random variable that's a function of another, try working with the cumulative distribution function because that's often easier than densities. For discrete random variables, the cumulative distribution function is also a good bet, but densities also work.

> Exercise 3.3. Let $U \sim U[0,1]$ and let $X = U^2$. Find the density of X.

Let's first find the cumulative distribution of X:

$$\begin{aligned}\mathbb{P}(X \leq x) &= \mathbb{P}(U^2 \leq x) \qquad \textit{by definition of } X \\ &= \mathbb{P}(U \leq \sqrt{x}) \qquad \textit{assuming } x \geq 0\textit{, otherwise the probability is 0} \\ &= \sqrt{x} \qquad \textit{since for a Uniform r.v. } \mathbb{P}(U \leq y) = y \textit{ for } y \in [0,1]\end{aligned}$$

Now we can find the density by differentiation:

$$\Pr_X(x) = \frac{d}{dx}\mathbb{P}(X \leq x) = \frac{1}{2\sqrt{x}}.$$

∎

> Exercise 3.4 (Generating an Exponential and a Geometric).
> Here is code for generating a random variable:
>
> ```
> p = ...    # a constant in the range (0,1)
> λ = − math.log(1−p)
> u = random.random()
> x = − math.log(u) / λ
> y = math.ceil(x)
> ```
>
> Find the distributions of the random variables generated in lines 4 and 5.
>
> *When you are asked "find the distribution", you can give your answer either as a density, or as a cumulative distribution function, or, if it's a standard distribution, by naming the distribution.*

Write U, X, and Y for the three random variables. Since U is generated by random.random() *it's a simple Uniform random variable, for which*

$$\mathbb{P}(U \leq x) = x \quad \textit{for} \quad x \in [0,1].$$

*To find the distribution of $X = -$*math.log*$(U)/\lambda$, let's try working with the cumulative distribution function:*

math.log is the natural logarithm.

$$\mathbb{P}(X \leq x) = \mathbb{P}\Big(-\frac{1}{\lambda}\log(U) \leq x\Big) = \mathbb{P}\big(\log(U) \geq -\lambda x\big) = \mathbb{P}(U \geq e^{-x}) = 1 - e^{-x}.$$

This is the cumulative distribution function of an Exponential random variable of rate λ.

List of common random variables: appendix page 62

Now for the distribution of $Y=$ math.ceil(X). This is a discrete random variable, and so we could work with either the cumulative distribution function or the density. Let's try the former. For any integer k,

math.ceil rounds u
the nearest integer

$$\mathbb{P}(Y \le k) = \mathbb{P}(\text{math.ceil}(X) \le k) = \mathbb{P}(X \le k) = 1 - e^{-\lambda k}.$$

(This is the same formula as the cumulative distribution function for X, but X and Y are not the same: X is a continuous random variable and the formula holds for any $x > 0$, whereas Y is a discrete random variable and the formula is only correct for integer k.) Rewriting in terms of p,

$$\mathbb{P}(Y \le k) = 1 - (e^{-\lambda})^k = 1 - (1-p)^k.$$

We might recognize this as the cumulative distribution function for a Geometric random variable and stop here. Or we could also go a step further and work out the density:

$$\mathbb{P}(Y = k) = \mathbb{P}(Y \le k) - \mathbb{P}(Y \le k-1) = (1-p)^{k-1} - (1-p)^k = (1-p)^{k-1}p.$$

The probabilities subtract like this because the event $\{Y \le k-1\}$ is nested inside the event $\{Y \le k\}$.
∎

Exercise 3.5 (Equality of distributions).
Here are two pieces of code for generating random variables:

```
def rgeom(p):
    λ = − math.log(1−p)
    u = random.random()
    x = − math.log(u) / λ
    return math.ceil(x)

def rgeom2(p):
    x = 1
    while random.random() > p:
        x = x + 1
    return x
```

Show that the two pieces of code generate identically distributed random variables.

We have shown in exercise 3.4 that rgeom(p) *generates a* Geom(p) *random variable, for which*

$$\Pr(k) = (1-p)^{k-1}p, \quad \textit{for } k \in \{1,2,\dots\}.$$

"One thing after another" probability models are called Markov chains. See part IV.

Now turn to rgeom2(p). *This is a "one thing after another" piece of code, and for such cases we often brute-force the calculation by using densities. Let's work out some examples.*

- *In order for* rgeom2 *to output 1, we need the* random.random()$>p$ *test to fail on the first pass, i.e. we need* random.random()$<=p$, *which happens with probability p.*
- *For* rgeom *to output 2, we need the test to succeed on the first pass then fail on the second. Let's call the first* random.random() *value U_1 and the second U_2. By the way the code is written, U_1 and U_2 are independent, so*

Why are U_1 and U_2 independent? Look back at page 7.

$$\mathbb{P}(X_1 > p \textit{ and } X_2 \le p) = \mathbb{P}(X_1 > p)\,\mathbb{P}(X_2 \le p) = (1-p)p.$$

- *Generalizing, if X is the output of* rgeom2(p), *then*

$$\mathbb{P}(X = k) = (1-p)^{k-1}p.$$

Thus the output of rgeom2 *is a* Geom(p) *random variable, the same distribution as produced by* rgeom.
∎

OTHER TRICKS

The example sheet will guide you through some other useful tricks for probability calculations.

3.4. Random tuples

Two or more random variables may be linked. For example, here's some code that simulates a pair of dice throws repeatedly until they give different values. If I generate (X,Y) with this code, and tell you the value of X, then that gives you some information about the value of Y.

```
def rxy():
    while True:
        x = random.randint(1,6)
        y = random.randint(1,6)
        if x != y:
            break
        return (x,y)
```

We often work with probability models that produce random tuples and we're only interested in one of them, so we just ignore the others. This is called *marginalization*.

```
xysample = [rxy() for _ in range(10000)]
xsample = [x for x,y in xysample]
```

In mathematical notation, a pair of discrete random variables is described by their *joint density* $\Pr_{X,Y}(x,y) = \mathbb{P}(X = x \text{ and } Y = y)$, so that

$$\mathbb{P}\big((X,Y) \in C\big) = \sum_{(x,y)\in C} \Pr_{X,Y}(x,y)$$

and the marginal distribution of X is

$$\Pr_X(x) = \sum_y \Pr_{X,Y}(x,y).$$

These two equations also hold for a pair of continuous random variables; just replace sums by integrals.

Formally, the pair $Z = (X,Y)$ is itself a random variable, taking values e.g. in $\mathbb{R}^2$, and $\Pr_{X,Y}$ is just its density. The "probability densitites sum to one" rule applies to Z: $\sum_{x,y} \Pr_{X,Y}(x,y) = 1$. And all of these ideas extend to tuples of any length.

Recall from section 1.4 that X and Y are *independent* if

$$\Pr_{X,Y}(x,y) = \Pr_X(x)\Pr_Y(y).$$

FINDING MARGINAL DISTRIBUTIONS

Mathematicians who work in probability theory spend a lot of their time doing calculations with joint distributions. For the purposes of this course, the only calculation that really matters is finding a marginal distribution.

Exercise 3.6. The precise number of people on the Titanic is unclear due to confusion over passenger lists. One dataset lists 2201 passengers, and for each it gives the class $C \in \{\text{1st, 2nd, 3rd, crew}\}$, the sex $S \in \{\text{male, female}\}$, the age $A \in \{\text{child, adult}\}$, and whether or not this person survived $Y \in \{\text{survived, died}\}$.

Find the marginal distribution of Y. Find the marginal distribution of (Y,S). Explain the relationship between the two.

The dataset is available at `https://teachingfiles.blob.core.windows.net/datasets/titanic.csv`.

```
titanic = pandas.read_csv('https://teachingfiles.blob.core.windows.net/datasets/titanic.csv')

# The marginal distribution of Y
x = titanic.groupby('Survived').apply(len)
x/sum(x)
```

```
Survived
died        0.676965
survived    0.323035
```

```
# The marginal distribution of (Y, S), with row and column totals.
x = titanic.groupby(['Survived','Sex']).apply(len)
x = x / sum(x)
x = x.unstack()                     # convert the answer to an array
x.loc['TOTAL'] = x.sum(axis=0)      # add column and row sums
x['TOTAL'] = x.sum(axis=1)
x
```

```
Sex          Female      Male       ALL
Survived
died       0.057247  0.619718  0.676965
survived   0.156293  0.166742  0.323035
ALL        0.213539  0.786461  1.000000
```

The marginal distribution of Y shows itself as the row totals in a table showing the distribution of (Y,S). (It's called "marginal distribution" because the obvious place to write row and column totals is in the margin of the table.)

∎

Exercise 3.7. Consider a pair of random variables with joint density

$$\Pr_{X,Y}(x,y) = \frac{3}{16}xy^2, \qquad 0 \le x \le 2,\ 0 \le y \le 2.$$

Find $\Pr_X(x)$ and $\Pr_Y(y)$.

$$\Pr_X(x) = \int_{y=0}^{2} \Pr_{X,Y}(x,y)\,dy = \frac{3x}{16}\int_{y=0}^{2} y^2\,dy = x,$$
$$\Pr_Y(y) = \int_{x=0}^{2} \Pr_{X,Y}(x,y)\,dx = \frac{3y^2}{16}\int_{x=0}^{2} x\,dx = \frac{3}{2}y^2.$$

∎

MORE CALCULATIONS

The general advice from section 3.3 applies here.

- First, try to work with equations involving random variables if you can; it's always helpful to be able to reduce the problem to equations that only involve simple independent random variables.
- Second, when working with continuous random variables, try working with cumulative distribution functions if possible.
- If all else fails, the last resort is to work with density functions and integration. For pairs of random variables, this means double integrals.

For continuous random variables, if we want to find the joint density, it's usually easier to start with the cumulative distribution function and differentiate. To see why this is true: the fundamental equation for probabilities of joint random variables tells us that

$$\mathbb{P}\big(X \le x \text{ and } Y \le y\big) = \int_{x'=-\infty}^{x}\int_{y'=-\infty}^{y} \Pr_{X,Y}(x,y)\,dxdy$$

and so, differentiating,

$$\Pr_{X,Y}(x,y) = \frac{\partial^2}{\partial x\,\partial y}\,\mathbb{P}\big(X \le x \text{ and } Y \le y\big).$$

Exercise 3.8 (Joint density of discrete r.v.).
For the rxy() function on page 43, find the joint distribution of (X,Y), and the marginal distributions of X and Y.

Every outcome $(x,y) \in \{1,\dots,6\}^2$ is equally likely, except those where $x=y$ which are impossible. (We could wave our hands here and mumble "they are equally likely, by symmetry", or we could be ultra-formalistic and use the mathematical tools for Markov Chains from part IV.) So the joint distribution is

1_b is the indicator function, for converting from boolean to integer: $1_{\text{true}} = 1$, $1_{\text{false}} = 0$.

$$\begin{aligned}\Pr_{X,Y}(x,y) &= \begin{cases}\kappa & \text{if } x \neq y, \text{ for some constant } \kappa \\ 0 & \text{if } x = y\end{cases} \\ &= \kappa 1_{x\neq y}.\end{aligned}$$

By the "densities sum to one" rule,

$$\sum_{x=1}^{6}\sum_{y=1}^{6} \kappa 1_{x\neq y} = 1 \quad \Longrightarrow \quad \kappa = \frac{1}{30}.$$

The marginal distribution of X is

$$\Pr_X(x) = \sum_y \Pr_{X,Y}(x,y) = \sum_{y=1}^{6} \frac{1}{30} 1_{y\neq x} = \frac{5}{30} = \frac{1}{6},$$

hence X is uniformly distributed in $\{1,\dots,6\}$, and likewise Y.

∎

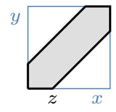
Exercise 3.9 (Findinding density by differentiation).
Let X and Y be independent Uniform$[0,1]$ random variables. Find $\mathbb{P}(|X-Y|<z)$. Hence find the density of $|X-Y|$.

Because X and Y are independent their joint density is

$$\Pr_{X,Y}(x,y) = \Pr_X(x)\Pr_Y(y) = 1 \quad \text{for } x,y \in [0,1].$$

Then,

$$\mathbb{P}\big(|X-Y|<z\big) = \int_{(x,y)\,:\,|x-y|<z} 1\,dx\,dy = \int_{x=0}^{1}\int_{y\,:\,|y-x|<z} 1\,dy\,dx = 1-(1-z)^2$$



and the density of $|X-Y|$ is

$$\Pr_{|X-Y|}(z) = \frac{d}{dz}\Big\{1-(1-z)^2\Big\} = 2(1-z).$$

∎

Exercise 3.10 (Joint density of continuous r.v.).
Consider the code

```
def rxy():
    x = random.random()
    z = random.random()
    y = z * x
    return (x,y)
```

Find the joint density of (X,Y).

The general advice says that to find the joint density of (X,Y) we should start by finding the joint cumulative distribution function $\mathbb{P}(X\leq x,\ Y\leq y)$. The general advice also says to start by rewriting equations in terms of simple indendent random variables, if we can, which in this case are X and Z; and then if all else fails to resort to integrals. We'll follow this advice here. The equations are hairy, but there's nothing to them beyond basic integration and being absolutely meticulous about the limits *for the integrals. Let's proceed: for $x \in [0,1]$ and $y \in [0,1]$,*

$$\begin{aligned}
\mathbb{P}(X \leq x \text{ and } Y \leq y) &= \mathbb{P}(X \leq x \text{ and } ZX \leq y) \\
&= \mathbb{P}\big((X,Z) \in \{(x',z) \,:\, x' \leq x \text{ and } zx' \leq y\}\big) \\
&= \int_{(x',z)\,:\,x'\leq x \text{ and } zx'\leq y} 1_{x\in[0,1]} 1_{z\in[0,1]}\, dx'dz \\
&= \int_{x'=0}^{x} \int_{z=0}^{\min(y/x',1)} 1\, dzdx' \quad \text{assuming } x,y \in [0,1] \\
&= \int_{x'=0}^{x} \min\Big(\frac{y}{x'},1\Big)\, dx' \\
&= \int_{x'=0}^{x} \begin{Bmatrix} 1 \text{ if } y > x' \\ y/x' \text{ if } y \leq x' \end{Bmatrix} dx' \\
&= \begin{cases} \int_{x'=0}^{x} 1\, dx' & \text{if } y > x \\ \int_{x'=0}^{y} 1\, dx' + \int_{x'=y}^{x} y/x'\, dx' & \text{if } y \leq x \end{cases} \\
&= \begin{cases} x & \text{if } y > x \\ y + y\log(x/y) & \text{if } y \leq x. \end{cases}
\end{aligned}$$

In situations like this, where the limits depend on what's being integrated, I like to write out all the cases together in the equation, using lots of $\{$ braces. It's a way to include all the "if statements" inside the maths equation, rather than separating out the algebra and the conditions. It helps me refactor freely. Here, for example, I rewrote the $\min(y/x',1)$ term as a case statement, then moved the case statement outside the integral.

Now we can differentiate to find the joint density.

$$\begin{aligned}
\Pr_{X,Y}(x,y) &= \frac{\partial}{\partial x}\frac{\partial}{\partial y} \begin{cases} x & \text{if } y > x \\ y + y\log(x/y) & \text{if } y \leq x \end{cases} \\
&= \frac{\partial}{\partial x} \begin{cases} 0 & \text{if } y > x \\ \log(x/y) & \text{if } y \leq x \end{cases} \\
&= \begin{cases} 0 & \text{if } y > x \\ 1/x & \text{if } y \leq x \end{cases} \\
&= \frac{1}{x} 1_{y \leq x}.
\end{aligned}$$

∎

Exercise 3.11 (Testing for independence).

I throw a fair die. Let Z be the result. Let $X = Z \bmod 2$ and let $Y = Z \operatorname{div} 3$, so for example $Z = 3$ gives $X = 1$ and $Y = 1$.

1. Show that X and Y are not independent.
2. Show that $X' = (Z-1) \bmod 2$ and $Y' = (Z-1) \operatorname{div} 2$ are.

The definition gives a condition that has to be satisfied for all x and y, namely that $\Pr_{X,Y}(x,y) = \Pr_X(x)\Pr_Y(y)$. Let's try some (x,y) values. First, a table of possible tuples:

Z:	1	2	3	4	5	6
X:	1	0	1	0	1	0
Y:	0	0	1	1	1	2

- *Try $x = 0$, $y = 0$. The only way to get $X = 0$ and $Y = 0$ is if $Z = 2$, so $\Pr_{X,Y}(0,0) = \mathbb{P}(Z = 2) = 1/6$. And $\Pr_X(0) = 1/2$ and $\Pr_Y(0) = 2/6$. So this pair passes the test.*
- *Try $x = 0$, $y = 1$. The only way to get $X = 0$ and $Y = 1$ is if $Z = 4$, so $\Pr_{X,Y}(0,1) = \mathbb{P}(Z = 4) = 1/6$. But $\Pr_X(0) = 3/6$ and $\Pr_Y(Y) = 3/6$. So the test fails.*

Thus X and Y are not independent. An exhaustive test of all x and y shows that X' and Y' are independent.

∎

Exercise 3.12 (Identifying independence by factorization).
Suppose that $P_{X,Y}(x,y) = g(x)h(y)$ for some functions g and h. Show that X and Y are independent, and

$$\Pr_X(x) \propto g(x), \quad \Pr_Y(y) \propto h(y).$$

* ✻ *

In probability theory, "independence" has a technical meaning involving terms of the density function: X and Y are independent if

$$\Pr_{X,Y}(x,y) = \Pr_X(x) \Pr_Y(y) \quad \text{for all } x, y.$$

In most situations this coincides with the general English meaning of the word, i.e. "unrelated" or "not depending on one another". But in the situation where there are unknown parameters, it's easy to be misled, as the following example illustrates.

Example 3.13. In the code

```
def rxy(p):
    # sample k = 2 independent Bin(1,p) random variables
    x,y = random.choices([0,1], weights=[1-p,p], k=2)
    return (x,y)
```

the joint density of X and Y is

$$\mathbb{P}(X=x,\ Y=y) = p^x(1-p)^{1-x}\, p^y(1-p)^{1-y}.$$

Obviously, learning the value of X tells us something about p (exercise: show that the maximum likelihood estimator for p given X is $\hat{p} = X$); and the distribution of Y obviously depends on p; so it's reasonable to say that X and Y are related—but nonetheless they are independent, because that's what random.choices outputs.

Whenever you hear "independent random variables", it's a good idea to whisper to yourself the coda "given their parameters", so you don't confuse "unrelated" and "independent".

3.5. Conditional random variables

The ‘learning’ in machine learning is ‘learning from data’. It’s important to be able to update a probability model in the light of data.

For example, suppose we’re modelling the climate with a simulator that reports two outputs, the level of CO_2, and some deep sea temperature reading. If we see that current CO_2 levels are a certain value 407.4 parts per million, how can we deduce the likely deep sea temperature? Or, if we want to model a scenario where CO_2 levels hit the range $[500, 510]$, how can we predict what the sea level temperatures are likely to be?

Here’s a toy example to illustrate how we might compute the latter case. Here X is the CO_2 levels and Y is the deep sea temperature. This code simulates (X, Y), and if X doesn’t conform to the scenario we want to consider it throws away the result and tries again.

```
# some simulator that produces (X,Y)
def rxy():
    x = random.random()
    z = random.random()
    return (x, x*z)

# sample Y, conditional on X lying in the range [a,b].
def ry_given_xrange(a,b):
    while True:
        x,y = rxy()
        if x >= a and x <= b:
            break
    return y
```

This is straightforward code. But its performance depends on how many (x, y) pairs have to be thrown away—and in the extreme case, where we want to condition on an exact value for X, it is not a good way to proceed. We need maths, not computation, to handle this.

In mathematical notation, we write $(Y \mid C)$ to denote the random variable Y *conditioned on an event* C. The distribution of $\tilde{Y} = (Y \mid C)$ is defined to be

$$\mathbb{P}(\tilde{Y} \in A) = \mathbb{P}(Y \in A \mid C) \quad \text{for all sets } A.$$

In the example above, if $\tilde{Y}$ is the random variable produced by ry_given_xrange(a,b), then we’d write it as $\tilde{Y} = (Y \mid X \in [a, b])$, and

$$\mathbb{P}(\tilde{Y} \in A) = \mathbb{P}(Y \in A \mid X \in [a, b]).$$

This $(Y \mid C)$ is a genuine random variable, and anything we can do with random variables we can also do with conditional random variables. For example, $(Y \mid C)$ has a density, written as $\Pr_Y(y \mid C)$—though perhaps it would be more logical to write it as $\Pr_{(Y \mid C)}(y)$. It has an expectation, written $\mathbb{E}(Y \mid C)$.

HANDLING CONDITIONAL RANDOM VARIABLES *

We can manipulate conditional random variables using all the tools described in section 3.3; just stick on “conditional on C” onto all the probabilities.

For the purposes of this course, the only manipulation that matters is Bayes’s rule for random variables, described in the next section. But for the sake of getting more comfortable with conditional random variables, here is an example calculation, finding the conditional density. If Y is discrete it’s straightforward—we just calculate $\Pr_Y(y \mid C)$ using conditional probability,

$$\Pr_Y(y \mid C) = \mathbb{P}(Y = y \mid C).$$

For continuous random variables we can find it by differentiating the cumulative distribution function, as in section 3.3 page 41:

$$\Pr_Y(y \mid C) = \frac{d}{dy} \mathbb{P}(Y \leq y \mid C).$$

Exercise 3.14 (Conditional density for continuous r.v.).
Let $X \sim \text{Exp}(\lambda)$. Find the density of $(X \mid X \leq a)$.

The Exponential has density $\Pr_X(x) = \lambda e^{-\lambda x}$ and cumulative distribution function $\mathbb{P}(X \leq x) = 1 - e^{-\lambda x}$. This and other standard distributions are listed in the appendix, page 62.

$$\begin{aligned}
\Pr_X(x \mid X \leq a) &= \frac{d}{dx}\,\mathbb{P}(X \leq x \mid X \leq a) \\
&= \frac{d}{dx}\,\frac{\mathbb{P}(X \leq x \textit{ and } X \leq a)}{\mathbb{P}(X \leq a)} \qquad \textit{by defn. of conditional probability} \\
&= \frac{d}{dx}\,\frac{\mathbb{P}\big(X \leq \min(x,a)\big)}{\mathbb{P}(X \leq a)} \\
&= \frac{d}{dx}\,\frac{1 - e^{-\lambda \min(x,a)}}{1 - e^{-\lambda a}} \\
&= \frac{d}{dx}\begin{cases} (1 - e^{-\lambda x})/(1 - e^{-\lambda}) & \textit{if } x \leq a \\ (1 - e^{-\lambda a})/(1 - e^{-\lambda}) & \textit{if } x > a \end{cases} \\
&= \begin{cases} \lambda e^{-\lambda x}/(1 - e^{-\lambda a}) & \textit{if } x \leq a \\ 0 & \textit{if } x > a. \end{cases}
\end{aligned}$$

It's intuitively obvious that the density should be 0 *for* $x > a$*, and it's reassuring to see how this drops out from the equations.*

∎

3.6. Bayes's rule for random variables

A common problem in 'learning from data' is the following: we have a simulator that generates an output X, which is fed into another simulator that generates another output Y; we make an observation $Y = y$; and we want to deduce what X was. A classic application of this is interpreting medical tests, and the way to solve it is with Bayes's rule. Bayes's rule says that for two events A and B, with $\mathbb{P}(B) > 0$,

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A)\ \mathbb{P}(B|A)}{\mathbb{P}(B)} = \frac{\mathbb{P}(A)\ \mathbb{P}(B|A)}{\mathbb{P}(A)\,\mathbb{P}(B|A) + \mathbb{P}(\neg A)\,\mathbb{P}(B|\neg A)}.$$

It derives from the definition of conditional probability,

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \text{ and } B)}{\mathbb{P}(B)}.$$

Example 3.15 (Bayes's rule for binary outcomes).
A screening test is 99% effective in detecting a certain disease when a person has the disease. The test yields a 'false positive' for 0.5% of healthy persons tested. Suppose 0.2% of the population has the disease. What is the probability that a person whose test is positive has the disease?

Let $X \in \{\textit{healthy}, \textit{sick}\}$ be the true health of the person, and let $Y \in \{\textit{positive}, \textit{negative}\}$ be the outcome of the test. The question tells us about the distribution of Y conditional on X:

$$\mathbb{P}\big(Y = \textit{positive} \mid X = \textit{sick}\big) = 0.99, \qquad \mathbb{P}\big(Y = \textit{positive} \mid X = \textit{healthy}\big) = 0.005,$$
$$\mathbb{P}\big(Y = \textit{negative} \mid X = \textit{sick}\big) = 0.01, \qquad \mathbb{P}\big(Y = \textit{negative} \mid X = \textit{healthy}\big) = 0.995.$$

It also tells us about the distribution of X absent any diagnostic information:

$$\mathbb{P}\big(X = \textit{sick}\big) = 0.002, \quad \mathbb{P}\big(X = \textit{healthy}\big) = 0.998.$$

Applying Bayes's to the events $A = \{X = \textit{sick}\}$ and $B = \{Y = \textit{positive}\}$,

$$\begin{aligned}&\mathbb{P}(X = \textit{sick} \mid Y = \textit{positive})\\ &= \frac{\mathbb{P}(X = \textit{sick})\,\mathbb{P}(Y = \textit{positive}|X = \textit{sick})}{\mathbb{P}(X = \textit{sick})\,\mathbb{P}(Y = \textit{positive}|X = \textit{sick}) + \mathbb{P}(X = \textit{healthy})\,\mathbb{P}(Y = \textit{positive}|X = \textit{healthy})}\\ &= \frac{0.002 \times 0.99}{0.002 \times 0.99 + 0.998 \times 0.005}.\end{aligned}$$

■

tl;dr. For two random variables X and Y, Bayes's rule says that

$$\Pr_X(x \mid Y = y) = \frac{\Pr_X(x)\,\Pr_Y(y \mid X = x)}{\Pr_Y(y)} = \frac{\Pr_X(x)\,\Pr_Y(y \mid X = x)}{\int_{x'} \Pr_X(x')\,\Pr_Y(y \mid X = x')\,dx'}.$$

As usual, replace the integral by a sum for discrete random variables. Bayes's rule derives from the conditional density of X given $\{Y = y\}$,

$$\Pr_X(x \mid Y = y) = \frac{\Pr_{X,Y}(x, y)}{\Pr_Y(y)} \quad \text{if } \Pr_Y(y) > 0.$$

This is just a straightforward application of Bayes's rule to the events $A = \{X = x\}$ and $B = \{Y = y\}$, in the case of discrete random variables. In the case of continuous random variables though, $\mathbb{P}(B = 0)$, and so the standard version of Bayes's rule doesn't apply; it takes some extra subtlety to prove Bayes's rule for continuous random variables.

The integral version of the denominator comes from writing $\Pr_Y$ as a marginal density and then rewriting as a conditional probability:

$$\Pr_Y(y) = \int_x \Pr_{X,Y}(x, y)\,dx = \int_x \Pr_Y(y \mid X = x)\,\Pr_X(x)\,dx.$$

3.6.1. APPLIED USING MATHEMATICS

In some rare situations, usually only seen in textbooks and exam questions, we can apply Bayes's rule using only mathematics. Typically we write it as

$$\Pr_X(x \mid Y = y) = \kappa \Pr_X(x) \Pr_Y(y \mid X = x) \quad \text{for some constant } \kappa.$$

We could of course calculate κ as an integral. But sometimes we're lucky and we don't have to—as we saw in section 3.2, the "densities sum to one" rule might let us work out κ without doing any calculus.

Exercise 3.16. Consider the pair of random variables (Θ, Y) where $\Theta \sim \text{Uniform}[0, 1]$ and $(Y \mid \Theta = \theta) \sim \text{Binom}(1, \theta)$. In other words, for $\theta \in [0, 1]$ and $y \in \{0, 1\}$,

$$\Pr_\Theta(\theta) = 1, \qquad \Pr_Y(y \mid \Theta = \theta) = \begin{cases} \theta & \text{if } y = 1 \\ 1 - \theta & \text{if } y = 0 \end{cases}$$

Find the distribution of $(\Theta \mid Y = 1)$.

For $\theta \in [0, 1]$,

$$\Pr_\Theta(\theta \mid Y = 1) = \kappa \Pr_\Theta(\theta) \Pr_Y(1 \mid \Theta = \theta) = \kappa \times 1 \times \theta.$$

This is where it's helpful to have out our fingertips a collection of standard random variables. The Beta distribution $\text{Beta}(a, b)$ *has density*

$$\Pr(x) = \binom{a + b - 1}{a - 1} x^{a-1}(1 - x)^{b-1} \quad \textit{for } x \in [0, 1].$$

The density we found for $(\Theta \mid Y = 1)$ *is* $\Pr_\Theta(\theta) = \kappa\theta$, *which is proportional to the density of a* $\text{Beta}(2, 1)$ *Since densities sum to one, there's no wiggle room for* κ*—it must simply be the same as the term at the front of* $\text{Beta}(2, 1)$. *In conclusion,* $(\Theta \mid Y = 1)$ is *a* $\text{Beta}(2, 1)$ *random variable.*

∎

3.6.2. APPLIED USING WEIGHTED MONTE CARLO

When working with random variables we often have a choice between exact calculations (often involving integrals) and computational approximations (based on sampling). For example, for a continuous random variable X, and any set A and any real-valued function h, if we have drawn a sample $(x_1, \dots, x_n)$ from X, then

This approximation is Monte Carlo integration, section 3.1

$$\mathbb{P}(X \in A) = \int_{x \in A} \Pr_X(x)\, dx \approx \frac{1}{n} \sum_{i=1}^{n} 1_{x_i \in A}$$

$$\mathbb{E}\, h(X) = \int_{x \in A} h(x) \Pr_X(x)\, dx \approx \frac{1}{n} \sum_{i=1}^{n} h(x_i).$$

The purpose of Bayes's rule is to find out about the conditional random variable $\tilde{X} = (X \mid Y = y)$. And the message of sampling-based approximation is that if we can generate samples of $\tilde{X}$ then we can learn whatever we like about its distribution. So, how can we sample from $\tilde{X}$?

Much work has been put into developing fast methods for sampling from conditional distributions. Such methods are covered in masters courses on advanced machine learning. Here is one simple method, based on weighted samples. It isn't the most efficient, but it's easy to use and doesn't depend on advanced theory.

tl;dr. Suppose we have a sample $(x_1, \dots, x_n)$ drawn from distribution X. To each value x_i in the sample, attach a weight

$$w_i = \frac{\Pr_Y(y \mid X = x_i)}{\sum_{j=1}^{n} \Pr_Y(y \mid X = x_j)}.$$

Then, use a weighted version of Monte Carlo integration to approximate probabilities and expectations:

$$\mathbb{P}(X \in A \mid Y = y) \approx \sum_{i=1}^{n} w_i 1_{x_i \in A}, \qquad \mathbb{E}\big(h(X) \mid Y = y\big) \approx \sum_{i=1}^{n} w_i h(x_i).$$

Exercise 3.17. For the pair of random variables (Θ, Y) in exercise 3.16, write a function that returns approximately $\mathbb{P}(\Theta \le \theta \mid Y = y)$, call it condprob($\theta$,$y$).

```
def condprob(θ,y,n=10000):
    # Generate a sample of size n from the distribution of Θ
    θsamp = numpy.random.uniform(low=0, high=1, size=n)
    # Define weights w_i proportional to Pr_Y(y | Θ = θsamp_i)
    w = (θsamp if y==1 else 1−θsamp)
    w = w / numpy.sum(w)
    # Return the weighted sum ∑ w_i 1[θsamp_i ≤ θ]
    return numpy.sum(w * numpy.where(θsamp <= θ, 1, 0))
```

∎

Derivation *. You don't need to know how to derive this approximation, you just need to know how to use it. But it's not very hard to derive: it comes simply from writing out the expectation we want as an integral, and approximating it with Monte Carlo integration. As we noted in section 3.1, the probability version is just the special case of $h(x) = 1_{x\in A}$, so we'll only derive the expectation version.

$$\begin{aligned}
&\mathbb{E}\big(h(X) \mid Y = y\big)\\
&\quad = \int_x h(x)\,\mathrm{Pr}_X(x \mid Y = y)\,dx \qquad \text{by the definition of expectation}\\
&\quad = \int_x h(x)\,\kappa\,\mathrm{Pr}_X(x)\,\mathrm{Pr}_Y(y \mid X = x)\,dx \qquad \text{by Bayes's rule, where } \kappa \text{ is a constant}\\
&\quad = \int_x g(x)\,\mathrm{Pr}_X(x)\,dx \quad \text{where } g(x) = h(x)\,\kappa\,\mathrm{Pr}_Y(y \mid X = x)\\
&\quad = \mathbb{E}\,g(X)\\
&\quad \approx \frac{1}{n}\sum_{i=1}^{n} g(x_i) \quad \text{by Monte Carlo integration.}
\end{aligned}$$

The constant κ is there to make the conditional density sum to one:

$$\begin{aligned}
\kappa &= 1 \Big/ \int_x \mathrm{Pr}_X(x)\,\mathrm{Pr}_Y(y \mid X = x)\,dx\\
&= 1 \Big/ \int_x f(x)\,\mathrm{Pr}_X(x)\,dx \quad \text{where } f(x) = \mathrm{Pr}_Y(y \mid X = x)\\
&= 1 \big/ \mathbb{E}\,f(X)\\
&\approx 1 \Big/ \frac{1}{n}\sum_{i=1}^{n} f(x_i) \quad \text{by Monte Carlo integration.}
\end{aligned}$$

Putting these two approximations together,

$$\mathbb{E}\big(h(X) \mid Y = y\big) \approx \frac{1}{n}\sum_{i=1}^{n} \kappa\, h(x_i)\,\mathrm{Pr}_Y(y \mid X = x_i) \approx \frac{\sum_{i=1}^{n} h(x_i)\,\mathrm{Pr}_Y(y \mid X = x_i)}{\sum_{i=1}^{n} \mathrm{Pr}_Y(y \mid X = x_i)}$$

4. Empirical methods

The word ‘empirical’ means ‘based from observation’. In classical Greece, there were two schools of medicine, the empiric and the rational. Rational physicians held that treatments and explanations of disease should be based on deduction from theoretical principles of how the body works. At that time the standard theory was based on the four humours, blood, phlegm, yellow bile, and black bile. Empirics on the other hand based their treatments on what they had seen to work in the past. The empiricists were considered to be inferior physicians, peddling in folk remedies (“my neighbour swears by a frog skin to cure a sore throat, worn in a pouch around the neck”) without any real understanding of the disease.

In this section we will look at probability models that take the dataset to be the ground truth. This is as opposed to all the simulations and calculations in section 3, which take as their starting point a piece of simulator code or a mathematical equation. In data science, as in any science, “all models are wrong”[12], and so it’s useful to see how far we can go without even writing down any simulator code or mathematical description.

* ✻ *

From the point of view of this course, the concept of the empirical distribution (section 4.2) is fundamental for frequentist inference in Part III, and it’s also the big idea behind cross-validation. There’s virtually no maths, but the concept is subtle, hence the build-up (section 4.1).

[12] G. E. P. Box. “Robustness in the Strategy of Scientific Model Building”. In: *Robustness in Statistics*. Vol. 1. May 1979, p. 40. URL: `http://www.dtic.mil/docs/citations/ADA070213`. Box has been described as “one of the great statistical minds of the 20th century”. The full quotation:

> *All models are wrong but some are useful [...] there is no need to ask the question “Is the model true?” If “truth” is to be the “whole truth” the answer must be “No”. The only question of interest is “Is the model illuminating and useful?”*

4.1. The empirical cumulative distribution function

Given a dataset of n numerical values $(x_1, x_2, \ldots, x_n)$, the *empirical cumulative distribution function* of the dataset is

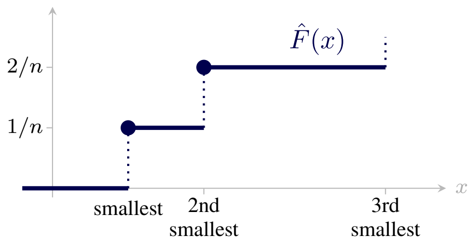
$$\hat{F}(x) = \frac{1}{n}\big(\text{how many items there are } \leq x\big).$$

This parallels the cumulative distribution function for a random variable X,

$$F(x) = \mathbb{P}(X \leq x).$$

PLOTTING THE ECDF

It's easy to plot the empirical distribution function: just sort the data and put it on the x-axis.



The convention for matplotlib plotting is to `import matplotlib.pyplot as plt`

In Python, using matplotlib, here is code to plot a histogram and the ecdf.

```
x = [...]   # the dataset, stored as a list
ef = numpy.arange(1,len(x)+1)/len(x)
fig,(ax1,ax2) = plt.subplots(1,2, figsize=(8,4))
ax1.hist(x, bins=50)
ax2.plot(numpy.sort(x), ef, drawstyle='steps-post')
plt.show()
```

What's nice about the ecdf, as opposed to a histogram, is that it shows every single datapoint and it doesn't rely on an arbitrary choice of bin size. Also, if you want to show more detail for example by using a log scale, you don't need to mess around with bothersome details like "do I take the log before or after binning?"

WHAT A GOOD FIT LOOKS LIKE

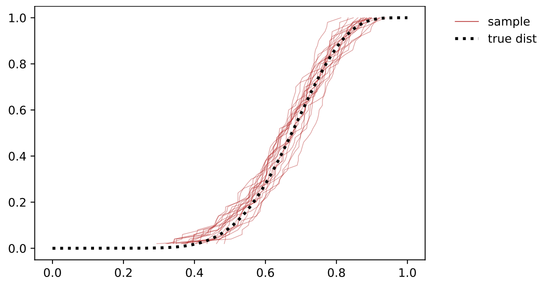
If a particular random variable X truly were the distribution from which the x_i were drawn, then we'd expect $F(x) \approx \hat{F}(x)$. (This can be justified in several ways. For example, from the Monte Carlo method for approximating a probability,

$$\mathbb{P}(X \leq x) \approx \frac{1}{n}\sum_{i=1}^{n} 1_{x_i \leq x}.$$

In this approximation, the left hand side is $F(x)$ and the right hand side is $\hat{F}(x)$.)

Here's an illustration, 20 random samples each of size 50 drawn from the Beta$(10,5)$ distribution. For each sample we plot its empirical distribution.

Beta distribution: appendix page 64



```
α,β = 10,5
fig,ax = plt.subplots(figsize=(6,4))
```

```

# Plot the ecdf, 20 times
for _ in range(20):
    x = numpy.random.beta(α, β, size=50)
    ef = numpy.arange(1,len(x)+1)/len(x)
    plt.plot(numpy.sort(x), ef, alpha=0.5, color='firebrick', linewidth=.5)
# A hack to force a legend entry for the samples
plt.plot([], [], color='firebrick', linewidth=.6, label='sample')

# Plot the true cdf
x = numpy.linspace(0,1,1000)
f = scipy.stats.beta.cdf(x, alpha, beta)
plt.plot(x, f, color='black', linestyle='dotted', linewidth=2.5, label='true dist')

plt.legend(bbox_to_anchor=(1.05, 1), loc=2, frameon=False)
plt.show()
```

4.2. The empirical distribution

If we're programming a simulator as a model of some real-world system, we'll want to use random number generators that are a good fit for real-world data. Here's a simple way to do this:

tl;dr. Given a dataset $(x_1, \dots, x_n)$, let X^* be the random variable obtained by picking one of the x_i at random. This is a discrete random variable taking values in $\{x_1, \dots, x_n\}$. Its distribution is

$$\mathbb{P}(X^* \in A) = \frac{1}{n}\bigl(\text{number of datapoints } x_i \text{ that are} \in A\bigr)$$

or equivalently

$$\Pr_{X^*}(x) = \frac{1}{n}\bigl(\text{number of datapoints } x_i \text{ that are equal to } x\bigr).$$

This is called the *empirical distribution* of the dataset.

In Python, generate a single value from X^* with random.choice(x) and generate k independent values with

```
numpy.random.choice(x, replace=True, size=k)
```

The empirical distribution is a perfect fit to the dataset. Consider the cumulative distribution function[13] for X^*: it's simply

$$\mathbb{P}(X^* \le x) = \frac{1}{n}\bigl(\text{number of datapoints } x_i \text{ that are} \le x\bigr).$$

This is precisely the same as the *empirical* cumulative distribution function for the dataset, $\hat{F}(x)$, defined in section 4.1. We suggested in that section that a good-fitting distribution is one whose cdf matches the ecdf; by that criterion X^* is perfect.

EMPIRICAL OR PARAMETRIC MODELLING?

We saw a different approach for fitting a random number generator in section 1.5. The approach in that section was (1) pick a family of random variables, one that has parameters; (2) use maximum likelihood estimation to estimate the parameters, so as to fit the dataset. The empirical approach is (1) use the empirical distribution, (2) that's it.

When should we use parametric models and when should we use the dataset itself? There are no general rules.

- If all we have is a dataset, and no one has told us which family of random variables to use, isn't it daft to shoehorn ourselves into some standard random variable—especially when a perfect fit to the data is staring us in the face, namely the empirical distribution?

  Conversely, if we have general scientific knowledge about the domain we're trying to model, and if that knowledge tells us what family of random variables is appropriate, then it makes perfect sense to use a parametric model.

- A dataset cannot tell us about values beyond the dataset. This has to come from our background knowledge or intuition. If we believe that values beyond the dataset are plausible, then we need to find a way to express this belief as a parametric model.

- A parametric distribution saves space: it only needs us to store a handful of parameters, rather than the full dataset.

  But this is often a premature optimization. For a small dataset of a few tens of thousands of values, on a modern computer, you should spend your time thinking about modeling and not

[13]This argument, based on the cumulative distribution function, only makes sense for numerical data. However, the definition of X^* applies to random variables taking values in any set. See section 4.3 for a discussion of how to measure closeness of distributions for arbitrary random variables.

about optimizing storage. For a large dataset, a model with a handful of parameters cannot hope to capture the richess of the data.

- Neural networks are parametric models. A neural network trained for simple image classification might take 140 million parameters, one for each connection in the network. The human brain has roughly 10^{15} connections, and a human lifetime is roughly 2.5×10^9 seconds. It seems that making sense of data is more about what you do with it than how you can compress it.

  High-dimensional modeling, i.e. modeling with more parameters than there are samples in the dataset, is an area of active research.

MONTE CARLO AND THE EMPIRICAL DISTRIBUTION

Monte Carlo integration is a way to approximate an expectation,

$$\mathbb{E}\, h(X) \approx \frac{1}{n} \sum_{i=1}^{n} h(x_i)$$

where $x_1, \ldots, x_n$ is a sample drawn from distribution X and h is some arbitrary real-valued function. Equivalently, if we let X^* be a random value drawn from the sample,

$$\int_x h(x)\, \mathrm{Pr}_X(x)\, dx \approx \sum_x h(x)\, \mathrm{Pr}_{X^*}(x) = \mathbb{E}\, h(X^*).$$

In other words, Monte Carlo approximation consists in replacing the distribution of a random variable X with the empirical distribution of a sample drawn from X.

THE ZEN OF DATA SCIENCE *

The zen of data science is in seeing datasets and random variables as two sides of the same coin.

- When the true distribution is intractable, we can approximate it by the empirical distribution of a random sample. Instead of getting bogged down with integrals, we just use Monte Carlo integration.
- When the true distribution is unknown, we can use the dataset's empirical distribution instead. There's no need to approximate the dataset by fitting a standard random variable, when we can just pick items randomly from the dataset.

An earlier example gave code for computing the "marginal distribution of survival on the Titanic". But a dataset is a collection of numbers, not a random variable, and it is only random variables that have distributions—so what was the example asking for?

exercise 3.6 page 43

Behind that example there is a very subtle idea. When we talk about the "distribution" of a collection of values, what we're actually referring to is the empirical distribution. Similarly, when we talk about the "variance" of a collection of numbers, we mean the variance of a random X^* drawn from that collection. Or if we want to know whether gender and survival on the Titanic were independent, we have to imagine picking a random individual from the dataset, extracting the random tuple $(\mathsf{Gender}^*, \mathsf{Survival}^*)$ for that individual, and asking whether the two constituent random variables Gender^* and $\mathsf{Survival}^*$ are independent.

4.3. Goodness of fit and KL divergence *

In generative modelling, section 1.5, we start with a dataset and we look for a probability distribution that might have generated it. Typically we restrict our search to a family of distributions that has one or more tuneable parameters; selecting the best-fitting parameter is called 'fitting the distribution'.

How should we measure how good the fit is? In modelling, as in any engineering task, we ought to find or invent some sensible metric by which to judge whether we've been successful. There is one metric that you're likely to come across in advanced machine learning, the *Kullback-Leibler divergence*.

tl;dr. Let $\Pr_{X^*}(x)$ be the density of the empirical distribution of a dataset, and let $\Pr_X(x \mid \theta)$ be the proposed generative distribution, with parameter θ. The KL divergence from $\Pr_{X^*}(\cdot)$ to $\Pr_X(\cdot \mid \theta)$ is

$$\mathrm{KL}\Big(\Pr_{X^*}(\cdot) \;\Big\|\; \Pr_X(\cdot \mid \theta)\Big) = \sum_x \Pr_{X^*}(x) \log \frac{\Pr_{X^*}(x)}{\Pr_X(x \mid \theta)}$$

where the sum is over the support of X^*, i.e. over the values in the dataset.

(The definition can be applied to any pair of distributions, though for this course we only care about the case where X^* is the empirical distribution of a dataset. In the general case, if X^* is a continuous random variable, replace the sum by an integral.)

This intimidating-looking equation, it turns out, is nothing more than log likelihood dressed up in fancy clothing. Before explaining where it comes from, let's do a sanity check on the equation.

- The KL divergence is in the range $[0, \infty)$. There's a slick algebraic proof that you can find online (search for the key phrase "Jensen's inequality"). For the intuitive reason, see below.
- If it's a perfect fit, i.e. if $\Pr_X(x \mid \theta) = \Pr_{X^*}(x)$ for all x, then $\mathrm{KL} = 0$.
- If there is a datapoint in the dataset for which $\Pr_X(x \mid \theta) = 0$ then the denominator is zero and we take KL to be ∞. The interpretation is "the dataset contains a value x, the proposed model says that x is impossible, therefore the proposed model is rubbish".

WHERE DOES IT COME FROM?

All the probabilistic learning techniques from section 1 are based on likelihood, which is nothing more than the probability of observing the dataset. Let the dataset be $(x_1, \dots, x_n)$, and to streamline the notation define

$$p_x = \Pr_X(x \mid \theta), \quad q_x = \Pr_{X^*}(x), \quad n_x = n q_x = \text{number of occurrences of } x.$$

The likelihood of the dataset, assuming it consists of independent samples from X, is

$$\log \mathrm{lik}_X = \log\Big(\prod_{i=1}^n \Pr_X(x_i \mid \theta)\Big) = \sum_{i=1}^n \log p_{x_i} = \sum_x n_x \log p_x.$$

This isn't ideal as a metric of how good a fit X is, because it scales in proportion to n, the size of the dataset. A good metric should be independent of the size of the dataset. We get this by rescaling,

$$\frac{1}{n} \log \mathrm{lik}_X = \sum_x q_x \log p_x.$$

Furthermore, it's useful to have a reference point. The best possible fit to a dataset is its empirical distribution, so a sensible reference point is

$$\frac{1}{n} \log \mathrm{lik}_{X^*} = \sum_x q_x \log q_x.$$

The KL divergence is simply the difference between these two,

$$\mathrm{KL}(q \parallel p) = \frac{1}{n} \log \mathrm{lik}_{X^*} - \frac{1}{n} \log \mathrm{lik}_X = \sum_x q_x \log \frac{q_x}{p_x}$$

The perfect fit is when $X \sim X^*$, giving divergence 0. The worse the fit, the smaller lik_X will be, and so the bigger the divergence.

The quantity $-\sum_x q_x \log_2 p_x$ has another interpretation in information theory. It measures the average number of bits needed to transmit a value drawn randomly from the dataset, using a coding scheme designed under the assumption that values are drawn from distribution p_X. (If p_X is a bad fit for the dataset, then the coding scheme will optimize the wrong thing, hence we'll need more bits.) So think of KL as measuring "wasted bits due to imperfect model fit".

Appendix

A. Standard random variables

A.1. Python library commands

In Python, numpy and scipy.stats have useful functions for working with random variables. They have a consistent naming convention, shown here for the Normal distribution.

numpy.random.normal(..., size=n)
: Generate n independent random variables from the Normal distribution. The ... are parameters, different for each distribution.

scipy.stats.norm.pdf(x=x, ...)
: the probability density function $\Pr(x)$

scipy.stats.norm.cdf(x=x, ...)
: the cumulative distribution function $\mathbb{P}(X \leq x)$

scipy.stats.norm.ppf(q=q, ...)
: the inverse of the cumulative distribution function, returns x such that $\mathbb{P}(X \leq x) = q$;
: for discrete random variables, when cdf jumps up in steps, returns $\min\{x : \mathbb{P}(X \leq x) \geq q\}$

scipy.stats.norm.mean(...), median, var, std
: summaries of the distribution

Data science computation often involves small probabilities, so watch out for bugs arising from numerical overflow and underflow. It's usually a good idea to work with log probabilities and with the *survival function* sf$(x) = \mathbb{P}(X > x)$.

scipy.stats.norm.logpdf(x, ...)
: $\log \Pr(x)$

scipy.stats.norm.logcdf(x, ...)
: $\log \mathbb{P}(X \leq x)$

scipy.stats.norm.sf(x, ...), logsf
: $\mathbb{P}(X > x)$ and $\log \mathbb{P}(X > x)$

A.2. List of common random variables

Geometric: If we're playing a lottery, and each week the chance of winning is p, then our first win happens on week $X \sim \text{Geom}(p)$. This random variable takes values in $\{1, 2, \dots, n\}$, and

$$\mathbb{P}(X = r) = (1-p)^{r-1}p, \quad \mathbb{P}(X \geq r) = (1-p)^{r-1}.$$

Mean $1/p$, variance $(1-p)/p^2$. In Python, numpy.random.geometric(p).

Exponential: The Exponential random variable is a continuous-time version of the Geometric. It's used to model the time until an event, for many natural processes: for example the time until a lump of radioactive matter emits its next particle, or the time until a lightbulb blows, or the time until the next web request arrives. If $X \sim \text{Exp}(\lambda)$ then it takes values in $[0, \infty)$, and

$$\text{Pr}(x) = \lambda e^{-\lambda x}, \quad \mathbb{P}(X \geq x) = e^{-\lambda x}.$$

The parameter λ is called the *rate*. The chance of an event in a short interval of time $[t, t+\delta]$ is

$$\mathbb{P}(X \leq t + \delta \mid X \geq t) = \frac{\mathbb{P}(X \in [t, t+\delta])}{\mathbb{P}(X \geq t)} = \frac{\int_t^{t+\delta} \lambda e^{-\lambda x}\, dx}{e^{-\lambda t}} \approx \delta\lambda.$$

Mean $1/\lambda$, variance $1/\lambda^2$. In Python, numpy.random.exponential(scale=$1/\lambda$).

Binomial: If we toss a biased coin n times, and each coin has chance p of heads, the total number of heads is $X \sim \text{Binom}(n, p)$. This random variable takes values in $\{0, 1, \dots, n\}$, and

$$\mathbb{P}(X = r) = \binom{n}{r} p^r (1-p)^{n-r}.$$

When $n = 1$, i.e. a single coin toss, it's called a Bernoulli random variable.

Mean np, variance $np(1-p)$. In Python, numpy.random.binomial(n,p).

Multinomial: If we have n individuals each of whom falls into one of K categories, and the probability of falling into category k is p_k, then the total number in each category is a multivariate random variable $X \sim \text{Multinom}(n, p)$. It takes values in $\{0, 1, \dots, n\}^K$, and

$$\mathbb{P}(X = x) = \frac{n!}{x_1! x_2! \cdots x_K!} p_1^{x_1} p_2^{x_2} \cdots p_K^{x_K}.$$

(The binomial distribution is the special case when $k = 2$.)

In Python, numpy.random.multinomial(n,p).

Poisson: The random variable $X \sim \text{Poisson}(\lambda)$ takes values in $\{0, 1, \dots\}$, and

$$\mathbb{P}(X = r) = \frac{\lambda^r e^{-\lambda}}{r!}.$$

Suppose we're counting the number of events in a fixed interval of time, for example the number of buses passing a spot on the street, or the number of web requests, or the number of particles emitted by a lump of radioactive matter. If the time between events is $\text{Exp}(\lambda)$, then the total number of events in time t is $X \sim \text{Poisson}(\lambda t)$.

Mean λ, variance λ. In Python, numpy.random.poisson(lam=λ).

Normal / Gaussian: This distribution is a very popular choice for data analysis because it's often a good model for things that are the aggregate of many small pieces, for example height which is the aggregate of many influences from genetics and the environment. It's also easy to do probability calculations with it. If $X \sim \text{Normal}(\mu, \sigma^2)$, then X is a continuous random variable taking values in the entire real line, and

$$\Pr(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}, \quad \mathbb{E}\, X = \mu, \quad \text{Var}\, X = \sigma^2.$$

There is also a multivariate version, called the multivariate normal. Here are some useful facts about the Normal distribution. If $X \sim \text{Normal}(\mu, \sigma^2)$, and $Y \sim \text{Normal}(\nu, \rho^2)$ is independent, and a and b are real numbers, then

$$\begin{aligned}
&\mathbb{P}\big(\mu - 1.96\sigma \le X \le \mu + 1.96\sigma\big) = 95\% \\
&aX + b \sim \text{Normal}(a\mu + b, a^2\sigma^2) \\
&(X - \mu)/\sigma \sim \text{Normal}(0, 1) \\
&X + Y \sim \text{Normal}\big(\mu + \nu, \sigma^2 + \rho^2\big)
\end{aligned}$$

In Python, numpy.random.normal(loc=μ, scale=σ), and watch out for σ versus σ^2!

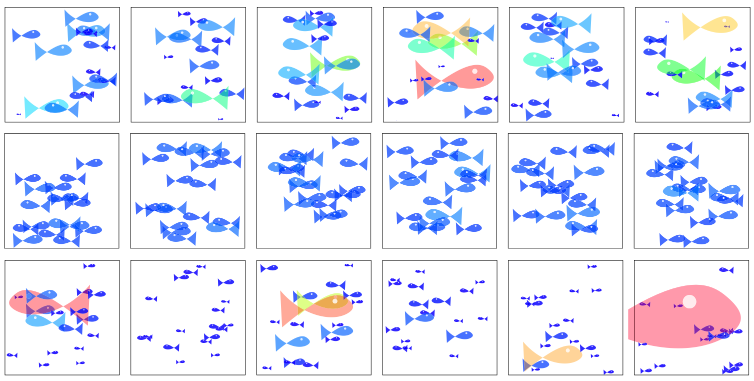
Pareto and lognormal: Some natural phenomena, like sizes of forest fires, or insurance claims, or Internet traffic volumes, or stock market crashes, have the characteristic that there are events of wildly different sizes. This tends to cause problems for simulations and forecasting, since the entire outcome can hinge on one 'black swan' event[14]. A common random variable with this characteristic is the Pareto distribution, $X \sim \text{Pareto}(\alpha)$, named after the Italian economist Vilfredo Pareto who studied extreme wealth inequality. It is a continuous random variable taking values in $[1, \infty)$, and

$$\Pr(x) = \alpha x^{-(\alpha+1)}, \quad \mathbb{P}(X \ge x) = x^{-\alpha}.$$

The mean and variance become ∞ for small α,

$$\mathbb{E}\, X = \begin{cases} \infty \text{ if } \alpha \le 1 \\ \alpha/(\alpha - 1) \text{ otherwise,} \end{cases} \qquad \text{Var}\, X = \begin{cases} \infty \text{ if } \alpha \le 2 \\ \alpha \;/\; (\alpha-1)^2(\alpha-2)^2 \text{ otherwise.} \end{cases}$$

For $\alpha < 2$ it tends to produce many small values ('mice') and very occasional huge values ('elephants'). To illustrate, here are some samples drawn from three different distributions, all with mean value 1.



$X \sim \text{Exp}(1)$,
$\mathbb{E}\, X = 1$

$X \sim \frac{\alpha-1}{\alpha} \text{Pareto}(\alpha)$
with $\alpha = 5$,
$\mathbb{E}\, X = 1$

$X \sim \frac{\alpha-1}{\alpha} \text{Pareto}(\alpha)$
with $\alpha = 1.1$,
$\mathbb{E}\, X = 1$

The lognormal distribution $X \sim e^{N(\mu,\sigma^2)}$ has similar characteristics to the Pareto but is not quite as extreme. It was invented by the Cambridge senior wrangler and medic Donald MacAlister.

Zipf: The random variable $X \sim \text{Zipf}(n, s)$ takes values in $\{1, 2, \dots, n\}$ and

$$\mathbb{P}(X = r) = \frac{r^{-s}}{1 + 2^{-s} + \cdots + n^{-s}}.$$

It is named after the American linguist Goerge Zipf, who used it to describe frequencies of words in texts. Take a large piece of text, and count the number of occurrences of each word, and rank the

[14] Nassim Nicholas Taleb. *The Black Swan: The Impact of the Highly Improbable*. 2nd ed. Random House, 2010.

words from most common to least common. Say that the most common word has rank 1, the next most common has rank 2, and so on. Zipf observed that the number of occurrences of the rth ranked word is roughly const $\times\, r^{-s}$ where $s \approx 1$ in English texts. Another way of putting this: if we pick a word at random from the entire body of text, then the rank of that word is Zipf(n, s), where n is the size of the vocabulary. The same phenomenon happens with cities: if we take a person at random from the entire population, and look at which city they come from, and rank cities by size, then the rank of that person's city is Zipf(n, s) where n is the number of cities and s is roughly 1.07.

There is a direct link between the Pareto(α) and Zipf$(n, 1/\alpha)$ distributions. First, create a 'pseudo-random' sample of n city sizes, to match the Pareto(α) distribution. Make the largest city have size $x_{(1)}$ such that $x_{(1)}^{-\alpha} = 1/N$, make the second-largest city have size $x_{(2)}$ such that $x_{(2)}^{-\alpha} = 2/N$, etc. This is a deterministic equivalent of the Pareto(α) distribution, in which $\mathbb{P}(X \geq x) = x^{-\alpha}$. Then, the city of rank r has size const $\times\, r^{-1/\alpha}$, which fits with Zipf$(n, 1/\alpha)$.

Beta: If we toss a biased coin n times, and each coin has chance p of heads, then the number of heads has a Bin(n, p) distribution. In Bayesian inference, a common prior distribution for p is Beta(α, β). It takes values in $(0, 1)$, and has parameters $\alpha > 0$ and $\beta > 0$, and density

$$\Pr(p) = \binom{\alpha+\beta-1}{\alpha-1} p^{\alpha-1}(1-p)^{\beta-1}$$

(but with a generalized form of the binomial coefficient when α and β are non-integer). It has mean $\alpha/(\alpha + \beta)$, and the rough interpretation is "I've seen α heads and β tails".

In Python, numpy.random.beta(a=α, b=β).

Dirichlet: The Dirichlet distribution Dir(α) is a generalization of the Beta distribution. Instead of two categories (heads and tails), it allows $K \geq 2$ categories, and α is a vector in $\mathbb{R}^K$. It is a continuous random variable, and it takes values in

$$\Omega = \big\{[x_1, \dots, x_K] \in (0,1)^K \;:\; x_1 + \cdots + x_K = 1\big\}.$$

In other words, it generates probability distributions over the K categories. It is used in Bayesian inference to describe belief about a multinomial distribution, and the rough interpretation is "I've seen α_k items in category k". Its density function is

$$\Pr\big([x_1, \dots, x_K]\big) \propto x_1^{\alpha_1 - 1} x_2^{\alpha_2 - 1} \cdots x_K^{\alpha_K - 1}.$$

In Python, numpy.random.dirichlet(alpha=α).

Gamma: The Gamma distribution $X \sim \Gamma(k, \lambda)$ is a continuous random variable taking values in $[0, \infty)$, and its parameters are $k > 0$ and $\lambda > 0$. It arises in two places: it's the sum of k independent Exponential random variables; and it's a common choice of prior distribution for $1/\sigma^2$ in Bayesian calculations with Normal(μ, σ^2) random variables. (Engineers call $1/\sigma^2$ the 'precision'.) It has density

$$\Pr(x) = \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}$$

(but with $(k-1)!$ replaced by the gamma function $\Gamma(k)$ for non-integer k).

Mean $k\lambda$, variance k/λ^2. In Python, numpy.random.gamma(shape=k, scale=$1/\lambda$).

B. Abstract linear mathematics

B.1. Definitions and useful properties

- Let V be a set whose elements are called *vectors*, denoted by Roman letters[15] u, v, w, etc.
- Let F be a field whose elements are called *scalars*, denoted by Greek letters λ, μ, etc. For our purposes, take F to be either the real numbers or the complex numbers.
- Let there be a binary operation $V \times V \to V$, called *addition*, written $v + w$.
- Let there be a binary operation $F \times V \to V$, called *scalar multiplication*, written λv.
- Let there be a binary operation $V \times V \to F$, called *inner product*, written $v \cdot w$.

Vector space. V is called a *vector space* over F if the following properties hold:

1. Associativity: $(u + v) + w = u + (v + w)$ for all vectors u, v, w.
2. Commutativity: $u + v = v + u$ for all vectors u, v
3. Zero vector: there is a vector 0 such that $v + 0 = v$ for all vectors v
4. Inverse: for every vector v there is a vector denoted $-v$ such that $v + (-v) = 0$
5. $\lambda(v + w) = \lambda v + \lambda w$ for every scalar λ and vectors v, w
6. $(\lambda + \mu)v = \lambda v + \mu v$ and $(\lambda\mu)v = \lambda(\mu v)$ for all scalars λ, μ and vector v
7. $1v = v$ for every vector v, where 1 is the unit scalar (i.e. $1\lambda = \lambda$ for every scalar λ).

Linear combinations and bases. Let $v_1, \ldots, v_n$ be vectors in a vector space and $\lambda_1, \ldots, \lambda_n$ be scalars. Then the vector $\lambda_1 v_1 + \cdots + \lambda_n v_n$ is called a *linear combination* of $v_1, \ldots, v_n$. The set of all linear combinations

$$S = \{\lambda_1 v_1 + \cdots + \lambda_n v_n \ : \ \lambda_i \in F \text{ for all } i\}$$

is called the *span* of $\{v_1, \ldots, v_n\}$, and the vectors v_i are said to *span* S. Clearly $S \subseteq V$, and it is not hard to check that S is also a vector space. It is called a *subspace* of V.

Vectors $v_1, \ldots, v_n$ in a vector space are said to be *linearly independent* if

$$\lambda_1 v_1 + \cdots + \lambda_n v_n = 0 \quad \Longrightarrow \quad \lambda_1 = \cdots = \lambda_n = 0.$$

If this is not the case, then they are said to be *linearly dependent*.

If there is a finite set of vectors $e_1, \ldots, e_n$ that span a vector space V, and they are linearly independent, then they are called a *basis* for V. It can be shown that any two bases for a vector space must have the same number of elements; this number is called the *dimension* of the vector space.

Given a basis $\{e_1, \ldots, e_n\}$ of a vector space, it can be proved that any vector x can be uniquely written as

$$x = \lambda_1 e_1 + \cdots + \lambda_n e_n \quad \text{for some scalars } \lambda_1, \ldots, \lambda_n.$$

The n-tuple $(\lambda_1, \ldots, \lambda_n)$ is called the *coordinates* of x with respect to the given basis. If we pick a different basis we'll get different coordinates, but of course the vector x itself is still the same regardless of the basis.

Inner products and orthogonality. Consider a vector space V over the field of real numbers. It is said to be an *inner product space* if the inner product satisfies these properties:

8. $v \cdot v \geq 0$ for all vectors v, and $v \cdot v = 0$ if and only if $v = 0$
9. $(\lambda u + \mu v) \cdot w = \lambda(u \cdot w) + \mu(v \cdot w)$ for all vectors u, v, w and scalars λ, μ
10. $v \cdot w = w \cdot v$ for all vectors v and w

An inner product space over the field of complex numbers is defined similarly, except that condition 10 is replaced by $v \cdot w = \overline{w \cdot v}$ where $\overline{\lambda}$ is the complex conjugate of the complex number λ. Also, the first part of condition 8 should be interpreted as $\mathrm{Im}(v \cdot v) = 0$ and $\mathrm{Re}(v \cdot v) \geq 0$.

Two vectors v and w in an inner product space are said to be *orthogonal* if $v \cdot w = 0$. A set of vectors (which may be finite or infinite) is said to be an *orthogonal system* if every pair of vectors in the set is orthogonal and in addition none of them is equal to 0.

The *Euclidean norm* for an inner product space is

$$\|v\| = \sqrt{v \cdot v}.$$

[15] In introductory geometry it's common to use bold symbols for vectors, e.g. $\mathbf{v} + \mathbf{0} = \mathbf{v}$ and $1\mathbf{v} = \mathbf{v}$. This notation makes it clear that $\mathbf{0}$ is a vector and 1 is a scalar. The bold notation is less common in more advanced applications, so you have to rely on type inference to spot that 0 is a vector and 1 is a scalar.

A vector v with $\|v\| = 1$ is called a *unit vector*. An orthogonal system is said to be an *orthonormal system* if every vector in it is a unit vector.

Useful properties. Here are some useful properties that can be proved from the abstract definitions. They are mostly obvious when we're working with finite dimensional Euclidean space. For abstract vector spaces, they must be proved directly from the defining properties 1–10. The proofs are just careful definition-pushing, but it's reassuring to know that it can be done.

11. $0v = 0$, for every vector v in a vector space.
12. $(-\lambda)v = -(\lambda v)$, for every vector v in a vector space and every scalar λ.
13. $(\lambda v) \cdot w = \lambda(v \cdot w)$, for all scalars λ and vectors v, w in an inner product space.
14. $0 \cdot v = 0$, for every vector v in an inner product space.
15. For all n and all scalars $\lambda_1, \dots, \lambda_n$ and vectors $v_1, \dots, v_n, w$ in an inner product space,
$$\Big(\sum_{i=1}^{n} \lambda_i v_i\Big) \cdot w = \sum_{i=1}^{n} \lambda_i (v_i \cdot w).$$
16. If $\{e_1, \dots, e_n\}$ is an orthonormal system in an inner product space, then for every vector x in the span of $\{e_1, \dots, e_n\}$, the coordinates of x are given by
$$x = \sum_{i=1}^{n} (x \cdot e_i)\, e_i.$$
17. $\|u + v\| \le \|u\| + \|v\|$ for all vectors u, v; this is known as the *triangle inequality*.

Exercise B.1. Prove useful property 11

In this equation, the left hand side must be referring to the scalar $0 \in F$ *and the right hand side to the vector* $0 \in V$*, where* V *is the vector space over field* F*, because otherwise the equation doesn't make sense—the abstract definitions don't define multiplication of vectors, and scalar multiplication yields a vector.*

In both the real numbers and the complex numbers (and indeed in any field F*),* $0 = 0 + 0$*. So, by property 6,*
$$0v = (0 + 0)v = 0v + 0v.$$
By property 4, there is some vector $-(0v)$ *such that* $0v + \big(-(0v)\big) = 0$*. Adding this to each side of the equation,*
$$0v + \big(-(0v)\big) = \big(0v + 0v\big) + \big(-(0v)\big)$$
and so, using property 1,
$$0 = 0v + \big(0v + (-(0v))\big) = 0v + 0.$$
Finally, by property 3,
$$0 = 0v.$$

∎

Exercise B.2. Prove useful property 12

Property 6 says that
$$\lambda v + (-\lambda)v = \big(\lambda + (-\lambda)\big)v.$$
In both the real numbers and the complex numbers (and indeed in any field F*),* $\lambda + (-\lambda) = 0 \in F$*, thus*
$$\lambda v + (-\lambda)v = 0v$$
which we showed in the previous exercise to be equal to $0 \in V$*. So* $(-\lambda)v$ *satisfies property 4 and it is therefore* $-(\lambda v)$*.*

∎

Exercise B.3. Prove useful property 13

$$\begin{aligned}(\lambda v)\cdot w &= \big((\lambda+0)v\big)\cdot w \quad \textit{since } \lambda = \lambda + 0 \in F\\ &= (\lambda v + 0v)\cdot w \quad \textit{by property 6}\\ &= \lambda(v\cdot w) + 0(v\cdot w) \quad \textit{by property 9}\\ &= \lambda(v\cdot w) \quad \textit{since } 0\mu = 0 \in F.\end{aligned}$$

∎

B.2. Orthogonal projection and least squares

The Projection Theorem. Let V be an inner product space, let $\{e_1,\dots,e_n\}$ be a finite collection of vectors, and let S be the subspace spanned by these vectors. Given a vector $x\in V$, there is a unique vector $\tilde{x}$ that is closest to x, i.e. that solves[16]

$$\min_{x'\in S}\|x-x'\|^2.$$

Furthermore, $x-\tilde{x}$ is orthogonal to S, i.e.

$$(x-\tilde{x})\cdot y=0 \quad \text{for all } y\in S.$$

The vector $\tilde{x}$ is called the *orthogonal projection* of x onto S, and $x-\tilde{x}$ is called the *residual*.

If the e_i are linearly independent, i.e. if they form a basis for S, then we can find the coordinates of $\tilde{x}$ with respect to the e_i, and the coordinates are unique. If the e_i are linearly dependent, then there are multiple ways to write $\tilde{x}$ as a linear combination of the e_i.

Example B.4 (Closest point via calculus).
Let $e_1=[1,1,0]$, let $e_2=[1,0,-1]$, and let $x=[1,2,3]$. Find the closest point to x in the span of $\{e_1,e_2\}$. Show that the residual is orthogonal to S.

Just write out the optimization problem we want to solve:

$$\min_{\lambda_1,\lambda_2}\left\|x-(\lambda_1e_1+\lambda_2e_2)\right\|^2.$$

We can compute the solution numerically:

```
e₁,e₂,x = np.array([1,1,0]), np.array([1,0,−1]), np.array([1,2,3])
λ₁,λ₂ = scipy.optimize.fmin(lambda λ: np.linalg.norm(x−λ[0]*e₁−λ[1]*e₂), [0,0])
λ₁*e₁ + λ₂*e₂ # outputs: array([ 0.33332018, 2.66666169, 2.33334151])
```

Or we can try algebra. Expanding the definition of $\|\cdot\|$, we want to minimize

$$x\cdot x-2\big(\lambda_1\,x\cdot e_1+\lambda_2\,x\cdot e_2\big)+\big(\lambda_1^2\,e_1\cdot e_1+2\lambda_1\lambda_2\,e_1\cdot e_2+\lambda_2^2\,e_2\cdot e_2\big).$$

Differentiating with respect to λ_1 and λ_2 and setting the derivatives equal to 0,

$$\begin{aligned}\frac{\partial}{\partial\lambda_1}=0: &\qquad -2\,x\cdot e_1+2\lambda_1\,e_1\cdot e_1+2\lambda_2\,e_1\cdot e_2=0\\ \frac{\partial}{\partial\lambda_2}=0: &\qquad -2\,x\cdot e_2+2\lambda_1\,e_1\cdot e_2+2\lambda_2\,e_2\cdot e_2=0\end{aligned}\tag{2}$$

or equivalently

$$\begin{aligned}\lambda_1\,e_1\cdot e_1+\lambda_2\,e_1\cdot e_2&=x\cdot e_1\\ \lambda_1\,e_1\cdot e_2+\lambda_2\,e_2\cdot e_2&=x\cdot e_2.\end{aligned}$$

We can compute the solution to these equations:

```
e₁ = numpy.array([1,1,0])
e₂ = numpy.array([1,0,−1])
x = numpy.array([1,2,3])
λ₁,λ₂ = numpy.linalg.solve([[e₁@e₁, e₁@e₂], [e₁@e₂, e₂@e₂]], [x@e₁, x@e₂])
λ₁*e₁ + λ₂*e₂ # array([ 0.33333333, 2.66666667, 2.33333333])
```

[16]Mathematicians prefer to write inf rather than min in equations like this, where the minimum is being taken over an infinite set and it hasn't yet been established that the minimum is attained.

For geometrical insight, rearrange equations (2) *to get*

$$\big(x - (\lambda_1 e_1 + \lambda_2 e_2)\big) \cdot e_1 = 0$$
$$\big(x - (\lambda_1 e_1 + \lambda_2 e_2)\big) \cdot e_2 = 0$$

In other words, the residual is orthogonal to e_1 and to e_2, and hence it's orthogonal to every linear combination of e_1 and e_2.

∎

Example B.5 (Closest point via explicit projection).
Let $x = [1, 2, 3]$, and let $\tilde{x}$ be the projection onto the subspace spanned by $e_1 = [1, 1, 0]$ and $e_2 = [1, 0, -1]$. Create an orthonormal basis out of $\{e_1, e_2\}$, and thence find the coordinates of $\tilde{x}$ with respect to the basis $\{e_1, e_2\}$.

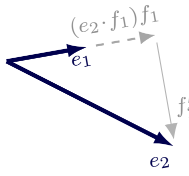
Hint: first use Useful Property 16 on page 66 to get the coordinates of $\tilde{x}$ in the orthonormal basis.

First create the orthonormal basis. Start by setting f_1 to be a unit vector in the same direction as e_1:

$$f_1 = \frac{e_1}{\|e_1\|}.$$

Next, construct f_2 by subtracting the part that's parallel to f_1:

$$f_2' = e_2 - (e_2 \cdot f_1) f_1, \quad f_2 = \frac{f_2'}{\|f_2'\|}.$$



This construction ensures that $f_2' \cdot f_1 = 0$ therefore $f_2 \cdot f_1 = 0$, and it also ensures that both f_1 and f_2 are unit vectors. We've written f_1 and f_2 as linear combinations of e_1 and e_2, and it's easy to check that e_1 and e_2 can be written as linear combinations of f_1 and f_2, thus $\operatorname{span}\{e_1, e_2\} = \operatorname{span}\{f_1, f_2\} = S$. *Thus, $\{f_1, f_2\}$ is an orthonormal basis for S.*

Useful Property 16 now tells us exactly what the coordinates are for $\tilde{x}$:

$$\tilde{x} = (\tilde{x} \cdot f_1) f_1 + (\tilde{x} \cdot f_2) f_2.$$

Furthermore, the Projection Theorem tells us that the residual is orthogonal to $S = \operatorname{span}\{f_1, f_2\}$, which means $(x - \tilde{x}) \cdot f_1 = (x - \tilde{x}) \cdot f_2 = 0$, thus

$$\tilde{x} = (x \cdot f_1) f_1 + (x \cdot f_2) f_2.$$

which with some algebra can be rewritten in terms of e_1 and e_2. In numpy,

```
f1 = e1 / numpy.linalg.norm(e1)
f2' = e2 − (e2@f1) * f1
f2 = f2' / numpy.linalg.norm(f2')

#x̃ in original coordinate system
(x@f1)*f1 + (x@f2)*f2 #array([ 0.33333333, 2.66666667, 2.33333333])

#x̃ in terms of e1 and e2
g1 = numpy.array([1,0]) / numpy.linalg.norm(e1)
g2 = numpy.array([−(e2@f1)/numpy.linalg.norm(e1), 1]) / numpy.linalg.norm(f2')
(λ1,λ2) = (x@f1)*g1 + (x@f2)*g2
λ1*e1 + λ2*e2 #array([ 0.33333333, 2.66666667, 2.33333333])
```

∎

* ✻ *

Colinearity and matrix rank. In Euclidean space, if we have a collection of vectors and we stack them to form a matrix, then the *rank* of the matrix is the dimension of the space spanned by those vectors. In Python, use `numpy.linalg.matrix_rank(numpy.column_stack([e1,e2]))`.

In this example, we projected onto basis vectors e_1 and e_2 that were linearly independent. What happens if we project onto a collection of linearly dependent vectors, e.g. if $e_2 = \alpha e_1$? The Projection Theorem doesn't assume linear independence, so the overall result still holds: there is still a unique projection $\tilde{x}$. The explicit projection method would still work, but it would give $f_2' = 0$, so we'd just discard that vector from the orthonormal basis. Equations (2) would still be correct, but they would have multiple solutions for λ_1 and λ_2.

B.3. Advanced application: Fourier analysis *

In this course on data science, the only vector space we're interested in is a simple finite-dimensional Euclidean space over the real numbers. Before returning to data science, and to illustrate that there's some merit in defining vector spaces abstractly, here's an advanced application—a step on the way to Fourier analysis.

Inner product space. Let V consist of all continuous complex-valued functions on the interval $[-\pi, \pi]$. Define addition of functions in the obvious way, define multiplication by a complex number in the obvious way, and define the inner product to be

$$f \cdot g = \frac{1}{\pi} \int_{-\pi}^{\pi} f(\tau)\overline{g(\tau)}\, d\tau.$$

It is easy to check that properties 1–7 are satisfied, i.e. that this is a vector space over the field of complex numbers. Using some standard results about integration one can also show that properties 8–10 are also satisfied, therefore this is an inner product space. (A typical result: if f is a continuous function, then it is integrable over a finite interval.)

Orthonormal system. Every vector in V is a continuous function. Consider the vectors

$$\{e_1, e_2, \dots\} = \left\{ \frac{1}{\sqrt{2}},\ \cos(\tau),\ \sin(\tau),\ \cos(2\tau),\ \sin(2\tau),\ \cos(3\tau),\ \dots \right\}.$$

(The first element $1/\sqrt{2}$ is a way of writing the constant function $f(\tau) = 1/\sqrt{2}$.) With some A-level trigonometry and calculus, it can be shown that $e_i \cdot e_j = 0$ if $i \neq j$, and $e_i \cdot e_i = 1$ for every i, i.e. that this set is an orthonormal system.

Fourier series. This orthonormal system spans the subspace of V consisting of 'well-behaved' functions, and such functions can be written in coordinate form as

$$f = \sum_{i=1}^{\infty} (f \cdot e_i)\, e_i \tag{3}$$

or equivalently

$$f(\tau) = \frac{a_0}{2} + \sum_{i=1}^{\infty} \Big(a_i \cos(i\tau) + b_i \sin(i\tau) \Big)$$

where

$$\begin{aligned} a_0 &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(\tau)\, d\tau, \\ a_i &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(\tau) \cos(i\tau)\, d\tau \quad \text{for } i \geq 1 \\ b_i &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(\tau) \sin(i\tau)\, d\tau \quad \text{for } i \geq 1. \end{aligned}$$

This is known as the *Fourier series* for f. There are however some technical caveats associated with infinite series—Useful Property 16 only applies to finite bases, but equation (3) is an infinite series corresponding to an infinite orthornormal system, and this is why we need the restriction 'well-behaved functions'. In Part II *Computer Vision* and *Digital Signal Processing* you will learn more about Fourier analysis and other related ways to decompose functions.