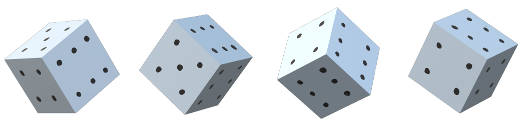
IB Data Science, parts I and II

Damon Wischik, Computer Science, Cambridge University



Prerequisites. This course builds on IA Probability. You should be able to calculate probabilities of events involving ‘and’ and ‘or’. You should be able to calculate with discrete and continuous random variables, and recognize probability density functions and probability mass functions. You should be able to apply Bayes’s rule. You should understand what is meant by independence, and by a joint distribution. These notes give code snippets in Python using numpy, as taught in IA Scientific Computing.

Examinable material. The sections marked * are not part of this course and they are not examinable. A few of these sections have been included in the printout, for your general interest. The others may be found online.

Contents

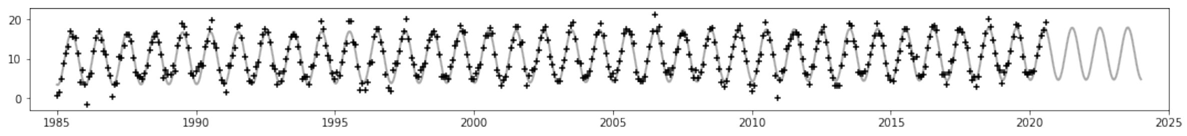
ACKNOWLEDGEMENTS

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

October 2, 2020 at 13:16:00

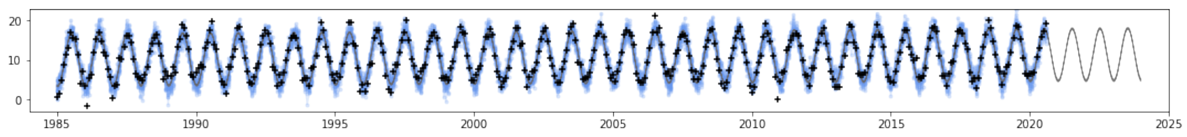
Part I
Learning with probability models

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.



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, plus there's a gradual long-term increasing trend, and on top of this there is variability, but a variation of more than a few $^\circ C$ 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. The next plot shows this cloud.[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. You have most likely come across the basic random variables like Geometric and Gaussian in an introductory course on probability, and section 1 builds them up into richer models, including models for clustering and classifying data.

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.*

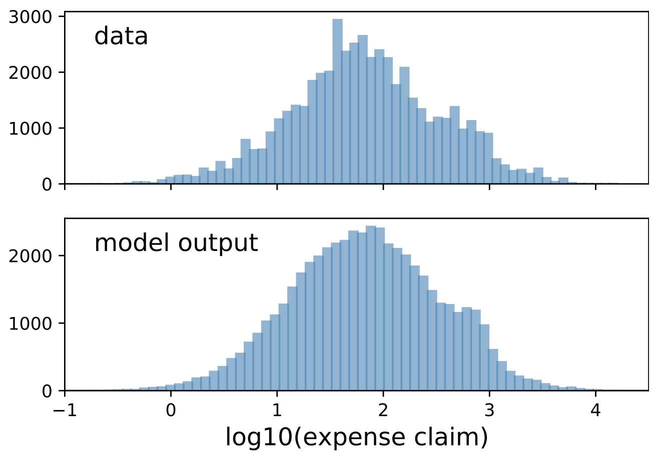
Software 2.0 still needs a human to decide out what to optimize—which comes from probability models. Sections 1.1 and 1.2 describe how to take a given probability model and fit it to data using numerical optimization, first with maths and then with Python.

[1]The plot also shows many grey lines superimposed, each corresponding to a different counterfactual possibility for the entire dataset. 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

Here is data[3] of MP expense claims for office expenses from 2018, The top plot shows a histogram of the data (using a log scale, because otherwise the many small claims are swamped out by a few much larger claims). It looks like a Gaussian distribution, except for a hump around 2.9 which translates to £630. This invites all sorts of questions: why is there a hump? can we figure out which claims are part of the hump and which are just the normal spread? can we even be confident that there is a hump, and it's not just random noise? *The starting point for all of these questions is to invent a probability model for the data.*



The bottom plot shows a histogram of the output from a simple probability model, implemented in the code below. The goal of this part of the course is to understand where the magic constants in the code come from. And to become fluent in building probability models, so that "perhaps there is a hump?" turns immediately into reasonable code.

```
# Load the data, and exclude claims of <= £0
url = 'https://www.cl.cam.ac.uk/teaching/2021/DataSci/data/mpexpenses_2018_office.csv'
mpexpenses = pandas.read_csv(url)['Amount Claimed'].values
mpexpenses = mpexpenses[mpexpenses>0]

plt.hist(np.log10(mpexpenses), bins=75)

# Generate a synthetic sample of the same size as the data
def rx(p=0.966, μ1=1.797, μ2=2.861, σ1=0.679, σ2=0.116):
    # let k='a' with probability p, 'b' with probability 1-p
    k = np.random.choice(['a','b'], p=[p,1-p])
    if k == 'a':
        y = np.random.normal(loc=μ1, scale=σ1)
    else:
        y = np.random.normal(loc=μ2, scale=σ2)
    return np.power(10, y)
xsample = [rx() for _ in range(len(mpexpenses))]

plt.hist(np.log10(xsample), bins=75)
```

[3] Data obtained from `https://www.theipsa.org.uk/mp-costs/your-mp/`. Expense claims for ≤£0 have been excluded.

1.1. Maximum likelihood estimation

We'll begin with some basic probability models starting with the most elementary of them all, coin tossing. Generally our probability model will have unknown parameters which we'd like to estimate using the observed data—this is the 'learning' in machine learning. The first step is to write down the probability of the observed data, and in these basic models the formulae will likely be familiar from an introductory course in probability.

Definitions. The *likelihood* is the probability of the observed data. The *maximum likelihood estimator*, or *mle*, is the parameter value that maximizes the likelihood.

Here are some worked example of maximum likelihood parameter estimation, starting with probability models. For more advanced models, remember two rules:

- The 'observed data' should include absolutely all data that has bearing on the unknown parameter or parameters. It's a venial sin to throw data away when you could use it to learn.
- The probability model should include every single parameter that might have bearing on the observed data. Even if you're only interested in one of them, you should still estimate them all.

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

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

where p is the probability of heads and $1 - p$ is the probability of tails. In code,

```
x = np.random.binomial(p, n=10)
```

Estimate p.

The likelihood is the probability of the observed data[4] $x = 6$*:*

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

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

We've written it as a function of p since that's what we want to estimate, though of course it depends on x also. A sensible estimate for p is the mle, i.e. the value of p that maximizes the likelihood. To find it, solve

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

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,*

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

[4]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.

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 $p/(1-p)$.

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 $p = h/(1+h)$:

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

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. (This is the Multinomial distribution, an extension of the Binomial that allows for more than two categories.)

The log likelihood is

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

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

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

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

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

∎

Exercise 1.4 (Using indicator functions to handle boundaries).
We throw a k-sided dice, and get the answer 10. (It's a fancy dice with lots of sides, tricky to count by eye.) Estimate k, using the probability model

$$\mathbb{P}(\text{throw } x) = \frac{1}{k}, \qquad x \in \{1, \ldots, k\}.$$

First, here's the wrong approach. Write out the likelihood

$$\operatorname{lik}(k) = \frac{1}{k}$$

and try to maximize it. The solution is $k = \infty$. This is intuitively wrong: surely a low value for x should tell us that k is low?

The better approach is to use indicator functions. Indicator functions are an algebraic trick to make working with if/then conditions a bit simpler, such as the "if $x \geq 1$ and $x \leq k$" condition hidden in the probability equation. The indicator function is simply

$$1_A = \begin{cases} 1 & \textit{if statement } A \textit{ is true} \\ 0 & \textit{if statement } A \textit{ is false}. \end{cases}$$

So the likelihood function is really

$$\operatorname{lik}(k) = \frac{1}{k} 1_{x \geq 1 \textit{ and } x \leq k} = \frac{1}{k} 1_{x \geq 1} 1_{x \leq k}.$$

Note how the indicator notation turns 'and' into 'times': the product $1_{x\geq 1} 1_{x \leq k}$ is only equal to 1 when both factors are 1 i.e. when both conditions are true.

Using indicator function notation, the likelihood is

$$\operatorname{lik}(k) = \frac{1}{k} 1_{x \geq 1} 1_{x \leq k} = \frac{1}{k} 1_{x \geq 1} 1_{k \geq x}$$

Note the second trick with indicator functions. I flipped the condition $x \leq k$ into $k \geq x$, to emphasize that I'm interested in this as a function of k. The $1_{x \geq 1}$ term can be ignored, because it doesn't involve k. Now it's easy to sketch the likelihood, and the solution is clearly $k = x$.

■

THINGS TO WATCH OUT FOR

It's common to write $\mathbb{P}(\text{num. heads} = x \,|\, p)$ 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. I prefer $\mathbb{P}(\text{num. heads} = x\,;\, p)$.

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

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

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

* ✻ *

The maximum likelihood procedure is intuitively sensible, but why should we use the mle rather than some other estimator, and how accurate is it? We'll revisit this in part III. In the meantime, here's a question to get you thinking:

If we toss 3 coins and get 3 heads, the maximum likelihood estimator is $\hat{p} = 1$. If we toss 1 million coins and get 1 million heads, it's still 1. We should surely be more confident in the latter case. How should we measure confidence? And what would it take to persuade us we'll *never* see a tail?

1.2. Numerical optimization with scipy

Learning from data requires optimization. The workhorse of machine learning is numerical optimization, since mathematical solutions aren't known except for the most basic of probability models. 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.

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. Here's a simple general-purpose tool that will be adequate for most of this course.

Optimization with scipy. 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 $z = 1 - x - y$.

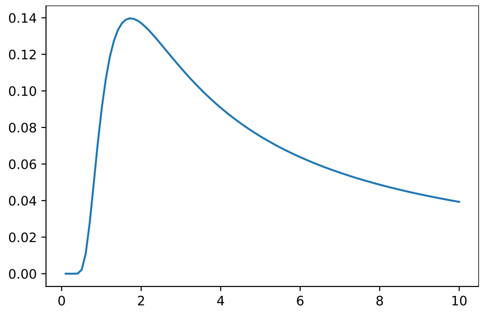
If these sorts of tricks don't work, then you need a specialist optimizer.

Exercise 1.5. 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 as np
import matplotlib.pyplot as plt

def f(σ):
    return np.exp(-3*0.5/np.power(σ,2)) / np.sqrt(2*np.pi*np.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 = np.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(np.exp(τ)), np.log(2))
σ̂ = np.exp(τ̂)
```

∎

Exercise 1.6 (Softmax transformation).
Find the maximum of

$$f(p_1, p_2, p_3) = 0.2\log p_1 + 0.5\log p_2 + 0.3\log p_3$$

over $p_1, p_2, p_3 \in [0,1]$ such that $p_1 + p_2 + p_3 = 1$.

Let's optimize over $(s_1, s_2, s_3) \in \mathbb{R}^2$*, with the transform*

$$p_1 = \frac{e^{s_1}}{e^{s_1}+e^{s_2}+e^{s_3}}, \quad p_2 = \frac{e^{s_2}}{e^{s_1}+e^{s_2}+e^{s_3}}, \quad p_3 = \frac{e^{s_3}}{e^{s_1}+e^{s_2}+e^{s_3}}.$$

The exponentiation makes sure we get positive values, even for negative s_i*. The normalization by* $e^{s_1}+e^{s_2}+e^{s_3}$ *ensures we get* $p_1+p_2+p_3 = 1$*. Since the* p_i *are positive and sum to one, they must be in the range* $[0,1]$*. And every valid* (p_1, p_2, p_3) *corresponds to some* (s_1, s_2, s_3)*, so we're not missing anything by this transform.*

```
def f(θ): # input: a vector of length 3
    s₁,s₂,s₃ = θ
    p = np.exp([s₁,s₂,s₃])
    p₁,p₂,p₃ = p / np.sum(p)
    return 0.2*np.log(p₁) + 0.5*np.log(p₂) + 0.3*np.log(p₃)

# Initial guess s₁ = s₂ = s₃ = 0, i.e. p₁ = p₂ = p₃ = 1/3
s₁,s₂,s₃ = scipy.optimize.fmin(lambda θ: −f(θ), [0,0,0])
p = np.exp([s₁,s₂,s₃])
p / np.sum(p)   # answer: [0.2, 0.5, 0.3]
```

We don't actually need three parameters because we know $p_3 = 1 - p_1 - p_2$ *so there are only two free variables in this problem; another way to see this is to note that for example* $(s_1, s_2, s_3) = (1, 5, 2)$ *gives exactly the same* p_i *as* $(s_1, s_2, s_3) = (1+c, 5+c, 2+c)$ *for any c, so we might as well fix one of the* s_i *at* 0*. But* scipy.optimize *is clever enough to cope with the redundancy.*
∎

There's nothing special about the transform we used; we could use any transform we like, as long as it produces (p_1, p_2, p_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.

training a deep learning image classifier, section 3.1 page 3.1

It's overkill to use a computer to solve this particular optimization problem. It's exactly the problem from exercise 1.3 on page 4, and we could have derived the optimum using maths.

1.3. Notation for random variables

A *random variable* is a function that can give different answers, e.g. a function that calls a random number generator. Two random variables are said to be *independent* if knowing the value of one gives no information about the value of the other. Here are examples to explain how we write random variables, both in code and in mathematics.

```
def ry():
    x = random.random()
    y = x ** 2
    return y
```

$X \sim U[0,1],\ Y = X^2.$

The random.random() function generates a floating point value uniformly between zero and one, and is written as $U[0,1]$. We say X is *generated* or *sampled* from $U[0,1]$. The numpy equivalent is np.random.uniform().

```
def ri(a,b):
    x = random.random()
    i = math.floor(a*x + b)
    return i
```

$X \sim U[0,1],\ I = \lfloor aX + b \rfloor.$

By convention, use capital letters to denote random variables and lower case for constants. This is how we can tell from the maths notation that X and I are meant to be random variables while a and b are meant to be constants. They are called *parameters* for the random variable I.

```
x = random.random()
y = x + 1
```

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

Any piece of code has to live somewhere. The randomness might be wrapped up inside a function, or it might be at the top level of a script as in this example. From the point of view of the person running the script, every time this code is run it will produce different values, and so they'd consider X and Y to be random variables. From the point of view of line-by-line debugging, when we're stepping through the second line we'll treat x and y as values, not as a random numbers.

```
def rz():
    x_1 = random.random()
    x_2 = random.random()
    return x_1 * math.log(x_2)
```

$X_1, X_2 \sim U[0,1],\ Z = X_1 \log X_2.$

X_1 and X_2 are generated *independently*, both of them from $U[0,1]$. Knowing the value of X_1 gives no information about the value of X_2, at least for an idealized random number generator. In maths notation, it's usually implied that random variables are meant to be generated independently, unless stated otherwise. But well-written maths will explicitly add "X_i independent".

```
(x_1,x_2) = [
    random.random()
    for _ in range(2)]
```

$X_i \sim U[0,1],\ i \in \{1,2\}$

Again, this code produces independent random variables.

```
def randpair():
    x_1 = random.random()
    x_2 = random.random()
    y,z = (x_1+x_2, x_1*x_2)
    return (y,z)
y,z = randpair()
```

$(Y,Z) \sim \text{Randpair}$

This random variable generates a pair of values. If I'm told that the value produced for Y is very small, I can deduce that the value for Z is also very small—hence they are not independent. The maths notation indicates "Y and Z are generated together and they are (potentially) not independent".

```
hi = 3
x_1 = random.uniform(0,hi)
x_2 = random.uniform(0,hi)
```

$X_1, X_2 \sim U[0, \text{hi}]$

X_1 and X_2 are sampled independently from the uniform distribution over the range $[0, \text{hi}]$. Technically, it's better to say "X_1 and X_2 are independent, given the parameter hi"—if we didn't know hi, then the value of X_1 would give information about hi, which would give information about X_2. Usually, when you see a probability model written out in maths notation, "independent given the parameters" is implied but not stated explicitly.

```
x = random.random()
y = 1 - x
```

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

The special symbol $X \sim U[0,1]$ denotes that the two random variables have identical distributions. In this example, it would be correct to write $Y \sim U[0,1]$ and $X \sim Y$. These are *not* imperative statements about how we generate Y, they're mathematical statement that the outputs of X and Y and $U[0,1]$ are statistically indistinguishable. Read $X \sim Y$ as "if you plot a histogram of X, and separately plot a histogram of Y, they'll be the same".

The expression $Y = 1 - X$ likewise is not an imperative statement, it's a statement that every time we run the script this equation is true. It would be just as correct to write $X = 1 - Y$.

```
x = random.random()
y = np.random.normal(loc=x,
scale=0.1)
```

$X \sim U[0,1]$, $Y \sim N(X, 0.1)$

In maths notation, the random variable X appears on the right hand side of $Y \sim \cdot$, which means "given a value for X, Y has the following distribution". We don't write $Y = N(X, 0.1)$ because $N(X, 0.1)$ specifies a distribution, not a value, and $=$ is the symbol for relating values whereas $\sim$ is the symbol for relating distributions.

EXOTIC RANDOM VARIABLES

The output type of many basic random variables is a number, either a real number as in ry() or an integer as in ri(a,b). We say a random variable is $\mathcal{S}$-valued, or *takes values in* $\mathcal{S}$, or *has support* $\mathcal{S}$, if $\mathcal{S}$ is the set to which all its output values belong.

It's worth stressing that random variables don't have to be numerical, they can return tuples as with randpair(), or lists. The probability models in sections 1.6 and 1.7 describe random lists. Random variables can return functions, as in the code below. Or even infinite sequences; random infinite sequences are known as random processes, and they are the topic of part IV

```
def rf(a):
    # a sine wave with given amplitude a and random phase
    φ = random.random()
    def f(x):
        return a * math.sin(2*π*(x + φ))
    return f

f = rf(1)      # generate a random function F
print(f(2))    # F is a function, returning the same F(x) every time we call it
print(f(2))
g = rf(1)      # generate a second random function G
print(g(2))    # G is a different function to F
```

```
0.71319
0.71319
-0.91915
```

1.4. Standard random variables

As a data scientist you should be familiar with a repertoire of standard random variables, what they are used for, and the parameters they take. Here are some of the standard random variables that you will use over and over again. To generate a list of n independent random variables, call the numpy routine with an extra argument size=n.

More random variables: appendix, page 75

DISCRETE RANDOM VARIABLES (integer-valued)

Uniform	$X \sim U\{a,\dots,b\}$	An integer, uniformly distributed in $\{a,\dots,b\}$. random.randint(a,b), numpy.random.randint(low=a,high=b+1)
Geometric	$X \sim \text{Geom}(p)$	Counts the number of failures before a success, where success happens with probability p. numpy.random.geometric(p)-1
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. numpy.random.binomial(n,p)
Poisson	$X \sim \text{Pois}(\lambda)$	Used for modelling counts such as the number of buses that arrive in an interval of time. If the average number of buses per hour is μ then the number in t hours is $\text{Pois}(\mu t)$. numpy.random.poisson(lam=λ)

CONTINUOUS RANDOM VARIABLES (real-valued)

Uniform	$X \sim U[a,b]$	A floating point value, uniformly distributed in the interval $[a,b]$. random.random() for $U[0,1]$, random.uniform(a,b), numpy.random.random(low=a,high=b)
Exponential	$X \sim \text{Exp}(\lambda)$	Used for modelling waiting times such as the time until the next bus arrives. If the average number of buses per hour is μ then the time until the next bus is $\text{Exp}(\mu)$. numpy.random.exponential(scale=$1/\mu$)
Normal / Gaussian	$X \sim N(\mu,\sigma^2)$	Used to model magnitudes, e.g. the height of a person. numpy.random.normal(loc=μ,scale=σ)
Beta	$X \sim \text{Beta}(a,b)$	Common in Bayesian inference. numpy.random.beta(a,b)

Bayesian use of the Beta distribution, page 54

RANDOM SAMPLING to generate samples from a list $a = [a_1,\dots,a_n]$

Categorical	Given a probability vector $p = [p_1,\dots,p_n]$, pick a_i with probability p_i. numpy.random.choice(a, p=p)
Sample with replacement	Pick k items with replacement. random.choices(a, k=k) numpy.random.choice(a, size=k, replace=True)
Sample without replacement	Pick k items without replacement. random.sample(a, k) numpy.random.choice(a, size=k, replace=False)

LOOKING UP RANDOM VARIABLES

Wikipedia is fine for looking up random variables. Here are two examples, the entries for the Poisson distribution and the Normal distribution The Poisson random variable is discrete (its support is the set of integers) so Wikipedia tells us its probability mass function; the Normal distribution is continuous (its support is the set of all real numbers) so Wikipedia tells us its probability density function.

‘support’ is defined on page 9

Also see scipy.stats, which has library functions for evaluating pmf, pdf, and cdf, among others. Be careful about parameters: Wikipedia and scipy sometimes use different parameterizations.

Poisson distribution

Notation	Pois(λ)
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	$N(\mu, \sigma^2)$ or Normal(μ, σ^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	$\Phi\Big(\dfrac{x-\mu}{\sigma}\Big)$
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 two properties here.

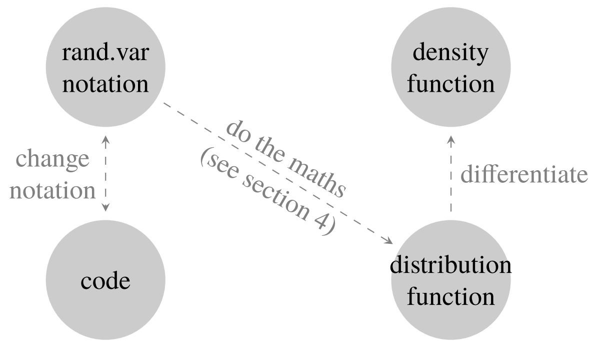
see IA Probability lecture 6

- If $X \sim \text{Normal}(\mu, \sigma^2)$ and a and b are constants, then $aX + b \sim \text{Normal}(a\mu + b, a^2\sigma^2)$. So, for example, $\mu + \sigma\,\text{Normal}(0, 1) \sim \text{Normal}(\mu, \sigma^2)$. Most random variables don't behave so nicely—for example, if $Y \sim \text{Pois}(\lambda)$ then $aY + b$ is not a Poisson (unless $a = 0$ and $b = 1$!)
- 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 so often nearly true. In IA Probability you learned about the Central Limit Theorem, says that we can approximate sums of random variables by a Normal, but the value of the approximation goes further.[5]

[5] See *Probability Theory: the Logic of Science* by E.T.Jaynes (CUP 2003) chapter 7 for why this might be.

1.5. Likelihood notation

There are four ways to describe a random variable: random variable notation, code, distribution function, and density function or probability mass function (depending on whether the random variable is continuous or discrete). None of these is more definitive than the others; they're just different views of the same thing. Here are the links between these views, and an example.



Random variable notation:
$X \sim -\log(U[0,1])/\lambda$

Code:
x = - np.log(np.random.uniform())/λ

Density function:
$f(x) = \lambda e^{-\lambda x}$

Cumulative distribution function:
$\mathbb{P}(X \leq x) = e^{-\lambda x}$

For many machine learning tools, it's most convenient to use the density function / probability mass function view, depending on whether the random variables are continuous or discrete. To save ourselves the bother of always having to write out two versions of every sentence, once for continuous and once for discrete, it's useful to invent a common symbol.

Likelihood. For a random variable X, the *likelihood function*[6], written Pr, is defined to be

$$\Pr_X(x) = \begin{cases} \mathbb{P}(X = x) & \text{if } X \text{ is a discrete r.v.} \\ f(x) & \text{if } X \text{ is a continuous r.v. with density } f. \end{cases}$$

The job of the likelihood notation is to keep track of two separate things: the random variable X, which is a function that can give different answers; and a value x which is one possible output value of the function. This separation is particularly helpful for more complex probability models. (As any mathematician will tell you, the way to deal with complexity is to invent good notation.)

Transformed random variables, e.g. $\Pr_{X+Y}(z)$ or $\Pr_{X^2}(z)$	We'll build up probability models by taking the basic building blocks (discrete and continuous random variables) and doing things to them. The Pr notation helps us keep track. Instead of writing out "Given two discrete random variables X and Y, let $Z = X + Y$, and then $\mathbb{P}(Z = z) = \ldots$" we just write "$\Pr_{X+Y}(z) = \ldots$"
Pairs of discrete r.v.: $\Pr_{X,Y}(x,y)$ $= \mathbb{P}(X = x \text{ and } Y = y)$	If we have generated a pair of discrete random variables, e.g. `x = random.randint(1,10)` `y = x + random.randint(0,1)` then we want to able to reason about both of them together, for example to figure out whether they are independent. In such cases, $\Pr_{X,Y}(x,y)$ is used to describe their *joint distribution*. Formally, the pair $Z = (X, Y)$ is itself a discrete random variable, and $\Pr_{X,Y}(x,y)$ is nothing other than the probability mass function of Z, namely $\mathbb{P}\big(Z = (x,y)\big)$.

[6]It's common especially in statistics to reserve the word 'likelihood' for parameters and hypotheses, for example "the likelihood that $\theta > 0$ given the data". In machine learning, it's more common to read "the likelihood of the data". In an earlier version of these notes I gave these two concepts different names, but students couldn't see the point—and I agree with them.

Pairs of continuous r.v.: $\Pr_{X,Y}(x,y) = \cdots$	Likewise if we have generated a pair of continuous random variables, we want to be able to reason about both of them together. It is possible to define a *joint probability density function* $\Pr_{X,Y}(x,y)$, as the derivative of an appropriate cumulative distribution function for two variables, and the details are left to section 4. What really matters is the intuition: think of it as an 'and' probability, similar to the discrete case.
Parameterized random variables: $\Pr_X(x\,;\,\theta)$	For a random variable which takes parameters, it's sometimes useful to emphasize those parameters, especially when they are parameters we want to learn from data. We can write $\Pr_X(x\,;\,\theta)$ to emphasize that the distribution of X depends on θ. Example: $X \sim U[0,\theta] \quad \Leftrightarrow \quad \Pr_X(x\,;\,\theta) = \frac{1}{\theta} \text{ for } x \in [0,\theta]$ It is a matter of style and taste whether or not you include the parameters in Pr.
Guess[7] $\Pr(x,y,z)$ $\Pr_{X,Y,Z}(u)$	When building up probability models, it's often fairly obvious from the context which random variables we are referring to. For example, if we see $\Pr(x,y,z)$, we know it's referring to a tuple of three random variables, probably called (X,Y,Z). Or if we see $\Pr_{X,Y,Z}(u)$, we know that we're describing a tuple random variable (X,Y,Z), and therefore the output u must be a tuple of length 3. Think of this as type inference for maths notation!
Random sample: $\Pr(x_1,\dots,x_n) = \Pr_X(x_1)\times\cdots\times\Pr_X(x_n)$	Many machine learning problems involve a sample of independent observations from identical distributions, also known as an i.i.d. sample or random sample. Random variables are independent if their joint likelihood factorizes, $\Pr_{X,Y}(x,y) = \Pr_X(x)\Pr_Y(y) \quad \text{for all } x,y,$ so the equation for $\Pr(x_1,\dots,x_n)$ describes a random sample of n values drawn from common distribution X.

MAXIMUM LIKELIHOOD ESTIMATION, AGAIN

The advantage of the Pr notation is that it allows many equations in data science and machine learning to be written in a uniform way, applying to all different types of random variable. Here is the general way to state maximum likelihood estimation.

Maximum likelihood estimation. If we have a probability model where the possible outcomes are described by a random variable X whose distribution depends on an unknown parameter θ, and we have observed the outcome x, then the maximum likelihood estimator for θ is that value that maximizes $\Pr_X(x\,;\,\theta)$,

$$\hat\theta = \operatorname*{arg\,max}_\theta \Pr_X(x\,;\,\theta).$$

This procedure applies whether x is a single observation, a tuple, a random sample, or any other type of random variable.

[7]The $\Pr_X(x)$ is not by any means universal. In engineering, it's common to write for example $p(x)p(y)$ and expect the reader to intuit that there are two different random variables being described, what this course refers to as $\Pr_X(x)\Pr_Y(y)$. Another common engineering notation is $\mathcal{N}(x;\mu,\sigma^2)$ to refer to the density of the Normal distribution, $\Pr_{N(\mu,\sigma^2)}(x)$; this at least is unambiguous, but it doesn't help if we want to refer to custom random variables. In mathematics, the style is to be absolutely precise but a little long-winded: "Let $f(x)$ be the pdf of X, and let $g(x)$ be the pdf of Y, and consider a random variable with density $f(x)g(x)$." The $\Pr_X(x)$ notation is, in the author's opinion, superior to both styles.

DERIVING THE LIKELIHOOD

Since so much of machine learning revolves around maximum likelihood estimation and Pr, the obvious question is: how do we find Pr? For creative data science and machine learning, we'll be inventing our own probability models, and we'll need to derive Pr ourselves. Useful guidelines, at least for continuous random variables, is (1) learn the cumulative distribution function for standard random variables, (2) find the cumulative distribution function for the random variable you're interested in, using common sense and the general rules of probability, (3) differentiate to get the likelihood. Section 4 has many more examples of this type of calculation, and some general tips.

Exercise 1.7. Find the density of the random variable Y generated by this code:

```
def ry():
    x = np.random.uniform()
    return x**2
```

This is a continuous random variable, so we'll work out $\mathbb{P}(Y \leq y)$ and then differentiate. It's impossible to get $Y < 0$ or $Y > 1$, so

$$\begin{aligned} \mathbb{P}(Y \leq y) &= 0 \quad \textit{for } y < 0 \\ \mathbb{P}(Y \leq y) &= 1 \quad \textit{for } y \geq 1. \end{aligned}$$

For the general case $y \in [0,1]$, a good strategy is to rewrite the probability we want in terms of simpler random variables, in this case X:

$$\mathbb{P}(Y \leq y) = \mathbb{P}(X^2 \leq y) = \mathbb{P}\big(X \in [-\sqrt{y}, \sqrt{y}]\big).$$

When we look up the cumulative distribution function of $U[0,1]$,

$$\mathbb{P}(X \leq x) = \begin{cases} 0 & \textit{if } x \leq 0 \\ x & \textit{if } x \in [0,1] \\ 1 & \textit{if } x \geq 1 \end{cases}$$

so

$$\mathbb{P}(Y \leq y) = \frac{\sqrt{y} - 0}{1 - 0} = \sqrt{y} \quad \textit{for } y \in [0,1].$$

Differentiating,

$$\Pr_Y(y) = \frac{d}{dy}\mathbb{P}(Y \leq y) = \begin{cases} 0 & \textit{if } y < 0 \\ {}^1\!/\!_2/\sqrt{y} & \textit{if } y \in [0,1] \\ 0 & \textit{if } y > 0. \end{cases}$$

(Don't worry about the value of $\Pr_Y$ at $y = 0$ or $y = 1$. The cumulative distribution function isn't differentiable at these points, and it doesn't matter what value we assign to the density.)

∎

Exercise 1.8 (Gaussian mixture model). For the code on page 2, show that the likelihood of Y is

$$\Pr_Y(y\,;\,p,\mu_1,\mu_2,\sigma_1,\sigma_2) = \frac{p}{\sqrt{2\pi\sigma_1^2}}e^{-(y-\mu_1)^2/2\sigma_1^2} + \frac{1-p}{\sqrt{2\pi\sigma_2^2}}e^{-(y-\mu_2)^2/2\sigma_2^2}.$$

(The code on page 2 then sets $X = 10^Y$ and proposes X as a model for MP expense claim amounts. But we might as well save some algebra by directly using Y as a model for $\log_{10}$ of MP expense claim amounts.)

1.6. Generative models / unsupervised learning

Generative modelling just means "given a set of observations, fit a distribution that they might have come from". It's good manners to give our answer as code or as random variable notation. This is why it's called 'generative' modelling: we can generate new synthetic observations by running the code. See for example the synthetic faces at `https://thispersondoesnotexist.com`, generated from a probability model trained on a dataset consisting of images of faces Let's spell out the procedure.

1. Suppose we're given a dataset of observations $x_1, \dots, x_n$
2. Choose a distribution with one or more tunable parameters, and model the dataset as independent observations from this distribution, i.e.
$$\Pr(x_1, \dots, x_n\,;\,\theta) = \Pr_X(x_1\,;\,\theta) \times \cdots \times \Pr_X(x_n\,;\,\theta)$$
where X is the random variable we've chosen and θ is its parameters
3. Fit the model using maximum likelihood estimation, i.e. solve
$$\hat\theta = \arg\max_\theta \log \Pr(x_1, \dots, x_n\,;\,\theta)$$

Exercise 1.9 (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.

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

The Normal distribution: page 77 in the appendix

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

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

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

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:*

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

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 1.10 (Gaussian mixture model).
At the beginning of this section, page 2, we considered a dataset of MP expense claims, and we proposed the probability model

```
def rx(p, μ1, μ2, σ1, σ2):
    k = np.random.choice(['a','b'], p=[p,1-p])
    if k == 'a':
        y = np.random.normal(loc=μ1, scale=σ1)
    else:
        y = np.random.normal(loc=μ2, scale=σ2)
    return np.power(10, y)
```

Exercise 1.8 on page 14 suggested we simplify by using Y directly as a model for $\log_{10}$ of MP expense claims, and asked us to derive the likelihood function

$$\Pr_Y(y\,;\,p,\mu_1,\mu_2,\sigma_1,\sigma_2) = \frac{p}{\sqrt{2\pi\sigma_1^2}}e^{-(y-\mu_1)^2/2\sigma_1^2} + \frac{1-p}{\sqrt{2\pi\sigma_2^2}}e^{-(y-\mu_2)^2/2\sigma_2^2}.$$

Fit this model to the data.

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 $p \in [0,1]$, $\sigma_i > 0$, so let's optimize the parameter

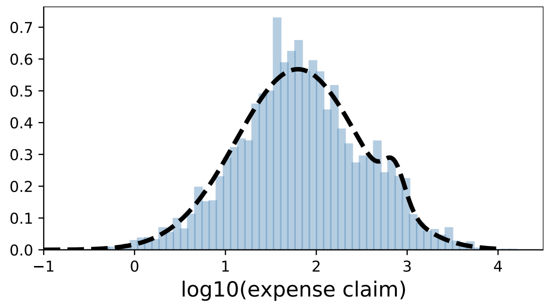
$$\theta = (q,\ \mu_1,\mu_2,\ \tau_1,\tau_2) \in \mathbb{R}^5$$

transformed into the parameters we want by

$$p = \frac{e^q}{1+e^q}, \qquad \sigma_i = e^{\tau_i}.$$

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

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

def logPr(y, θ):
    q,μ1,μ2,τ1,τ2 = θ
    p = np.exp(q) / (1 + np.exp(q))
    σ1,σ2 = numpy.exp([τ1,τ2])
    lik = p*φ(y, loc=μ1, scale=σ1) + (1-p)*φ(y, loc=μ1, scale=σ2)
    return np.log(lik)

y = np.log10(mpexpenses)
# initial guess inspired by the plot on page 2
initial_guess = [1, 1.8,2.7, math.log(1.2),math.log(0.5)]
θ̂ = scipy.optimize.fmin(lambda θ: -np.sum(logPr(y,θ)), initial_guess, maxiter=5000)

# Plot the fitted density, and a histogram of the actual log10(data)
# (Using density=True rescales the histogram to have total area
# equal to 1, making it directly comparable to the fitted density plot.)
plt.hist(np.log10(mpexpenses), bins=75, density=True)
y = np.linspace(-1,4,200)
f = np.exp(logPr(y,θ̂))
plt.plot(x, f)
```



∎

Exercise 1.11 (Categorical random variable).
Suppose we ask $n = 100$ people their views on Brexit, and let $x_i \in \{L,R,U\}$ be the choice of person i, L = Leave, R = Remain, U = Undecided. Fit the probability model

$$\mathbb{P}(X = \text{Leave}) = p_L, \quad \mathbb{P}(X = \text{Remain}) = p_R, \quad \mathbb{P}(X = \text{Undecided}) = p_U.$$

(This is called a Categorical random variable. We could also write it as $\Pr_X(x) = p_x$, or as $X \sim \text{Cat}([p_L,p_R,p_U])$. It's implied by the question that we want to estimate $p_L \geq 0$, $p_R \geq 0$,

$p_U \geq 0$ such that $p_L + p_R + p_U = 1$.)

The likelihood of the entire dataset is

$$\Pr(x_1, \dots, x_n \,;\, p_L, p_R, p_U) = \prod_{i=1}^{n} p_{x_i} = p_L^{n_L} p_R^{n_R} p_U^{n_U}$$

where n_L is the total number who voted Leave, and n_R and n_U likewise. So the log likelihood is

$$\log \Pr(x_1, \dots, x_n \,;\, p_L, p_R, p_U) = n_L \log p_L + n_R \log p_R + n_U \log p_U.$$

Finding the maximum likelihood for this model is exactly what we did algebraically in exercise 1.3 and computationally in exercise 1.6. The answer is

$$p_x = \frac{n_x}{n}.$$

∎

* ✻ *

'Unsupervised learning' is a general term for any machine learning method that works with unlabelled data. This is as opposed to supervised learning, in which our dataset consists of pairs (x_i, y_i) and we're interested in predicting y_i given x_i. For example, if we have a dataset of images of faces $x_1, \dots, x_n$ and we want to learn to generate a new synthetic face, this is unsupervised learning; if each face x_i has been annotated with an emotion y_i and we want to learn to classify images of faces according to the emotion they display, this is supervised learning.

Unsupervised learning includes all sorts of descriptive tools, many of them with an emphasis on clustering algorithms, not just generative modelling. The advantage of generative modelling over purely descriptive tools is that it comes with a general-purpose principled way to evaluate how good a model is, via the likelihood function. Modern approaches to unsupervised learning such as VAE (variational auto-encoders) and GANs (generative adversarial networks) are based on generative modelling.

1.7. Supervised learning

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* or *features*.

The goal of probabilistic supervised learning is to invent a probability model which, given the predictor, can predict the label. The procedure is simple: (1) invent a probability model with parameters we want to learn, (2) write out the likelihood, (3) maximize it.

Example 1.12.
For example, here are the first few rows of a dataset of Cambridge temperatures. To model climate change, we'd let the label variable be temp (the average monthly temperature measured in °C), and investigate how it depends on the predictor variable t = yyyy+(mm-1)/12 (the timestamp measured in years).

yyyy	mm	tmax	tmin	temp	af	rain	sun	station
1985	1	3.4	-2.2	0.60	23	37.3	40.7	Cambridge
1985	2	4.9	-1.9	1.50	13	14.6	79.0	Cambridge
1985	3	8.7	1.1	4.90	10	45.8	97.8	Cambridge

When we plot temp against t (page 1), we might suspect there's an annual sinusoid, superimposed on a long-term warming trend, plus noise. This suggests a probability model like

```
# five parameters that need to be estimated
α, φ = ...  # amplitude and phase of sinusoid
c, γ = ...  # baseline temperature, and long-term rate of warming
σ = ...  # variability
temp = α*np.sin(2*π*(t+φ)) + c + γ*t + np.random.normal(loc=0, scale=σ)
```

We will return to this example in section 2.2, and go about estimating the parameters.

Let's spell out the supervised learning procedure. Suppose we're given a dataset $(x_1, y_1), \ldots, (x_n, y_n)$ where y_i is the label in record i and x_i is the predictor variable or variables.

Probabilistic supervised learning.

1. Choose a probability distribution for the label, where the distribution depends on one or more unknown parameters as well as on the predictors. Write its likelihood as
$$\Pr_Y(y\,;\,x,\theta), \qquad \theta \text{ the unknown parameter(s).}$$
2. Model the dataset as independent observations of Y drawn from this distribution, i.e. let the likelihood of the dataset be
$$\Pr(y_1,\ldots,y_n\,;\,x_1,\ldots,x_n,\theta) = \Pr_Y(y_1\,;\,x_1,\theta)\times\cdots\times\Pr_Y(y_n\,;\,x_n,\theta).$$
(It gets tedious to repeat the predictors $x_1, \ldots, x_n$ all over the place, and we're treating them as known values not as observations or parameters, so feel free to omit them.)
3. Estimate θ using maximum likelihood estimation, i.e. by solving
$$\hat\theta = \arg\max_\theta \log \Pr(y_1,\ldots,y_n\,;\,x_1,\ldots,x_n,\theta).$$
This is called *fitting* or *training* 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.

Exercise 1.13 (Straight-line fit).
Given a labelled dataset $[(y_1, x_1), \dots, (y_n, x_n)]$ consisting of pairs of real numbers, fit the model

$$Y_i = a + bx_i + \text{Normal}(0, \sigma^2),$$

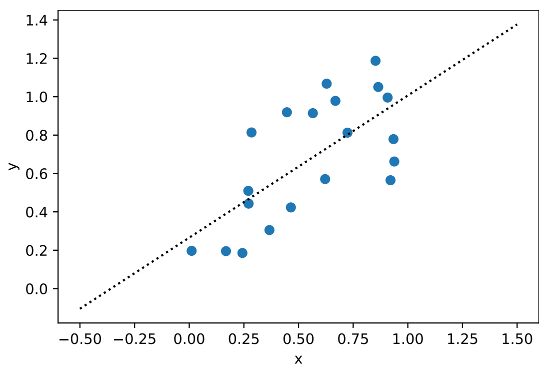
which we could alternatively write as

$$Y_i \sim \text{Normal}\big(a + bx_i, \sigma^2\big)$$

where σ is given, and a and b are parameters to be estimated.

the two ways of writing this model are equivalent: see properties of the Normal distribution, page 11, and IA Probability lecture 6

```
# get the data, and the known parameter σ
x = np.array([...])
y = np.array([...])
σ = ...

# define the likelihood, and maximize it
def loglik(y, x, θ):
    a,b = θ
    lik = scipy.stats.norm.pdf(y, loc=a+b*x, scale=σ)
    return np.log(lik)

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

# Plot the line y = â + b̂x, and superimpose the dataset
xnew = numpy.linspace(-.5,1.5,100)
plt.plot(xnew, â+b̂*xnew, linestyle='dotted', color='black')
plt.scatter(x, y)
```



∎

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

The likelihood function for Y_i is

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

so the log likelihood of the dataset is

$$\begin{aligned}\log \Pr(y_1, \dots, y_n \,;\, x_1, \dots, x_n, a, b) &= \sum_{i=1}^{n} \log \Pr_Y(y_i \,;\, x_i, a, b)\\ &= \sum_{i=1}^{n} \Big\{ -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(y_i - a - bx_i)^2}{2\sigma^2} \Big\}\\ &= -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - a - bx_i)^2.\end{aligned}$$

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:

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

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 1.14 (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 76

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, $0 \leq p \leq 1$ is the success probability, and the probability mass function is

$$\mathbb{P}(X = r) = \binom{n}{r} p^r (1-p)^{n-r}, \quad r \in \{0, \dots, n\}.$$

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

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

(We've omitted the $x_1, \dots, x_n$ on the left hand side, because it's tedious to write it them out every single time.) The log likelihood is

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

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

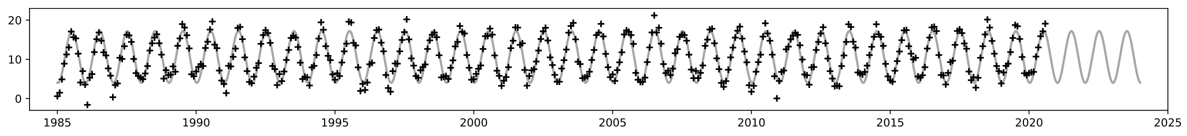
$$\hat{p} = \frac{\sum_i y_i}{\sum_i x_i}.$$

∎

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.

2. Feature spaces / linear regression

Here's a fitted model, for monthly average temperatures in Cambridge. The crosses mark the datapoints, and the grey line shows the model, which is a sinusoid whose amplitude, phase, and offset have been fitted to the data. When we see a plot like this, it's crying out for investigation: would the model fit better if we allowed for a long-term warming trend? should the trend be linear, or is warming accelerating? is the difference between winter and summer constant, or are the extremes getting more extreme?



There are so many questions we could ask, we need to be able to frame them in maths and turn them into code with barely a moment's thought. We need to spend our time thinking about the model, not messing around with compilers or optimizers.

This section introduces a class of models called *linear models*. Once we know a few standard tricks, described in section 2.2, it's easy to craft linear models to answer all sorts of modelling questions. They're also very fast to fit, using specialist routines derived from linear algebra. 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 tabulation!) to get a sense of the data you're working with.

It's possible to view linear models in a purely algorithmic way, called *least squares estimation*, without any probability at all. But there is also a probabilistic interpretation using Normal random variables. In fact, least squares estimation was invented by Gauss, as was the Normal distribution—another name for the Gaussian distribution. It's best to see linear models as just another example of probabilistic modelling, because least squares estimation only works for real-valued labels (like temperature) whereas the probability model can easily be extended to other types of data.

in section 3.3 we use linear modelling tricks for categorical data

There is another more important reason for preferring the probabilistic interpretation: measuring how certain we should be in our answers. The climate data shown above shows a steady increase of 0.35378°C per decade—but how precisely do we know this? surely not to 5 decimal places! All the tools for quantifying certainty, which we'll study in part III, are based on probability models.

2.1. Fitting a linear model

Linear modelling is a type of supervised learning. Suppose we're given a dataset $(x_1, y_1), \dots, (x_n, y_n)$ where y_i is the label for record i and x_i is a tuple of predictor variables, called *features*,

$$x_i = \big[e_{1,i}, \dots, e_{K,i}\big].$$

Linear modelling requires that all the variables in the dataset are numerical: $y_i \in \mathbb{R}$, and each $e_{k,i} \in \mathbb{R}$.

to use non-numerical predictors in a linear model, see the tricks in section 2.2

Linear model. A *linear model* is a model of the form

$$y_i \approx \beta_1 e_{1,i} + \cdots + \beta_K e_{K,i}$$

where $\beta_k \in \mathbb{R}$ is a parameter weighting the kth feature. It's convenient to write this in vector notation as

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

where $y = [y_1, \dots, y_n]$ is called the *response vector* and $e_k = [e_{k,1}, \dots, e_{k,n}]$ is called a *feature vector*. The $\approx$ means that the model is not expected to be a perfect fit; we define the *residual vector* ε to be

$$\varepsilon = y - \big(\beta_1 e_1 + \cdots + \beta_K e_K\big).$$

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, as described below. Once we have estimates for the parameters, call them $\hat\beta$, we can predict the response for a new case:

$$y_{\text{predicted}} = \hat\beta_1 e_{1,\text{new}} + \cdots + \hat\beta_K e_{K,\text{new}}.$$

Example 2.1.
The Iris dataset was collected by the botanist Edgar Anderson and popularized[8] 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://www.cl.cam.ac.uk/teaching/2021/DataSci/data/iris.csv`.

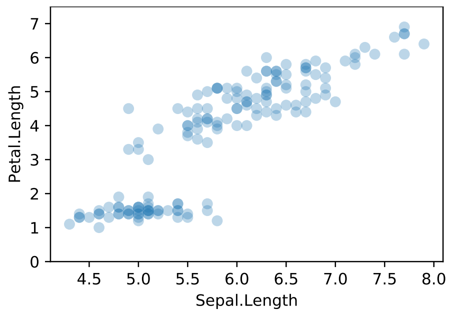
Petal length	Petal width	Sepal length	Sepal width	species
1.0	0.2	4.6	3.6	setosa
5.0	1.9	6.3	2.5	virginica
5.8	1.6	7.2	3.0	virginica
1.7	0.5	5.1	3.3	setosa
4.2	1.2	5.7	3.0	versicolor
...				

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

[8] 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`



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

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

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

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

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

```
iris = pandas.read_csv('https://www.cl.cam.ac.uk/teaching/2021/DataSci/data/iris.csv')

# A linear model with three feature vectors [one, x, x**2] and the response vector y
one, x = numpy.ones(len(iris)), iris['Sepal.Length']
y = 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_
```

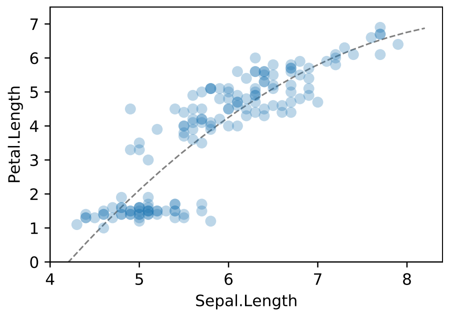
```
(-17.447,  5.392,  -0.296)
```

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 the quadratic linear model above as having two features, $\mathsf{Sepal.Length}$, and $(\mathsf{Sepal.Length})^2$. 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 $\mathsf{Petal.Length}$ 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, $\mathsf{Sepal.Length}$. 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. For the 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.

Notation warning. An equation like

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

can be read in two ways. One one hand it's a vector equation, where $\mathsf{Petal.Length}$ and $\mathsf{Sepal.Length}$ are vectors from the dataset. On the other hand it's a science-style equation, telling us the likely value of $\mathsf{Petal.Length}$ for a hypothetical new iris. When we use it for fitting, we're using it as a model for the dataset. When we use it to make predictions, we're using it as a model for the world.

* ✻ *

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.

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

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

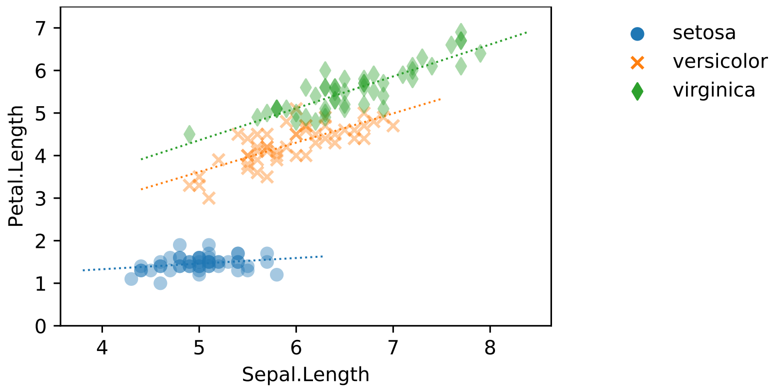
2.2. Feature design

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

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

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

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

Or writing symbols for the vectors:

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

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)
```

Notation warning. The model equation

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

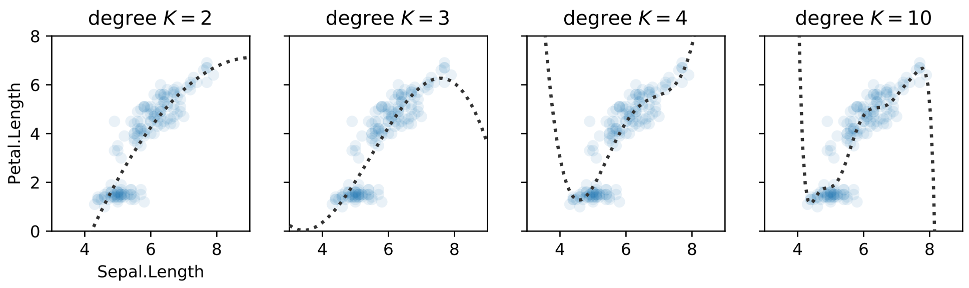
should be interpreted as a vector equation. In Python, the $\alpha_{\mathsf{species}}$ vector comes from

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

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,

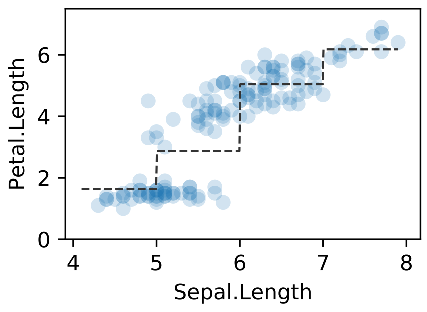
$$\text{Petal.Length} \approx \alpha + \beta_1\,\text{Sepal.Length} + \beta_2\,(\text{Sepal.Length})^2 + \cdots + \beta_K\,(\text{Sepal.Length})^K$$



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.

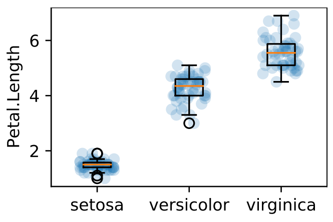
$\lfloor x \rfloor$ is x rounded down to the nearest integer

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



This model is more honest than polynomials 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. COMPARING GROUPS



We can also use one-hot coding to compare groups. Suppose we have a vector $x = [x_1, \ldots, x_m]$ from one group, and $y = [y_1, \ldots, y_n]$ from a second group, and we want to measure the difference between the groups. Obviously we could just compare the means, $\bar{y} - \bar{x}$, and there's nothing wrong with that! But it's good to see how to achieve the same thing via features.

For any sort of linear modelling question, a good starting point is to ask how we'd store the data in a spreadsheet or database table—identify the rows and the columns. Here we have two groups of measurements, so the natural way to store them is with a two-column spreadsheet, one column called group, the other called value. Now suppose we fit the linear model

$$\text{value} \approx \alpha + \beta 1[\text{group} == B].$$

The predicted value for items in group A is α, the predicted value for items in group B is $\alpha + \beta$, so β must be the difference between the two groups. (It will in fact be equal to $\bar{y} - \bar{x}$, but we need to linear algebra to prove that.)

linear algebra, section 2.5

group	value
A	x_1
$\vdots$	$\vdots$
A	x_m
B	y_1
$\vdots$	$\vdots$
B	y_n

2.2.4. PERIODIC PATTERNS

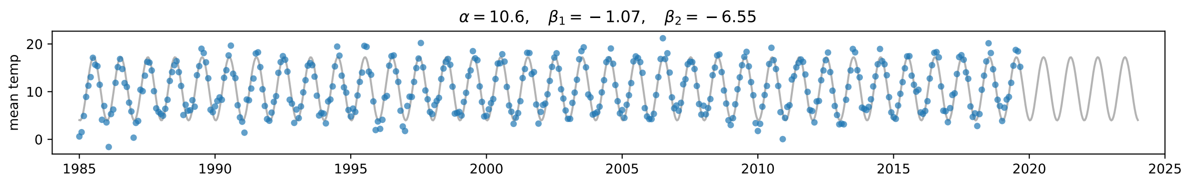
Example 2.2.
The UK Met Office makes available historic data[9] from 37 stations around the UK. Each station has monthly records for mean daily maximum temperature `tmax`, mean daily minimum temperature `tmin`, days of air frost `af`, total rainfall `rain`, and total sunshine duration `sun`. Coverage varies; the longest records are from Oxford and from Armagh, going back to 1853. A snapshot is available at `https://www.cl.cam.ac.uk/teaching/2021/DataSci/data/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://www.cl.cam.ac.uk/teaching/2021/DataSci/data/climate.csv')
df = climate.loc[(climate.station=='Cambridge') & (climate.yyyy>=1985)]
t = df.yyyy + (df.mm-1)/12
temp = (df.tmin + df.tmax)/2
```

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

```

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_)
```

2.2.5. SECULAR TREND

We might reasonably suspect that the periodic model

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

isn't a great fit to the data, because it doesn't allow for 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:

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

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

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

Intercepts. In the purely periodic fit 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? We'll return to this question in section 2.6.

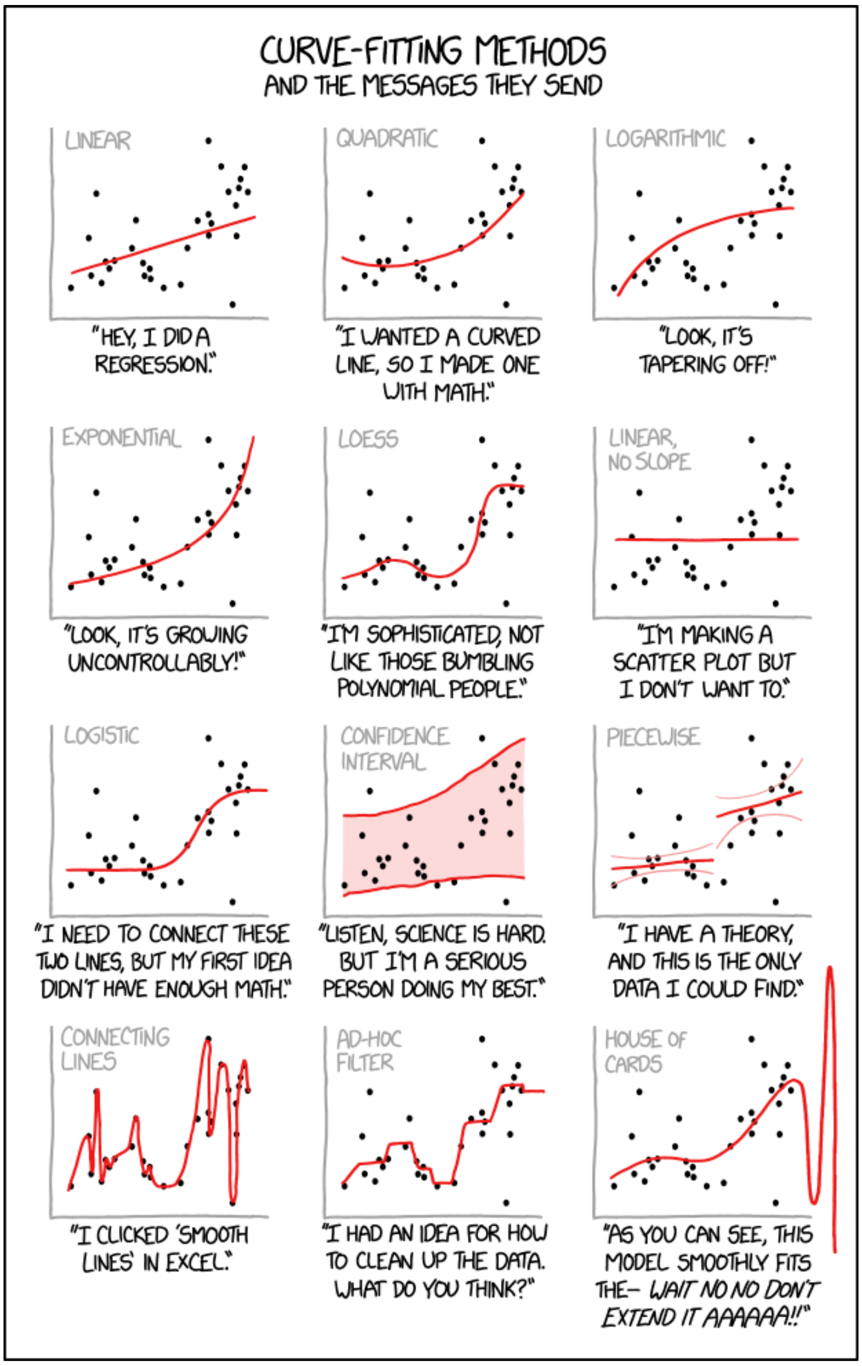
* ✻ *

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. Diagnosing a linear model

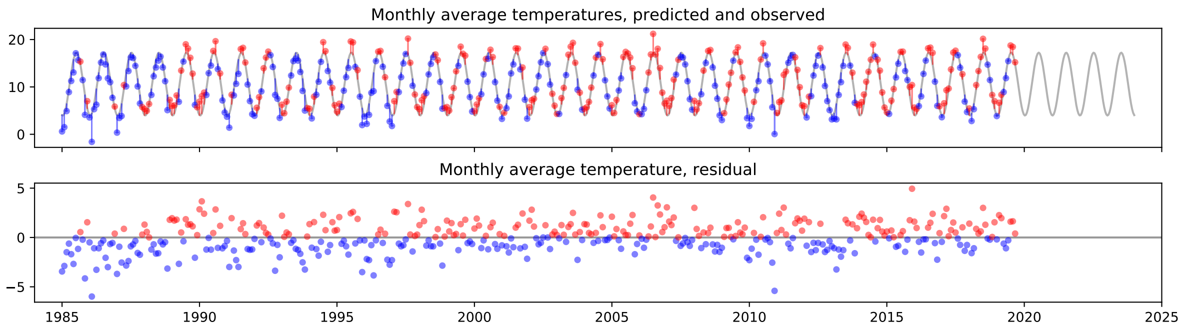
How do we know if we have the right features? It doesn't much matter if we have redundant features, since the parameter should come out close to zero, but what if we've missed an important feature?

It's often illuminating to plot the residual vector, to find out if we have missed any features worth incorporating. We should plot the residuals in every way we can, against any predictor variable we can think of, and if we see a systematic pattern then we should decide on a formula for it and add a term to our model.

Here's an example. For the climate data in section 2.2.4 we fitted the model

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

(this time we're writing the equation to make the residuals explicit). Here are two plots depicting ε: in the top plot the datapoints are colour coded by whether $\varepsilon > 0$ or < 0, and the bottom plot shows ε directly. These plots 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 the model fit better) by incorporating that feature into the model.



```
climate = pandas.read_csv('https://www.cl.cam.ac.uk/teaching/2021/DataSci/data/climate.csv
df = climate.loc[(climate.station=='Cambridge') & (climate.yyyy>=1985)]
t = df.yyyy + (df.mm-1)/12
temp = (df.tmin + df.tmax)/2

# fit the model
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)

# extract the residuals
pred = model.predict(X)
resid = temp - pred

with plt.rc_context({'figure.figsize': (15, 1.7*2.2), 'figure.subplot.hspace': 0.3}):
    fig,(ax1,ax2) = plt.subplots(nrows=2,ncols=1, sharex=True)

# Top plot: arrows and points
for (t1,pred1,resid1) in zip(t, pred, resid):
    ax1.arrow(t1, pred1, 0, resid1, alpha=0.5, color='blue' if resid1<=0 else 'red')
ax1.scatter(t, temp, s=15, alpha=0.5, c=numpy.where(resid<=0,'blue','red'))
# and a smooth line to show the fit
newt = numpy.linspace(1985, 2024, 1000)
newtemp = model.predict(numpy.column_stack([numpy.sin(2*numpy.pi*newt), numpy.cos(2*numpy.
ax1.plot(newt, newtemp, color='0.7', zorder=-1)

# Bottom plot: just the points
ax2.scatter(t, resid, s=15, alpha=0.5, c=numpy.where(resid<=0,'blue','red'))
ax2.axhline(0, color='0.6', zorder=-1)

ax1.set_xlim([1984, 2025])
ax1.set_title('Monthly average temperatures, predicted and observed')
```

```
ax2.set_title('Monthly average temperature, residual')
plt.show()
```

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 $e_1, \dots, e_K$ are covariates, Y is the random response, and σ and $\beta_1, \dots, \beta_K$ are unknown parameters. It is implicit that the Y_i are independent.

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 $i \in \{1, \dots, n\}$ 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 11) 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

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

and the log likelihood of the entire response vector y is

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

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 $[1, 1, \dots, 1]$. 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

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

2.5. Linear mathematics

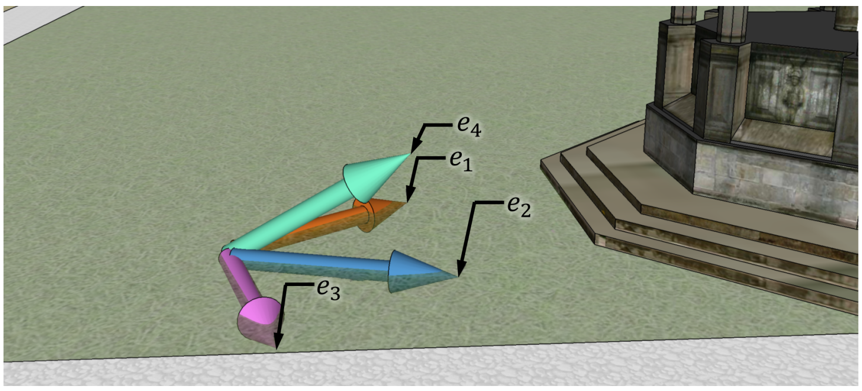
A linear model like the Iris model on page 23 is a vector equation,

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

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 the maths of linear algebra is abstract and can be applied to many other settings.[10] The appendix on page 79 presents the abstract maths.

For the purposes of this course, it's useful to study some linear algebra in order to understand how to interpret the parameters in a linear model. For some linear models, the parameter estimates we get out aren't reproducible—the fitting routine makes a seemingly arbitrary choice—but somehow it doesn't affect our predictions. This is a symptom of so-called 'confounded features'. By studying linear algebra, especially the idea of linear independence, we'll be able to craft reliable models that tell us what we're hoping to learn.

SPANS AND LINEAR INDEPENDENCE



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

$$\text{span} = \Big\{ \lambda_1 e_1 + \cdots + \lambda_K e_K \ : \ \lambda_k \in \mathbb{R} \ \text{ for all } k \Big\}.$$

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

$$\lambda_1 e_1 + \cdots + \lambda_K e_K = 0.$$

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

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

Computationally, to test if vectors are linearly independent, compute the rank of the matrix $[e_1, \dots, e_K]$. If rank $= K$ they are linearly independent, otherwise rank $< K$ and they are linearly dependent.

```
np.linalg.matrix_rank(np.column_stack([e_1,...,e_K]))
```

Exercise 2.3. In the snapshot of Trinity College Great Court shown above, three of the arrows are embedded in the lawn and the fourth points slightly up. Are the arrows linearly independent? If not, find a linearly independent subset that spans the same space.

[10]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).

The three arrows $\{e_1, e_2, e_3\}$ embedded in the lawn are linearly dependent—it looks like $e_2 = e_1 + e_3$, i.e. $e_1 - e_2 + e_3 = 0$. We could discard any one of them, and we'd still be able to get anywhere on the lawn with some linear combination of the other two.

As for e_4, it sticks up in the air, so there's no way it can be made up out of the others. So, for example, $\{e_1, e_2, e_4\}$ is a linearly independent subset. The span of these three is $\mathbb{R}^3$, the same as the span of the original four arrows.

∎

Exercise 2.4. Are the following five vectors linearly independent? If not, find a subset that is and that spans the same space.

$$\begin{aligned} e_1 &= [1,1,1,1] \\ e_2 &= [0,1,1,0] \\ e_3 &= [1,0,0,1] \\ e_4 &= [1,1,1,0] \\ e_5 &= [0,0,0,1] \end{aligned}$$

Just looking at these vectors, we can straight away see two linear relations:

$$\begin{aligned} e_2 + e_3 &= e_4 + e_5 \\ e_2 + e_3 &= e_1 \end{aligned}$$

We have a choice about what to discard. Let's discard e_5 (since $e_5 = e_2 + e_3 - e_4$, this won't reduce the span), and then discard e_3 (since $e_3 = e_1 - e_2$, this won't reduce the span) and we're left with $\{e_1, e_2, e_4\}$. Are these linearly independent? To check, suppose $\lambda_1 e_1 + \lambda_2 e_2 + \lambda_4 e_4 = 0$. Then

$$\lambda_1 \begin{bmatrix}1\\1\\1\\1\end{bmatrix} + \lambda_2 \begin{bmatrix}0\\1\\1\\0\end{bmatrix} + \lambda_4 \begin{bmatrix}1\\1\\1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\end{bmatrix}$$

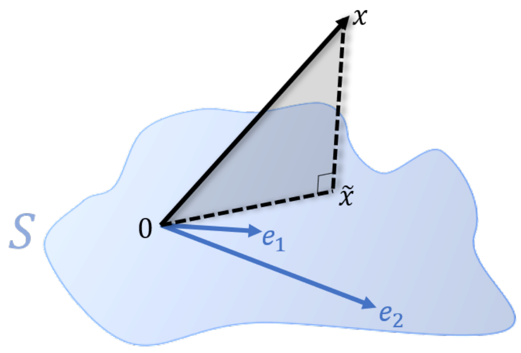
$$\Rightarrow \quad \left.\begin{aligned} &\lambda_1 + \lambda_4 = 0 \\ &\lambda_1 + \lambda_2 + \lambda_4 = 0 \\ &\lambda_1 + \lambda_2 + \lambda_4 = 0 \\ &\lambda_1 = 0 \end{aligned}\right\}$$

$$\Rightarrow \quad \lambda_4 = 0, \quad \lambda_2 = 0.$$

So they are linearly independent.

∎

PROJECTION ONTO FEATURE SPACE



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. 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,

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

Finding the $\hat{\lambda}_k$ is called least squares estimation, because the quantity 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.

2.6. Interpreting parameters

When we use least squares estimation to fit a linear model

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

we're looking for parameters that minimize the mean square error $\sum_i \varepsilon_i^2/n$. This is nothing other than projection, in the linear algebra sense. It's looking for a projection of the response vector y onto the subspace spanned by the feature vectors, known as the *feature space*. This linear algebraic view of least squares estimation can help us understand the coefficients we get out.

Interpretability of parameters. To interpret the parameters from a linear model,

1. Write out the predicted response for a representative datapoint. This helps us see what the parameters mean.
2. Write out the features and check if they're linearly dependent. If they are, the parameters have no intrinsic meaning; the parameters are said to be *non-identifiable*, and the features are said to be *confounded*.

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 $\{e, f, g\}$.

A warning: sklearn.linear_model.LinearRegression always returns parameters for *some* linear combination, and in the non-identifiable case it will make an arbitrary choice. It will nonetheless make a valid choice—whatever parameters it chooses, the model will still make exactly the same predictions.

Example 2.5.
For the climate data on page 27, consider these two models:

$$\text{temp} \approx \alpha + \beta_1 \sin(2\pi \text{t}) + \beta_2 \cos(2\pi \text{t}) + \gamma \text{t} \qquad \text{(Model A)}$$
$$\text{temp} \approx \alpha' + \beta_1' \sin(2\pi \text{t}) + \beta_2' \cos(2\pi \text{t}) + \gamma'(\text{t} - 2000) \qquad \text{(Model B)}$$

For model A we get $\hat\alpha = -60.5$ and for model B we get $\hat\alpha' = 10.6$. Interpret these parameters.

To figure out what the parameters in a linear model mean, let's write out predictions for a representative datapoint, say $\text{t} = 0$ *(January in 1BC, there being no year 0AD on the calendar) or* $\text{t} = 2000$.

model	t	*predicted temperature*
model A	t=*0*	$\alpha + \beta_2$
model B	t=*0*	$\alpha' + \beta_2' - 2000\gamma'$
	t=*2000*	$\alpha' + \beta_2'$

So $\hat\alpha$ is the predicted temperature in January 1BC (after removing the annual cycle), obtained by extrapolating a linear trend backwards from the present day. It's daft to believe such a wild extrapolation. Whereas $\hat\alpha'$ is the predicted temperature in January 2000, well within the region where we have data.

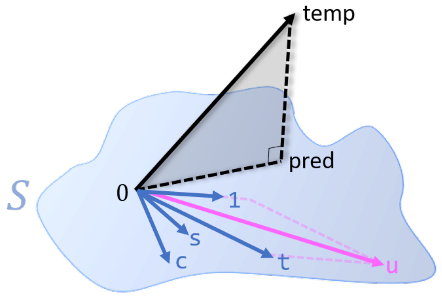
We can say something more about these two models. Let's list all the features:

$$1 = [1, 1, \dots]$$
$$t \quad \textit{given in the dataset}$$
$$s = \sin(2\pi \text{t})$$
$$c = \cos(2\pi \text{t})$$
$$u = \text{t} - 2000 \times 1.$$

The feature spaces of models A and B are identical, i.e.

$$\operatorname{span}\big(\{1,s,c,t\}\big) = \operatorname{span}\big(\{1,s,c,u\}\big).$$

Thus, when we fit the model (i.e. project temp *onto the feature space to get the model predictions), we'll obtain the same predictions in each case. The two models express exactly the same thing, they just use different coordinate systems to report their answers.*



∎

Example 2.6 (Contrasts).
Suppose we have measurements from two groups, group A and group B, and we assemble them into a spreadsheet where each row contains a measurement, and where there are two columns, g containing the group and y containing the measurement. How would you interpret the coefficients from these linear models?

$$y \approx \alpha 1[g=A] + \beta 1[g=B] \qquad \text{(model M1)}$$
$$y \approx \alpha + \beta 1[g=B] \qquad \text{(model M2)}$$
$$y \approx \alpha + \beta 1[g=A] + \gamma 1[g=B] \qquad \text{(model M3)}$$

For model M1: the predicted response for an individual from group A is α, and the predicted response for an individual from group B is β. Saying this the other way round, α and β are the predicted responses of the two groups.

For model M2: the predicted response for an individual from group A is α, and the predicted response for an individual from group B is $\alpha+\beta$, so β is the difference between the two groups.

For model M3: the predicted response for an individual from group A is $\alpha+\beta$, and the predicted response for an individual from group B is $\alpha+\gamma$. This is a bit confusing. Let's write out some example feature vectors.

$$\begin{array}{c} A \\ B \\ B \\ \\ \end{array} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \end{bmatrix} \approx \alpha \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \end{bmatrix} + \beta \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} + \gamma \begin{bmatrix} 0 \\ 1 \\ 1 \\ \vdots \end{bmatrix}$$

The three feature vectors are linearly dependent, because $1 = 1[g=A] + 1[g=B]$. Hence there is no unique way to write the projection of y onto the feature space as a linear combination of feature vectors. In other words, the parameters we end up with are arbitrary.

∎

Exercise 2.7. 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

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

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 2.8.

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://www.cl.cam.ac.uk/teaching/2021/DataSci/data/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

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

where eth is the vector of ethnicities. Write this model as a linear equation using one-hot coding. Are the parameters identifiable? If not, rewrite the model so that they are. Analyse whether this model suggests there is racial bias in policing actions.

Let's first understand how this model can describe racial bias. Write y_i for the response in record i, $y_i = 1$ if that record says the police found something, and $y_i = 0$ otherwise. The mean of y across a group is then

$$\text{mean response} = \frac{\#\text{stops where } y_i = 1}{\text{total }\#\text{stops}} = \mathbb{P}(\text{police find something}).$$

Suppose for example we find that β_{Black} is low, compared to the other groups. This means that the predicted value of y for people with eth=Black, $\alpha + \beta_{\text{Black}}$, is low, i.e. the probability that the police find something is low. Therefore the police must be stopping relatively more innocent people with eth=Black, which we can interpret as saying that the police are biased against eth=Black people.

ore rigorous
ent, see the logistic
ion in section 3.3

(It's a hack to treat a binary response as a real number with an implied Normal distribution. Section 3.3 offers a more refined probabilistic model. But the hack can still give us interesting answers about the distribution of response, and it's easy!)

With one-hot coding, the model is

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

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://www.cl.cam.ac.uk/teaching/2021/DataSci/data/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)
print('Missing values:', np.sum(pandas.isnull(police['officer_defined_ethnicity'])))
```

```
Asian    125646
Black    253315
Mixed      1644
Other     27809
White    532584
Missing values: 72917
```

So we're essentially only interested in the vectors

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

Clearly the five one-hot coded vectors sum up to one, *so the vectors are linearly dependent, and we could remove any of them to end up with a linearly independent set. For the sake of being fancy, let's remove* $1[\mathsf{eth{=}Asian}]$, *giving the model*

$$1[\mathsf{outcome{=}find}] \approx \alpha \mathsf{one} + \sum_{k \neq \mathit{Asian}} \beta_k 1[\mathsf{eth} = k].$$

The parameters are identifiable, since the feature vectors are linearly independent. To interpret the parameters from this second model, let's consider some representative datapoints:

for a person with	eth=Asian	*predicted y is*	α
	eth=Black		$\alpha + \beta_{Black}$
	eth=Mixed		$\alpha + \beta_{Mixed}$
	eth=Other		$\alpha + \beta_{Other}$
	eth=White		$\alpha + \beta_{White}$

Thus, α is the predicted response for a person with eth=Asian*, and each β_k measures the difference in response between* eth=k *people and* eth=Asian *people.*
∎

* ✻ *

Science looks for patterns in nature. Scientists look for interpretable patterns, and meaningful parameters in their models. Modern machine learning is also about finding patterns in data, but the models we use have millions of parameters and the patterns are beyond our comprehension. Scientists and social scientists have largely rejected neural networks as a way of learning about the world. This is a crisis for science. These machine learning models have the potential to work better—to make better predictions—but because they are uninterpretable, we don't know if we can rely on them to work in settings beyond their training set.

2.7. 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[11] 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*

[11]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.

3. Crafting a model

In this section we'll look in depth at several more advanced models. They're not more advanced as probability models—they don't use anything more sophisticated than the standard random variables like Categorical and Normal—but they use parameters in advanced ways so as to answer interesting questions.

3.1. Deep learning *

Any number of deep learning tutorials and blog posts talk excitedly about prediction accuracy and loss functions, but hardly any even mention probabilistic models of the sort we have been studying. And yet the best way to understand deep learning, and to be able to invent our own tools, is by seeing it as just another example of probabilistic supervised learning.

Example 3.1 (Classification).
The MNIST dataset consists of hand-written digits stored as 28×28 greyscale images, annotated with the digit they represent. Let $x_i \in \mathbb{R}^{28\times 28}$ be the ith image, and let $y_i \in \{0,1,\dots,9\}$ be its label.

For supervised learning, we need a probability model for the labels. Here's a general model for discrete outcomes. Let's imagine we have a function $\vec{p}(x\,;\,\theta)$ that takes an image x as input, as well as parameters θ, and returns a probability vector

$$\vec{p}(x_i\,;\,\theta) = \bigl[p_0(x_i\,;\,\theta),\dots,p_9(x_i\,;\,\theta)\bigr],$$

and let's use the Categorical random variable for our probability model:

$$Y_i \sim \text{Cat}\bigl(\vec{p}(x_i\,;\,\theta)\bigr)$$

i.e.

$$Y_i = y \text{ with probability } p_y(x_i\,;\,\theta), \quad y \in \{0,1,\dots,9\}.$$

The Categorical model is the simplest possible model for a random variable with discrete outcomes. We've used it before, exercise 1.11 page 16. In this model, all the cleverness is wrapped up inside the function $\vec{p}(x\,;\,\theta)$. This can be as complicated a function as we like, and θ could consist of millions of unknown parameters. We'd like to make it expressive enough so that, once the best parameters $\hat{\theta}$ have been fitted, $\vec{p}(x\,;\,\hat{\theta})$ puts nearly all its weight on the best guess for image x.

It's awkward to optimize functions over constrained domains—here we'd have to ensure we only pick θ that results in a valid probability vector, each probability ≥ 0 and all summing to one. It's easier to transform to an unconstrained space, using the softmax transform from section 1.2 page 7. Thus, let's seek a different function $\vec{s}(x\,;\,\theta) \in \mathbb{R}^{10}$, and then let

$$p_y(x\,;\,\theta) = \frac{e^{s_y(x\,;\,\theta)}}{e^{s_0(x\,;\,\theta)} + \cdots + e^{s_9(x\,;\,\theta)}} \quad \text{for } y \in \{0,1,\dots,9\}.$$

Then we're free to choose any parameters at all for $\vec{s}$, and we're guaranteed to end up with a probability vector $\vec{p}$.

To train this model we follow the standard maximum likelihood procedure: write out the log likelihood, and maximize it. The likelihood of an individual observation y is

$$\Pr_Y(y\,;\,x,\theta) = p_y(x\,;\,\theta) = \sum_{k=0}^{9} 1_{y=k}\, p_k(x\,;\,\theta).$$

(The last expression involves $1_{\{\cdot\}}$ which denotes the *indicator function*, $1_{\text{true}} = 1$ and $1_{\text{false}} = 0$. This is just an algebraic trick to say "only keep the p_y term where $y = k$". When we're aiming for fast code, a lookup like $p_y(\cdots)$ isn't helpful whereas a multiplication like $1_{y=k}p_k(\cdots)$ is better.) So the log likelihood of the whole dataset is

$$\log \Pr(y_1, \dots, y_n \,;\, \theta) = \sum_{i=1}^{n} \sum_{k=0}^{9} 1_{y_i=k} p_k(x_i \,;\, \theta).$$

We want to pick θ to maximize this. Equivalently, sticking a minus sign in front, we want to pick θ to minimize

$$L(\theta) = \sum_{i=1}^{n} \text{crossentropy}\Big(\text{onehot}(y_i), \text{softmax}\big(\vec{s}(x_i \,;\, \theta)\big)\Big)$$

where

$$\begin{aligned} \text{onehot}(y) &= \big[1_{y=0}, \dots, 1_{y=9}\big] \\ \text{softmax}(\vec{s}) &= \left[\frac{e^{s_0}}{e^{s_0} + \cdots + e^{s_9}}, \dots, \frac{e^{s_9}}{e^{s_0} + \cdots + e^{s_9}}\right] \\ \text{crossentropy}(\vec{q}, \vec{p}) &= -\big(q_0 \log p_0 + \cdots + q_9 \log p_9\big). \end{aligned}$$

This is known as “minimizing softmax cross-entropy loss with onehot coding”. It’s a sesquipadelian way of saying “I’m modelling this data with a Categorical random variable, the simplest possible model for discrete outcomes; I’m using the simplest possible transform to make the optimization easy; and I’m fitting using maximum likelihood estimation, the bog standard way of estimating parameters.”

* ✻ *

Deep learning is often presented as “we want to train a neural network to minimize a particular loss function”. Typically that loss function involves some sort of prediction term. (In the example above, the softmax can be interpreted as a prediction of the onehot term.) 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.
- If we think in terms of probability models, we know that unsupervised learning (in the form of generative modelling, section 1.6) and supervised learning are almost exactly the same thing, just with slightly different shapes of the dataset. If we think in terms of prediction and loss, it’s hard to even formulate what unsupervised learning is meant to achieve.

3.2. Numerical optimization with pytorch *

3.3. Quantifying a question: testing intersectionality *

3.4. Logistic regression *

These sections of the notes are non-examinable. They can be found in the online version.

Part II
Handling probability models

In this section we will study mathematical and computational tools for working with probability models.

Section 4 is about Bayes's rule and related maths. You have most likely learnt about the simple version of Bayes's rule and used it for calculations like "What is the probability that this patient is sick, given that the test came back with a positive test result?" In random variable notation, if $X \in \{\text{healthy}, \text{sick}\}$ is the patient's health, and $Y \in \{\oplus, \ominus\}$ is the test result, and if we know the distribution of X and of Y given X, then Bayes's rule says that

$$\mathbb{P}(X = \text{sick} \mid Y = \oplus) = \frac{\mathbb{P}(X = \text{sick})\ \mathbb{P}(Y = \oplus \mid X = \text{sick})}{\mathbb{P}(Y = \oplus)} \quad \text{when } \mathbb{P}(Y = \oplus) > 0.$$

In data science and machine learning we often want to apply Bayes's rule to continuous random variables. For a continuous random variable, the probability of it taking any exact value is zero. So what does Bayes's rule mean if the test result Y is a continuous random variable, for example a blood level reading? Or if the medical condition X is a continuous random variable, for example the patient's life expectancy? The equation above becomes divide-by-zero!

There is a simple fix: use Bayes's rule written in terms of likelihood:

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

Section 4.1 explains what this equation means. The goal is to become comfortable with writing down equations of this general type, to have intuition and to understand the pitfalls behind what the terms mean, and to not get flummoxed by the notation. The rest of section 4 has many examples of how to do algebra with continuous random variables. If you find section 4.1 too hand-wavey, work through these examples which build up the maths in a more rigorous way.

In data science and machine learning, we hardly ever use algebra to actually solve equations like Bayes's rule. We're usually interested in models that are far too rich for there to be any chance of algebraic solutions. Instead we use computers and random sampling, *computational methods*. This is the topic of section 5. All the maths examples in section 4 are great for reassuring ourselves that we understand the ideas, but the algebraic skills aren't needed to be a good data scientist.

* ✻ *

Here is some backstory behind sections 4 and 5. Suppose we have crafted a probability model. A probability model can be expressed in mathematical equations, or it can be expressed as a simulator that uses random number generators—i.e. as a piece of code, an algorithm. Computer scientists are well 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:[12]

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

It's natural to want to be able to analyse a simulator using maths, Dijsktra-style. For example, we might have implemented a climate simulator that uses random variables, so that every time we run it we get a slightly different output, and we might want to calculate the distribution of the output. For example, we might want to know the frequency of extreme weather events. This will typically involve integrals, for calculating probabilities and expectations, and section 4 provides many examples of this sort of calculation.

There is another stance, diametrically opposed to Dijkstra's. If the maths is too hard, as it usually is, we can just run our simulator, and we'll *see* its output directly. We run it many times, and we'll see a random sample drawn from the model output's distribution. If we run it enough times, and plot a histogram of the output, we have the distribution in front of us. This is called Monte Carlo simulation, the topic of section 5. (Perhaps the integration-based 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 powerful. There are some situations where the maths is too hard and computation is the only tool available. There are other situations, such

[12]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.

"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. We might for example get a thousand-fold speedup by replacing an inner simulation loop with a deft equation.

* ✻ *

In data science, *every probability model we come up with is wrong*. We should never delude ourselves that we have a true mathematical description of the world, only that we have a reasonable approximation to the data we're given. So, given a dataset, what's the point in inventing an approximate model and then analysing that model to death, when we could just analyse the dataset itself? Section 6 looks at methods that take a dataset to be the ground truth, as opposed to sections 4&5 which take a model to be the ground truth. This shift in perspective is in fact the cornerstone of machine learning and especially cross-validation. It's slippery concept, more philosophy than maths, and it shines a new light on the mathematical and computational methods in sections 4 and 5.

4. The maths of random variables

4.1. Bayes's rule for random variables

Here are two examples using Bayes's rule, one discrete and the other continuous. The discrete example is trivial, but it's helpful to go through it step by step so that we can carry the ideas across to the continous example where it's decidedly non-trivial.

Example 4.1 (Discrete conditioning). Someone runs this function and tells us they got $Y = 3$. What can we infer about their X?

```
def rxy_disc():
    x = numpy.random.randint(low=-5, high=6)  # from -5 to +5 inclusive
    y = numpy.random.binomial(n=6, p=(x/6)**2)
    return (x,y)
```

Example 4.2 (Continuous conditioning). Someone runs this function and tells us they got $Y = 0.6$. What can we infer about their X?

```
def rxy_cts():
    x = numpy.random.uniform(-1,1)
    y = numpy.random.normal(loc=x**2, scale=0.1)
    return (x,y)
```

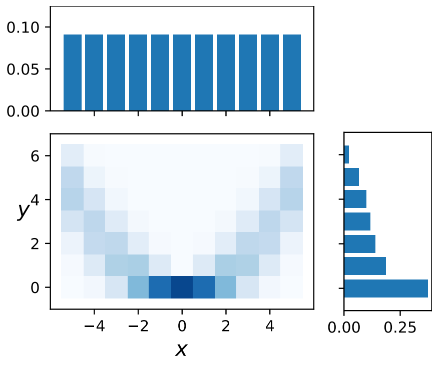
4.1.1. JOINT DISTRIBUTIONS AND MARGINALS

For the discrete example, we can immediately write out the joint distribution of X and Y:

This is a refresher of IA Probability lecture 7

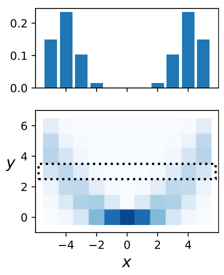
$$\begin{aligned}\mathbb{P}(X = x, Y = y) &= \mathbb{P}(X = x)\,\mathbb{P}(Y = y \mid X = x)\\ &= \frac{1}{11}\binom{6}{y}\left((x/6)^2\right)^y\left(1-(x/6)^2\right)^{6-y}.\end{aligned}$$

The plot below shows the joint distribution, and also the marginal distributions $\mathbb{P}(X = x)$ and $\mathbb{P}(Y = y)$ which we get by summing up columns and rows respectively.



We're told $Y = 3$, so we're in the $Y = 3$ row of the plot. We can just read off the values in this row, divide by the row sum so that the probabilities sum to one, and we get a distribution for X.

$$\mathbb{P}(X = x \mid Y = 3) = \frac{\mathbb{P}(X = x, Y = 3)}{\mathbb{P}(Y = 3)}$$

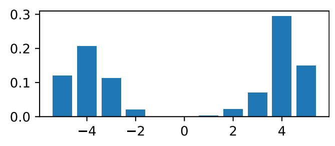


4.1.2. CONDITIONAL RANDOM VARIABLES

For a random variable X and an event C, we write $(X \mid C)$ for X *conditioned on* C. What we've just calculated is the distribution of the conditional random variable $(X \mid Y = 3)$, i.e. we've conditioned on the event $\{Y = 3\}$.

Here's code for generating conditional random variables. We can learn about the distribution of X given $Y = 3$ by simply generating lots of (X, Y) pairs, keeping those where $Y = 3$, and discarding the others. (This is called *rejection sampling*.)

```
def rx_given_y(yobs):
    while True:
        (x,y) = rxy_disc()
        if y == yobs: break
    return x

xsample = [rx_given_y(3) for _ in range(400)]
histogram(xsample)
```



This code `rx_given_y(3)` is a random number generator; it produces a different answer every time we call it. *In other words, it is a random variable.* It has everything we expect a random variable to have:

- $(X \mid Y = 3)$ has a sample space $\{-5, \dots, 5\}$, the same sample space as X
- $(X \mid Y = y)$ is a parametric random variable, with parameter y
- $(X \mid Y = y)$ has a probability mass function, $\text{pmf}(x) = \mathbb{P}\big((X|Y = y) = x\big)$. One can prove[13], using the mathematical tools from the rest of section 4, that $\text{pmf}(x) = \mathbb{P}(X = x \mid Y = y)$.

In keeping with the $\text{Pr}_\cdot(\cdot)$ notation from section 1.5, we ought to write the probability mass function as $\text{Pr}_{(X \mid Y=3)}(x)$, but it looks better and is easier to read if we write it

$$\text{Pr}_X(x \mid Y = y).$$

Now let's rewrite the equation for $\mathbb{P}(X = x \mid Y = 3)$. Writing $\text{Pr}_{X,Y}$ for the joint distribution, $\text{Pr}_{X,Y}(x, y) = \mathbb{P}(X = x, Y = y)$, and converting $\mathbb{P}(Y = y \mid X = x)$ to Pr notation, we get

$$\text{Pr}_X(x \mid Y = 3) = \frac{\text{Pr}_{X,Y}(x, 3)}{\text{Pr}_Y(3)} = \frac{\text{Pr}_X(x)\,\text{Pr}_Y(y \mid X = x)}{\text{Pr}_Y(3)}.$$

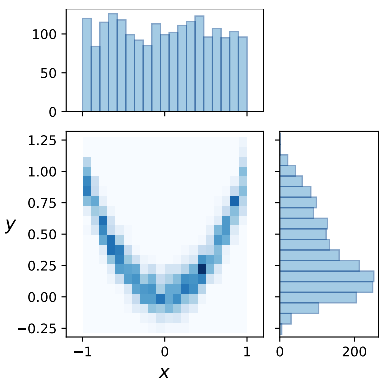
This is Bayes's rule, just written in terms of conditional random variables, and with Pr notation.

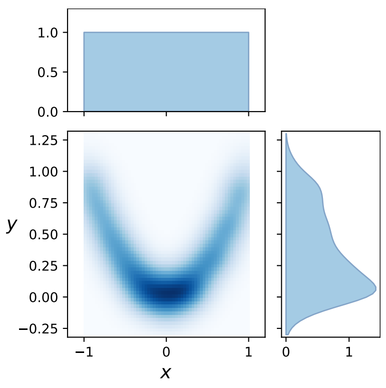
4.1.3. JOINT DISTRIBUTIONS AND MARGINALS (CONTINUOUS CASE)

Given a random tuple such as the function rxy_cts(), we can generate (X, Y) samples and plot a 2d histogram. The plot on the left shows the histogram.

```
xy_sample = [rxy_cts() for _ in range(2000)]
hist2d(xy_sample)
```

[13]Mathematicians prefer to go the other way: they define $(X \mid Y = y)$ via this function rather than by code, and then they'd have to work out a proof that our rejection sampling code achieves this pmf.


histogram of samples from rxy()


the joint density function

The plot on the left also shows marginal histograms. We can get them in two ways: either (a) build a matrix of counts, which we need anyway for the 2d histogram, and then add up row and column sums, and plot bar charts, or (b) take all the x values and plot a histogram of them, and likewise the y values. These two methods are clearly identical, because they each count the same thing.

```
x_sample = [x for (x,y) in xy_sample]
y_sample = [y for (x,y) in xy_sample]
hist(x_sample)
hist(y_sample)
```

The 2d histogram is an approximation to an idealized mathematical *joint density function*, plotted on the right hand side. This is a function of two variables, x and y, and it's written $\text{Pr}_{X,Y}(x,y)$. It's related to probabilities by

see IA Probability lecture 7

$$\mathbb{P}(x_1 \leq X \leq x_2,\, y_1 \leq Y \leq y_2) = \int_{x=x_1}^{x_2} \int_{y=y_1}^{y_2} \text{Pr}_{X,Y}(x,y)\, dy\, dx.$$

The larger the sample, and the finer the histogram bin size, the closer the histogram gets to the joint density function.

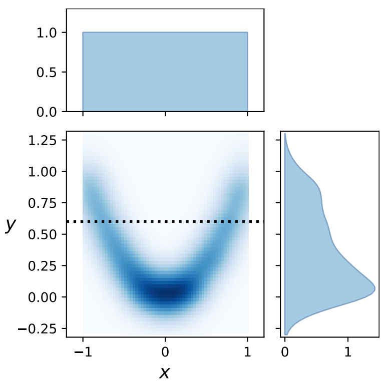
What about the probability density functions for X and Y? We'll write them as usual as $\text{Pr}_X(x)$ and $\text{Pr}_Y(y)$. We get them by integrating the joint density function, similar to how we summed row and column counts from the 2d histogram.

$$\text{Pr}_X(x) = \int_y \text{Pr}_{X,Y}(x,y)\, dy, \qquad \text{Pr}_Y(y) = \int_x \text{Pr}_{X,Y}(x,y)\, dx.$$

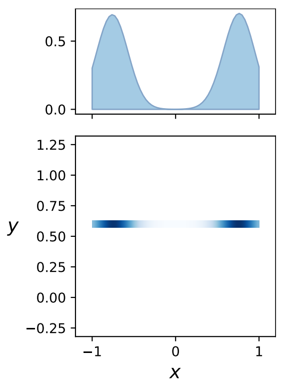
We referred to one-dimensional histograms as 'marginal histograms', for the obvious reason that we get them by writing row and column sums in the margins of the 2d table of counts. We also refer to the one-dimensional density plots as marginals. We also hear X or Y themselves, when taken in isolation, referred to as a marginal random variables.

4.1.4. CONDITIONAL RANDOM VARIABLES (CONTINUOUS CASE)

Now we can see by analogy how $(X \mid Y = 0.6)$ should be interpreted, in the case of example 4.2 where both X and Y are continuous random variables. It's what we get by taking an infinitesimal sliver of the joint density $\text{Pr}_{X,Y}(x,y)$ at $y = 0.6$. Another way of putting it: $(X \mid Y = 0.6)$ is the random variable whose density function, call it $f(x)$, is proportional to $\text{Pr}_{X,Y}(x, 0.6)$.



joint density of (X,Y)



slice at $y = 0.6$

To be precise, the density is $f(x) = \mathrm{Pr}_{X,Y}(x, 0.6)/\,\mathrm{Pr}_Y(0.6)$. The denominator is so that $f(x)$, like any other density function, integrates to 1. The correct scaling factor is $\mathrm{Pr}_Y(0.6)$ because of the formula for marginal densities:

$$\mathrm{Pr}_Y(y) = \int_x \mathrm{Pr}_{X,Y}(x,y)\,dx.$$

4.1.5. BAYES'S RULE WITH LIKELIHOOD NOTATION

Given a pair of random variables (X,Y) with joint density $\mathrm{Pr}_{X,Y}(x,y)$, we can take a horizontal slice through the joint density plot and get

$$\mathrm{Pr}_X(x \mid Y = y) = \frac{\mathrm{Pr}_{X,Y}(x,y)}{\mathrm{Pr}_Y(y)}$$

or we can take a vertical slice and get

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

Putting these two together, we get the random variable way of writing Bayes's rule:

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

Sometimes it's written with $\mathrm{Pr}_Y(y)$ expanded into an integral,

$$\mathrm{Pr}_Y(y) = \int_x \mathrm{Pr}_{X,Y}(x,y)\,dx = \int_x \mathrm{Pr}_X(x)\,\mathrm{Pr}_Y(y \mid X = x)\,dx$$

but it's rarely useful to try to calculate $\mathrm{Pr}_Y(y)$, as we shall see in the next section.

4.2. Calculations with Bayes's rule

Bayes's rule says that, for a pair of random variables (X, Y),

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

Typically we use it as

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

The idea is that we're looking for the marginal density for X, i.e. we're looking for a function of x, and so we might as well gather the non-x terms into a constant. The constant κ is whatever it must be to make the conditional density for X integrate to 1.

In some rare situations, usually only seen in textbooks and exam questions, we can find κ just from general knowledge of random variables, without having to do any calculus, as we'll see below.

Example 4.3.
Consider the pair of random variables (X, Y) generated by

```
def rxy(σ):
    x = numpy.random.uniform(−1,1)
    y = numpy.random.normal(loc=x**2, scale=σ)
    return (x,y)
```

Or, in maths notation,

$$X \sim \text{Uniform}[-1, 1], \qquad Y \sim N(X^2, \sigma^2).$$

Calculate $\Pr_X(x \mid Y = y)$.

Since $X \sim$ Uniform$[-1, 1]$, the density is constant over that range:

$$\Pr_X(x) = \frac{1}{2} \qquad \textit{for } -1 \le x \le 1.$$

Since Y is generated from X, with a Normal distribution, we can just write down its conditional density:

$$\Pr_Y(y \mid X = x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(y-x^2)^2/2\sigma^2}.$$

Bayes's rule says

$$\Pr_Y(y \mid X = x) = \kappa \Pr_X(x) \Pr_Y(y \mid X = x) = \kappa' e^{-(y-x^2)^2/2\sigma^2}.$$

Here we've gathered some more non-x terms into the constant. All that we're interested in is finding the density as a function of x. We know that densities must integrate to 1, so we can just write out what κ' must be:

$$\kappa' = 1 \Bigg/ \int_{x=-1}^{1} e^{-(x^2-y)^2/2\sigma^2}\, dx.$$

We can't go any further without some heavy calculus, unless we use computational methods to approximate the integral—the topic of section 5.

∎

Exercise 4.4. Consider the pair of random variables (Θ, Y) where $\Theta \sim$ Uniform$[0, 1]$ and $(Y|\Theta = \theta) \sim$ 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 $\theta \in [0,1]$,

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

We could calculate κ directly by integration, $\kappa = 1/\int_0^1 \theta\,d\theta$. Here's a different route, one which will serve us better for more complicated integrals.

We should have at our fingertips a collection of standard random variables. The Beta distribution 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 $(\Theta \mid Y=1)$ suggests looking at Beta$(2,1)$*, which has density*

$$\Pr(x) = \binom{2}{1} x \quad \textit{for } x \in [0,1]$$

and, since densitites must integrate to 1,

$$\binom{2}{1} = 1 \Big/ \int_{x=0}^{1} x\,dx.$$

In other words, our κ must simply be the constant in front of the Beta$(2,1)$ *density. We didn't have to do any integration, we just had to remember the standard random variable density functions, and rely on the fact that someone else has already figured out the correct constants.*

Here's the more streamlined way of writing all this: we found the density $\Pr_\Theta(\theta \mid Y=1) = \kappa\theta$, which is proportional to the density of Beta$(2,1)$*, hence $(\Theta \mid Y=1)$* is *a* Beta$(2,1)$ *random variable.*

∎

DENSITIES SUM TO ONE

Deft use of "densities sum to one" saves lots of hassle when applying Bayes's rule. Here are some more examples.

Exercise 4.5.
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 4.6 (Recognizing densities).
Let X be a random variable with density

$$\Pr_X(x) = \kappa x(1-x)^2, \qquad x \in [0,1]$$

for some constant κ. Name the distribution of X.

It's worth knowing the density functions for certain standard random variables. One distribution, very common in exam questions about Bayesian calculations, is the Beta distribution. This describes a continous $[0,1]$-valued random variable, with density function

$$\Pr_{\text{Beta}(a,b)}(x) = \binom{a+b-1}{a-1} x^{a-1}(1-x)^{b-1}$$

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

In this question, $\mathrm{Pr}_X(x)$ *has the same functional form (as a function of x) as* $\mathrm{Pr}_{\mathrm{Beta}(2,3)}(x)$*, the only difference is a constant at the front. But there's no choice about the constant: it has to be whatever will make the density integrate to one. There is only one constant that will work, namely the constant at the front of the Beta distribution. Therefore* $X \sim \mathrm{Beta}(2,3)$.

∎

Exercise 4.7. Suppose Y is a $[0,1]$-valued continuous random variable with density

$$\mathrm{Pr}_Y(y) = \kappa y^2(1-y)^3 .$$

Without using any calculus, find

$$\int_{y=0}^{1} y\, \mathrm{Pr}_Y(y)\, dy.$$

First, we should immediately recognize the density function for Y as that of a $\mathrm{Beta}(3,4)$ *distribution, so the normalizing constant is* $\kappa = \binom{6}{2} = 15$.

For the integral, we could just spot that it's the mean of the Beta distribution, and look it up on Wikipedia, and report the answer. Alternatively, rewrite it as

$$\int_{y=0}^{1} y\Big\{\kappa y^2(1-y)^3\Big\}\, dy = \frac{\kappa}{\binom{7}{3}} \int_{y=0}^{1} \binom{7}{3} y^{4-1}(1-y)^{4-1}\, dy\,.$$

The point of this rewriting is to make it so that the integrand is the density of a standard random variable, so it must integrate to 1. So the answer is $\kappa/\binom{7}{3} = 15/35 = 3/7$.

∎

4.3. Calculating density functions *

4.4. Random tuples *

These sections contain further examples of maths with random variables, along the lines of IA Probability. They are available online.

4.5. Conditional random variables *

4.6. Generating random variables *

These sections are non-examinable. They are available online.

5. Computational methods

5.1. Monte Carlo integration

Suppose we want to find $\mathbb{P}(X \in A)$ and we have code to generate X, but we haven't been able to derive the density function, or maybe we know the density but we can't solve the integral $\mathbb{P}(X \in A) = \int_{x\in A} \Pr_X(x)\,dx$. The obvious computational method is to simulate X many times, and count how often it lies in A. For example,

```
# 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
```

This is an example of a general tool called Monte Carlo integration.

First, a reminder about the expectation of a random variable. The expectation of a random variable X is

see IA Probability lectures 3 and 4 for the discrete case, lecture 7 for the continuous case

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

If h is some real-valued function, then $\mathbb{E}\,h(X)$ is[14]

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

tl;dr. We can approximate $\mathbb{E}\,h(X)$ 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 obscures the simplicity of the idea. To better convey the idea, we'll look at some different settings in which it's used, starting with the probability estimation with which we opened this section. Afterwards, on page 58, a note on efficient implementation in Python.

ESTIMATING PROBABILITIES

Suppose we want to estimate $\mathbb{P}(X \in A)$, and we have code to generate X. Define

$$h(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

and let $Y = h(X)$. This is a $\{0,1\}$-valued random variable. (This function h is called an *indicator function*, also written $h(x) = 1_{x\in A}$ or $h(x) = 1[x \in A]$.) Then

$$\begin{aligned} \mathbb{E}\,Y &= 0 \times \mathbb{P}(Y=0) + 1 \times \mathbb{P}(Y=1) \quad \text{defn. of expectation} \\ &= \mathbb{P}(Y=1) \\ &= \mathbb{P}(X \in A) \quad \text{from defn. of } Y. \end{aligned}$$

[14]There are two ways to get to $\mathbb{E}\,h(X)$: either use $\mathbb{E}\,h(X) = \int_x h(x)\Pr_X(x)\,dx$ directly, or let $Y = h(X)$, calculate $\Pr_Y(y)$, then use $\mathbb{E}\,Y = \int_y y \Pr_Y(y)\,dy$. These two methods give the same answer, a result known unkindly as 'The Law of the Unconscious Statistician', since it's easy to not even notice that there's a choice.

Alternatively, using the Monte Carlo estimate,

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

where $x_1, \dots, x_n$ is a sample from X. Thus

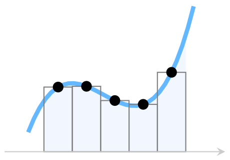
$$\mathbb{P}(X \in A) \approx \frac{1}{n}\sum_{i=1}^{n} 1_{x_i \in A}.$$

ESTIMATING AN INTEGRAL

Suppose we've been asked to find

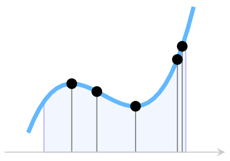
$$\int_{x=a}^{b} h(x)\, dx\,.$$

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 $X_1, \dots, X_n$, and approximate



$$= \text{width} \times \text{av. height} \approx (b-a)\Big(\frac{1}{n}\sum_{i=1}^{n} h(x_i)\Big).$$

To connect this to the abstract definition, let $X \sim U[a, b]$, 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 $(x_1, \dots, x_n)$ and then apply h to each item. In Python, the best coding style for this is vectorized.

for vectorized numpy code, see the course material for IA Scientific Computing, chapter 2

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.

5.2. Bayes's rule via computation

Bayes's rule says

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

It's for making inferences about probability models like this one:

```
def rxy():
    x = numpy.random.uniform(-1,1)
    y = numpy.random.normal(loc=x**2, scale=0.1)
    return (x,y)
```

where we know the underlying distribution of X, and of $(Y|X = x)$, and we've observed a value y for the random variable Y, and we want to make an inference about X. In other words, it's for finding the distribution of the conditional random variable $(X \mid Y = y)$. If we have more complicated code and it's too hard to derive formulae for $\Pr_X(x)$ or for $\Pr_Y(y)$, here's a computational approximation:

tl;dr. Generate a sample of values $(x_1, \dots, x_n)$ from 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, if we want to compute an expectation of the form $\mathbb{E}(h(X) \mid Y = y)$, use the weighted Monte Carlo estimate

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

As a special case, if we want to compute a probability such as $\mathbb{P}(X \in A \mid Y = y)$, by setting $h(x) = 1_{x \in A}$ in the above formula we find

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

For example, if we want to plot a histogram of the conditional random variable $(X \mid Y = y)$, we could split the x-axis into bins and plot a bar of height $\mathbb{P}(X \in \text{bin} \mid Y = y)$ for each of the bins.

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. The method described above isn't the most efficient, but it's easy to use and doesn't depend on advanced theory.

Exercise 5.1. Consider the probability model

```
def ry():
    θ = np.random.uniform(0,1)
    y = np.random.binomial(n=3, p=θ)
    return y
```

Suppose we have observed $Y = 2$ and we want to know the likely range of Θ. Plot a histogram of $(\Theta \mid Y = 2)$.

for the mathematical solution see exercise 4.4

The relevant distributions are

$$\Pr_\Theta(\theta) = 1, \qquad \Pr_Y(y \mid \Theta = \theta) = \binom{3}{y}\theta^y(1-\theta)^{3-y}$$

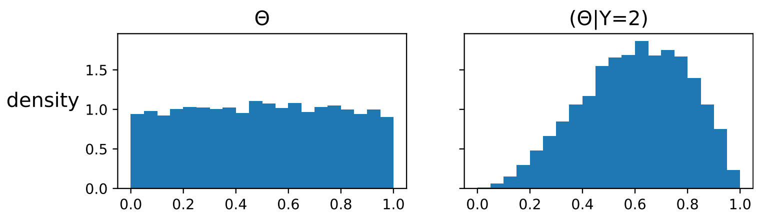
The heavy-handed way to draw a histogram is to manually split the θ-axis into bins, and compute $\mathbb{P}(\Theta \in bin \mid Y = 1)$ separately for each bin:

```
def prob_θ_given_y(θlo, θhi, y, n=10000):
    # Generate a sample of size n from the distribution of Θ
    θsamp = np.random.uniform(low=0, high=1, size=n)
    # Define weights w_i proportional to Pr_Y(y | Θ = θsamp_i)
    # (Note: the constant factor (2 choose y) cancels out when we normalize w, so ignore it)
    w = θsamp**y * (1−θsamp)**(2−y)
    w = w / np.sum(w)
    # Use weights to estimate the probability that Θ is in bin [θlo, θhi)
    return np.sum(w * np.where((θsamp >= θlo) & (θsamp < θhi), 1, 0))

breaks = np.linspace(0, 1, 20)
lows,highs = breaks[:−1], breaks[1:]
barheights = [prob_θ_given_y(θlo=lo, θhi=hi, y=2) for lo,hi in zip(lows, highs)]

plt.bar(x=(lows+highs)/2, height=barheights, width=1/len(breaks))
```

More elegant to notice that we can use the same θsamp *for each bin, and that the standard histogram command knows what to do with weights.*

```
θsamp = np.random.uniform(low=0, high=1, size=10000)
w = θsamp * (1−θsamp)
plt.hist(θsamp, weights=w, type='density', bins=20)
```

This produces a density-type histogram, in which the bar heights are rescaled so that the plot integrates to 1. This way, if we make the bins finer (and increase the number of Θ *samples in proportion), we'll approach a density plot.*



Quick sanity check: is it reasonable that that the distribution of $(\Theta \mid Y = 2)$ *should be pushed away from uniform and towards* $\theta = 2/3$*? The distribution of* Y *is* $\text{Binom}(3, \Theta)$*, so the expected value of* Y *is* 3Θ*. We observed* $Y = 2$*, so a reasonable guess is* $\Theta \approx 2/3$*.*

∎

DERIVATION

You don't need to know how to derive the computational approximation to Bayes's rule, 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 5.1, 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,

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

* ✻ *

The computational method shown here is a special case of a general method called importance sampling. Importance sampling is a way to approximate expectations of one random variable using samples drawn from a different random variable. In this case, we want to approximate expectations of $(X \mid Y = y)$ and we're using samples drawn from X. The general method is described in section 5.3.

5.3. Importance sampling *

5.4. Deep generative models *

5.5. Application: ray tracing *

These sections are non-examinable. They can be found online.

6. 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 sections 4 and 5, 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”[15], 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 book, the concept of the empirical distribution (section 6.3) is fundamental for frequentist inference in Part III, including cross-validation. This concept needs virtually no maths, but it is subtle, hence the meandering build-up in which we revisit ideas from section 5.

[15]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?”*

6.1. The empirical cumulative distribution function

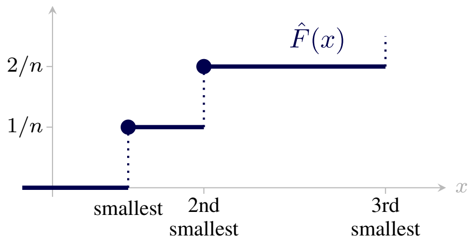
Given a dataset of n numerical values $(x_1, x_2, \dots, x_n)$, the *empirical cumulative distribution function* of the dataset is

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

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

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

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



In Python, using numpy and matplotlib,

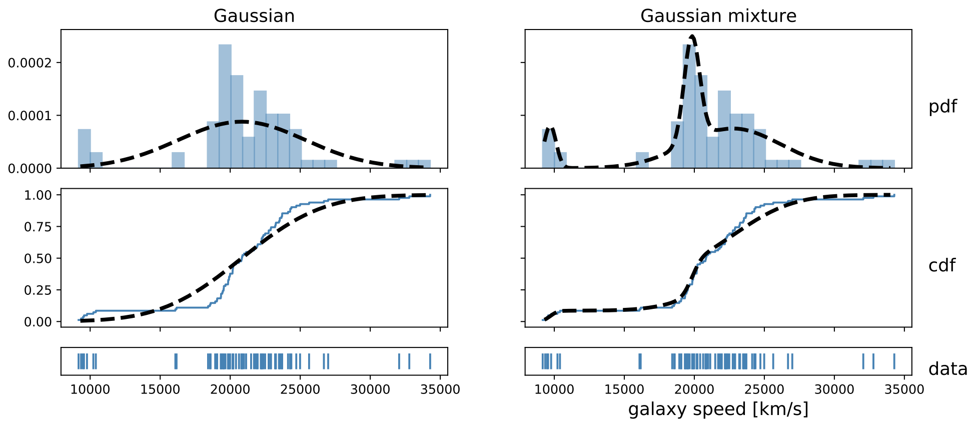
```
x = [...]   # the dataset, stored as a list
ef = np.arange(1,len(x)+1)/len(x)
plt.plot(np.sort(x), ef, drawstyle='steps-post')
```

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

fitting a generative model: section 1.6 page 15

When we fit a probability distribution to a dataset using maximum likelihood, we'd expect that the cdf of the fitted distribution's cdf should be reasonably close to the ecdf of the dataset. Here's an illustration, the Galaxies dataset from exercise 1.10. We're showing here the fitted Gaussian mixture model with three components from that exercise, and also a simple fitted Gaussian.[16]



```
url = 'https://www.cl.cam.ac.uk/teaching/2021/DataSci/data/galaxies.csv'
galaxies = pandas.read_csv(url)['speed'].values

fig,(ax1,ax2,ax3) = plt.subplots(3,1, sharex=True, gridspec_kw={'height_ratios':[1,1,.2]})

```

[16]The top plots show the fitted pdfs, superimposed on the *density histogram* of the dataset. The density histogram is a histogram in which the bar heights are scaled so that the total area of the histogram is 1; this means we get a similar plot however we choose the bin sizes, and the plots are comparable to probability density functions.

```
# Plot the ecdf and the fitted cdf
ef = np.arange(1, len(galaxies)+1)/len(galaxies)
ax1.plot(np.sort(galaxies), ef, drawstyle='steps-post', color='blue')
x = numpy.linspace(9300, 34000, 200)
ax1.plot(x, fitted_cdf(x), color='black', lw=3, linestyle='dashed')

# Plot the density histogram and the fitted pdf
ax2.hist(galaxies, bins=30, density=True, fc='blue', alpha=.3)
ax2.plot(x, fitted_pdf(x), lw=3, color='black', linestyle='dashed')

# Plot the raw data (a "rug plot")
for x in galaxies:
    ax3.plot([x,x],[0,1], color='blue')
```

The maximum likelihood procedure is one way to fit a parametric probability model to a dataset—but its goal is *not* to make the fitted cdf as close as possible to the dataset's ecdf. If we wanted this we'd have to first decide what we mean by 'close' (e.g. we could measure the area between the cdf curve and the ecdf), and then run an appropriate optimization; and there's no reason why this optimization should match up with the maximum likelihood optimization. Nonetheless, a maximum likelihood fit typically produces a cdf that's reasonably close to the ecdf, and if it's not close then we should investigate whether we need to pick a richer probability model.

6.2. Custom distributions

In generative modelling, we have a dataset $x_1, \dots, x_n$ and we want to find a distribution that might have generated it. We typically start from an off-the-shelf model, such as a Gaussian or Exponential random variable. What if none of these standard models fit our dataset well?

It's easy to *create* a random variable from scratch: all we have to do is draw the cumulative distribution function we want, and say "let this be my random variable". We can differentiate the cdf to get the pdf (in case the function we drew has parameters that we want to fit with maximum likelihood estimation), and we can translate the cdf into code (in case we want to use Monte Carlo methods). Some examples are shown below, to get into the spirit of treating cdfs and code as two sides of the same thing.

(Why start by drawing a cdf? It's fiddly to draw a custom pdf, because of 'densities sum to one'. It's tricky to start with code, since the link between code and dataset is indirect. It's easy to start with a cdf, since any function that starts at 0 and increases to 1 is a valid cdf.)

Since we can design a completely custom random variable with whatever cdf we like, it's natural to ask: what is the random variable that best fits our dataset? The answer is embarrassingly simple: plot the ecdf of the dataset, and let *that* be our custom cdf. The code for this random variable is also embarrassingly simple:

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 cumulative distribution function is

$$\mathbb{P}(X^* \leq x) = \frac{1}{n}\Big(\text{how many items there are } \leq x\Big)$$

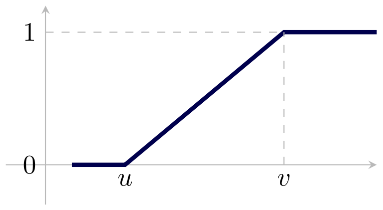
i.e. the cdf of X^* is the ecdf of the dataset.

So, if your boss asks you "You're a clever data scientist—tell me what distribution this dataset come from", you just answer "It comes from the distribution of the dataset."

ILLUSTRATIONS OF CUSTOM CDF

To understand where X^* comes from, here are some illustrations of custom cdfs and the corresponding code.

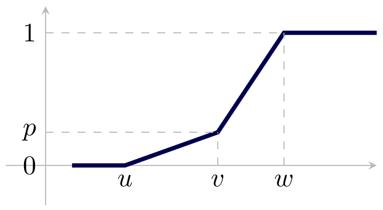
The Uniform$[u, v]$ random variable is a basic building block, and we need to be able to recognize its shape.



```
def rx(u,v):
    return np.random.uniform(u,v)
```

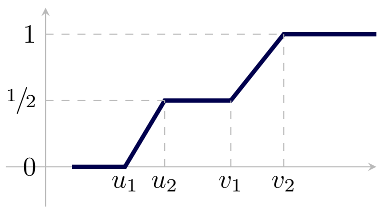
a more systematic approach for turning cdf into code is the inversion method, section 4.6

In the next plot, the cdf goes through (v, p), hence we want a random variable X with $\mathbb{P}(X \leq v) = p$ and $\mathbb{P}(X > v) = 1 - p$. We can achieve this by np.random.choice and an if statement. On each side of v the cdf is a straight line, and the pdf is the derivative of the cdf, so the pdf is constant over the interval (u, v) and also over (v, w). A constant pdf corresponds to a Uniform random variable.



```
def rx(u,v,w,p):
    k = np.random.choice(['leqv','gtv'], [p,1-p])
    if k == 'leqv':
        return np.random.uniform(u,v)
    else:
        return np.random.uniform(v,w)
```

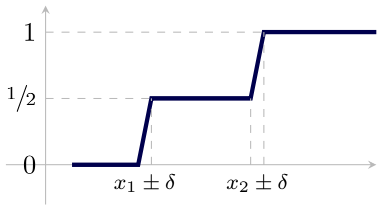
The next two are minor variations.



.

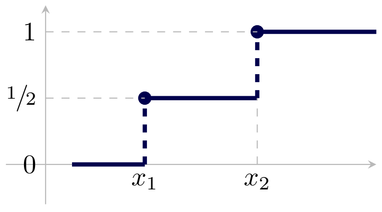
```
def rx(u₁,u₂,v₁,v₂):
    k = np.random.choice(['low','high'])
    if k == 'low':
        return np.random.uniform(u₁,u₂)
    else:
        return np.random.uniform(v1,v₂)
```

np.random.choice(xlist) returns a random element of xlist, each element equally likely



```
def rx(x₁,x₂,δ):
    k = np.random.choice(['low','high'])
    if k == 'low':
        return np.random.uniform(x₁−δ,x₁+δ)
    else:
        return np.random.uniform(x₂−δ,x₂+δ)
```

In the limit as $\delta \to 0$, a Uniform$[x_1 - \delta, x_1 + \delta]$ collapses to just x_1. Thus



```
def rx(x₁,x₂):
    k = np.random.choice(['low','high'])
    if k == 'low':
        return x₁
    else:
        return x₂
```

This last example was a random variable whose cdf matches the ecdf of the length-2 dataset $\{x_1, x_2\}$. The code can be reduced a one-liner:

```
def rx(x₁,x₂): return np.random.choice([x₁,x₂])
```

When we generalize to an arbitrary list we get X^* described above.

6.3. The empirical distribution

Empirical distribution. 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 probability mass function is

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

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

$$\mathbb{P}(X^* \in A) = \frac{1}{n}\Bigl(\text{number of datapoints } x_i \text{ that are in } A\Bigr) \quad \text{for every set } A.$$

(This statement makes sense whatever the type of values in the dataset, whereas the statement ‘cdf of X^* equals ecdf of dataset’ only makes sense for numerical random variables.)

In Python, generate a single value from X^* with

```
np.random.choice(xlist)
```

and generate m independent values with

```
np.random.choice(xlist, replace=True, size=m)
```

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

- 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. Isn’t it daft to shoehorn the data into a standard distribution, when the perfect fit is staring us in the face? If we use some standard distribution, then we’re bringinging in a belief that doesn’t reside in the dataset, which Occam’s razor tells us not to.
- When the true distribution is intractable, we can approximate it by taking a sample and working with the empirical distribution of that sample. Instead of getting bogged down with integrals, we just use Monte Carlo integration.
- A word like ‘variance’ comes from probability theory, but we often say things like ‘the variance of a dataset’. Strictly speaking, we’re referring to the variance of the empirical distribution of that dataset.

Now for some illustrations of the last two points. The first point is philosophy, not maths or computing, and we’ll return to it when we discuss cross validation in part III.

STATISTICS OF A DATASET

Exercise 6.1. Let X^* be the empirical distribution of a dataset $(x_1,\dots,x_n)$, where the x_i are real numbers. Let $\mu = \mathbb{E}\,X^*$ and let $\sigma^2 = \operatorname{Var} X^*$. Show that

$$\mu = \frac{1}{n}\sum_{i=1}^n x_i \qquad \sigma^2 = \frac{1}{n}\sum_{i=1}^n (x_i-\mu)^2\,.$$

It’s easiest to write $X^* = g(K)$ where K is a discrete uniform random variable on $\{1,2,\dots,n\}$ and $g(k)$ picks out the kth item in the dataset. Then, from the Law of the Unconscious Statistician,

law of the unconscious statistician, page 57

$$\mathbb{E}\,X^* = \mathbb{E}\,g(K) = \sum_{k=1}^n g(k)\Pr_K(k) = \frac{1}{n}\sum_{k=1}^n x_k.$$

The result for variance is similar.

∎

MONTE CARLO REINTERPRETED

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 $x_1, \dots, x_n$ is a sample drawn from distribution X and h is some arbitrary real-valued function. If we let X^* be the empirical distribution of that sample,

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

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.

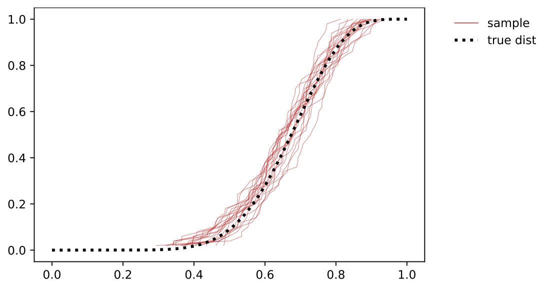
CONVERGENCE OF ECDF TO CDF

If a particular random variable X truly were the distribution from which the x_i were drawn, then the Monte Carlo approximation suggests cdf$(x) \approx$ ecdf(x), since

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

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 78



```
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()
```

6.4. Closeness of fit and KL divergence *

6.5. Empirical versus parametric modelling *

These sections are non-examinable. They can be found online.

Part III
Inference

Part IV
Advanced probability modelling

These parts of the notes will be released during term.

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 $X \sim \text{Pois}(\lambda)$ 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 $X \sim \text{Pois}(\lambda t)$.

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

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

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

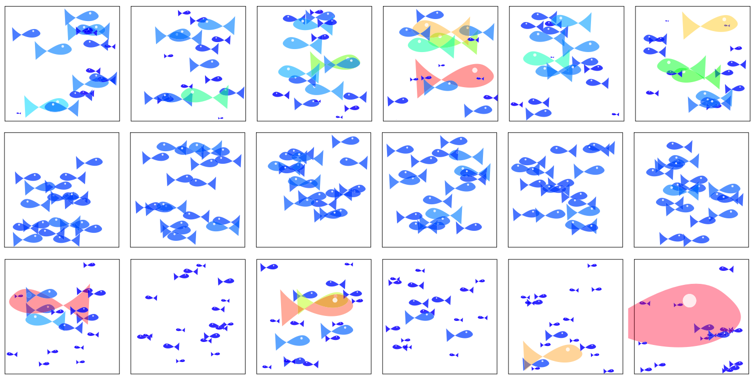
Pareto and lognormal: Some natural phenomena, like sizes of forest fires, or insurance claims, or Internet traffic volumes, or stock market crashes, have the characteristic that there are events of wildly different sizes. This tends to cause problems for simulations and forecasting, since the entire outcome can hinge on one 'black swan' event[17]. 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

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

The mean and variance become ∞ for small α,

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

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 $\{1, 2, \ldots, n\}$ 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

[17]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[18] 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 $v_1, \dots, v_n$ be vectors in a vector space and $\lambda_1, \dots, \lambda_n$ be scalars. Then the vector $\lambda_1 v_1 + \cdots + \lambda_n v_n$ is called a *linear combination* of $v_1, \dots, v_n$. 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 $\{v_1, \dots, v_n\}$, 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 $v_1, \dots, v_n$ 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 $e_1, \dots, e_n$ 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 $\{e_1, \dots, e_n\}$ of a vector space, it can be proved that any vector x can be uniquely written as

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

The n-tuple $(\lambda_1, \dots, \lambda_n)$ 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 $\text{Im}(v \cdot v) = 0$ and $\text{Re}(v \cdot v) \geq 0$.

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

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

[18] 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 $\lambda_1, \ldots, \lambda_n$ and vectors $v_1, \ldots, v_n, w$ 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 $\{e_1, \ldots, e_n\}$ is an orthonormal system in an inner product space, then for every vector x in the span of $\{e_1, \ldots, e_n\}$, the coordinates of x are given by
$$x = \sum_{i=1}^{n} (x \cdot e_i)\, e_i.$$
17. $\|u + v\| \leq \|u\| + \|v\|$ 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[19]

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

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:

$$\min_{\lambda_1, \lambda_2} \left\| x - (\lambda_1 e_1 + \lambda_2 e_2) \right\|^2.$$

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])
```

[19] 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.

∎

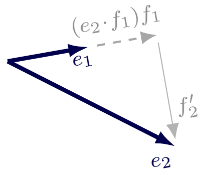
> Example B.5 (Closest point via explicit projection).
> Let $x = [1,2,3]$, and let $\tilde{x}$ be the projection onto the subspace spanned by $e_1 = [1,1,0]$ and $e_2 = [1,0,-1]$. Create an orthonormal basis out of $\{e_1, e_2\}$, and thence find the coordinates of $\tilde{x}$ with respect to the basis $\{e_1, e_2\}$.
>
> Hint: first use Useful Property 16 on page 80 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.