Statistics and Machine Learning

Yair Mau

Table of contents

home

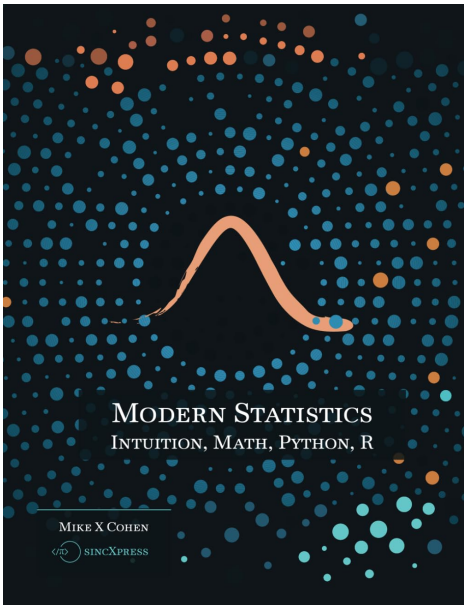
I'm teaching myself statistics and machine learning, and the best way to truly understand is to use the new tools I've acquired. This is what this website is for. It is mainly a reference guide for my future self.

books

These are the books that I've read and recommend.

Modern Statistics: Intuition, Math, Python, R

by Mike X Cohen



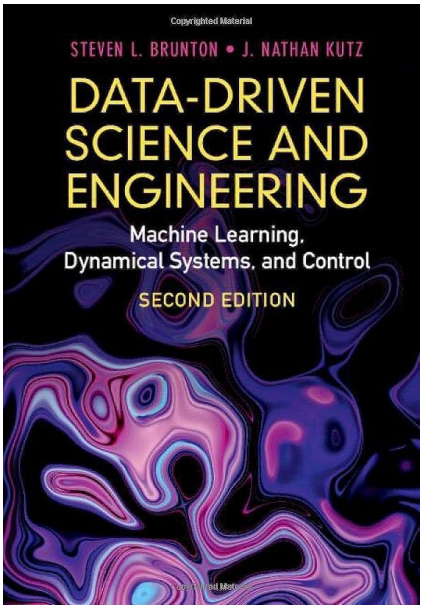
Github

This is a really approachable book, the author has a very nice conversational style, and I enjoyed it a lot. Highly recommended

---

Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control

by Steven L. Brunton, J. Nathan Kutz

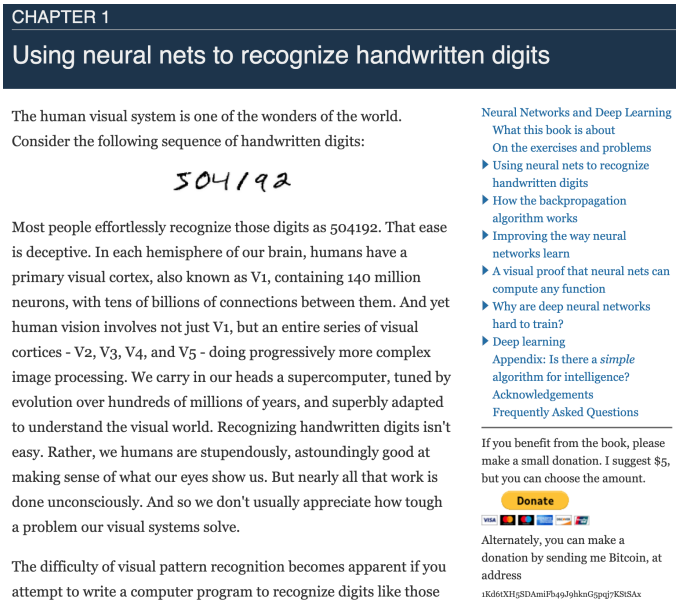


The whole book is available in this website.

This is the sort of books that is suitable for those who already know the subject. I would not recommend it as a first read. In any case, some chapters gave me new intuition on the subject. I do highly recommend Steve Brunton's youtube channel, it's fantastic.

Neural Networks and Deep Learning

by Michael Nielsen



CHAPTER 1

Using neural nets to recognize handwritten digits

The human visual system is one of the wonders of the world. Consider the following sequence of handwritten digits:

504192

Most people effortlessly recognize those digits as 504192. That ease is deceptive. In each hemisphere of our brain, humans have a primary visual cortex, also known as V1, containing 140 million neurons, with tens of billions of connections between them. And yet human vision involves not just V1, but an entire series of visual cortices - V2, V3, V4, and V5 - doing progressively more complex image processing. We carry in our heads a supercomputer, tuned by evolution over hundreds of millions of years, and superbly adapted to understand the visual world. Recognizing handwritten digits isn't easy. Rather, we humans are stupendously, astoundingly good at making sense of what our eyes show us. But nearly all that work is done unconsciously. And so we don't usually appreciate how tough a problem our visual systems solve.

The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those

Neural Networks and Deep Learning
What this book is about
On the exercises and problems
Using neural nets to recognize handwritten digits
How the backpropagation algorithm works
Improving the way neural networks learn
A visual proof that neural nets can compute any function
Why are deep neural networks hard to train?
Deep learning
Appendix: Is there a *simple* algorithm for intelligence?
Acknowledgements
Frequently Asked Questions

If you benefit from the book, please make a small donation. I suggest $5, but you can choose the amount.

Donate

Alternately, you can make a donation by sending me Bitcoin, at address

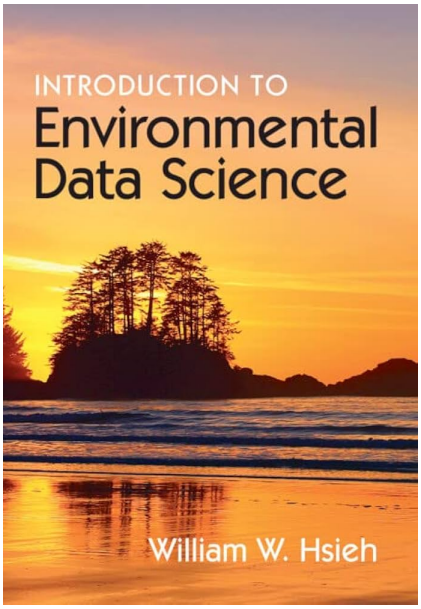
1Kd6tXH5SDAmiFb49J9hknG5pqj7KStSAx

This is an online book, freely available here. It can be tiring to read a whole book on a computer screen, so you can find Anton Vladyka's LaTeX rendition of this book in his GitHub repository. I wanted to read the pdf in my tiny kindle reader, so I recompiled Anton's LaTeX code to make it fit the screen, and on the way changed the font, and corrected typos here and there. Overleaf project. Download pdf.

Nielsen writes very well, I really enjoyed this book. The part on backprogation is a bit confusing, I would recommend watching 3b1b's youtube video on that.

Introduction to Environmental Data Science

by William W. Hsieh



This book's best quality is that it covers a bunch of topics, methods, techniques. It is not a good book to learn concepts for the first time, it's more useful as a menu of what exists, and maybe a brief reminder of topics you studied in the past but forgot. The "environmental" aspect is completely incidental, in my opinion. Hsieh brings examples from the Environment, but you don't need to have a background in environmental science to be able to read it.

websites

Dr. Roi Yehoshua's tutorials

Really good tutorials, you should check this out:
https://towardsdatascience.com/author/roiyeho/

It seems the he wrote a book, I haven't read it, but should be good:
Machine Learning Foundations, Volume 1: Supervised Learning

Part I

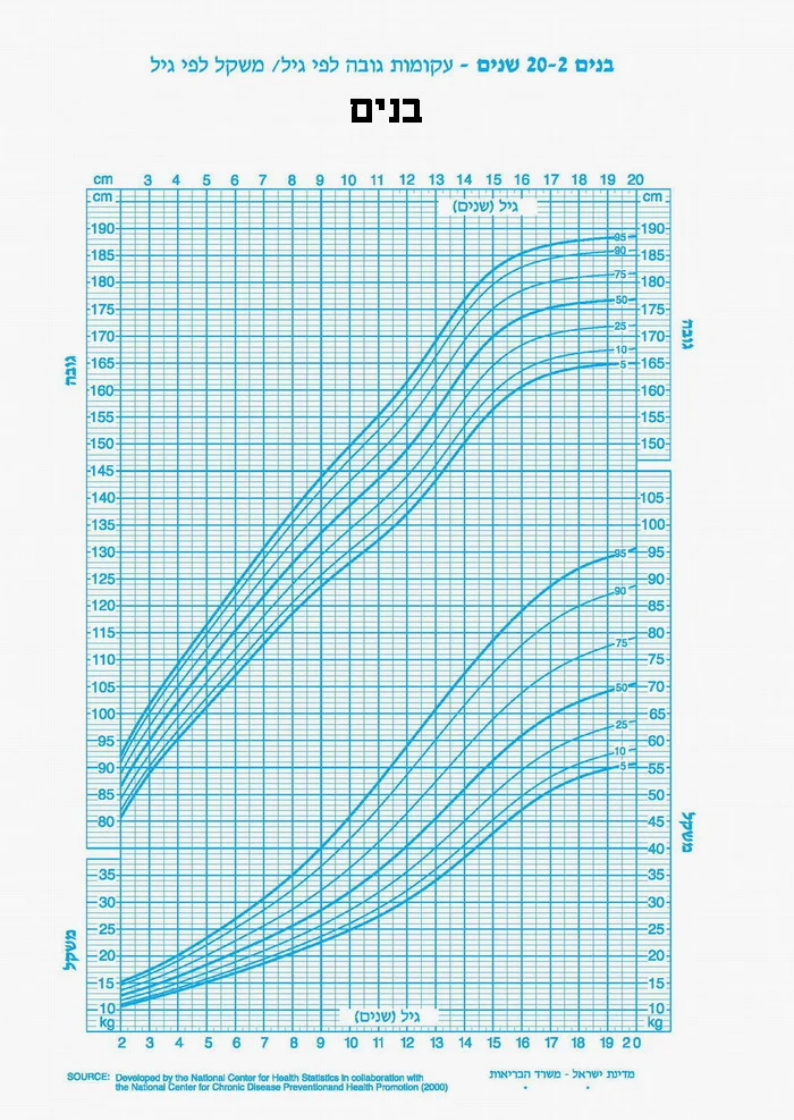
data

1 height data

I found growth curves for girls and boys in Israel:

- url girls, pdf girls
- url boys, pdf boys
- url both, png boys, png girls.

For example, see this:



I used the great online resource Web Plot Digitizer v4 to extract the data from the images files. I captured all the growth curves as best as I could. The first step now is to get interpolated versions of the digitized data. For instance, see below the 50th percentile for boys:

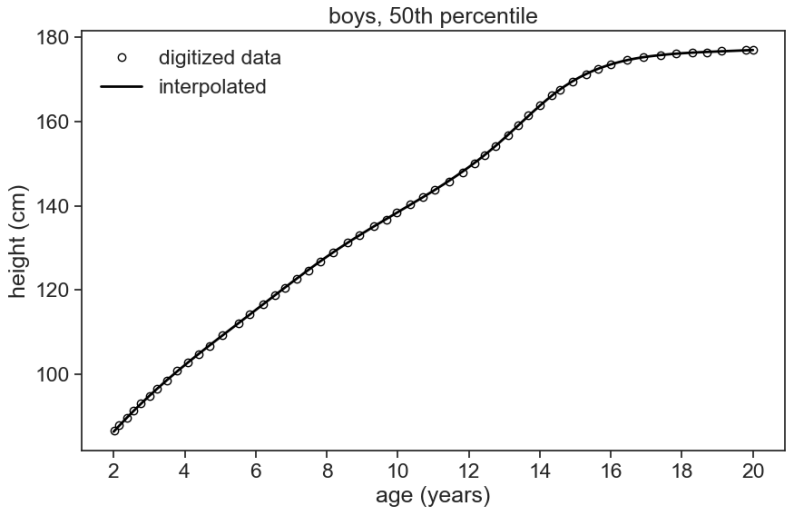
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.optimize import curve_fit
from scipy.special import erf
from scipy.interpolate import UnivariateSpline
import matplotlib.animation as animation
from scipy.stats import norm
import plotly.graph_objects as go
import plotly.io as pio
pio.renderers.default = 'notebook'
# %matplotlib widget
```

```
age_list = np.round(np.arange(2.0, 20.1, 0.1), 1)
height_list = np.round(np.arange(70, 220, 0.1), 1)
```

```
df_temp_boys_50th = pd.read_csv('../archive/data/height/boys-p50.csv', names=['age','height'])
spline = UnivariateSpline(df_temp_boys_50th['age'], df_temp_boys_50th['height'], s=0.5)
interpolated = spline(age_list)
```

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(df_temp_boys_50th['age'], df_temp_boys_50th['height'], label='digitized data',
        marker='o', markerfacecolor='None', markeredgecolor="black", markersize=6, linestyle='
ax.plot(age_list, interpolated, label='interpolated', color="black", linewidth=2)
ax.set(xlabel='age (years)',
       ylabel='height (cm)',
       xticks=np.arange(2, 21, 2),
       title="boys, 50th percentile"
       )
ax.legend(frameon=False);
```



Let's do the same for all the other curves, and then save them to a file.

```
col_names = ['p05', 'p10', 'p25', 'p50', 'p75', 'p90', 'p95']
file_names_boys = ['boys-p05.csv', 'boys-p10.csv', 'boys-p25.csv', 'boys-p50.csv',
                   'boys-p75.csv', 'boys-p90.csv', 'boys-p95.csv',]
file_names_girls = ['girls-p05.csv', 'girls-p10.csv', 'girls-p25.csv', 'girls-p50.csv',
                   'girls-p75.csv', 'girls-p90.csv', 'girls-p95.csv',]

# create dataframe with age column
df_boys = pd.DataFrame({'age': age_list})
df_girls = pd.DataFrame({'age': age_list})
# loop over file names and read in data
for i, file_name in enumerate(file_names_boys):
    # read in data
    df_temp = pd.read_csv('../archive/data/height/' + file_name, names=['age','height'])
    spline = UnivariateSpline(df_temp['age'], df_temp['height'], s=0.5)
    df_boys[col_names[i]] = spline(age_list)
for i, file_name in enumerate(file_names_girls):
    # read in data
    df_temp = pd.read_csv('../archive/data/height/' + file_name, names=['age','height'])
    spline = UnivariateSpline(df_temp['age'], df_temp['height'], s=0.5)
    df_girls[col_names[i]] = spline(age_list)

# make age index
df_boys.set_index('age', inplace=True)
```

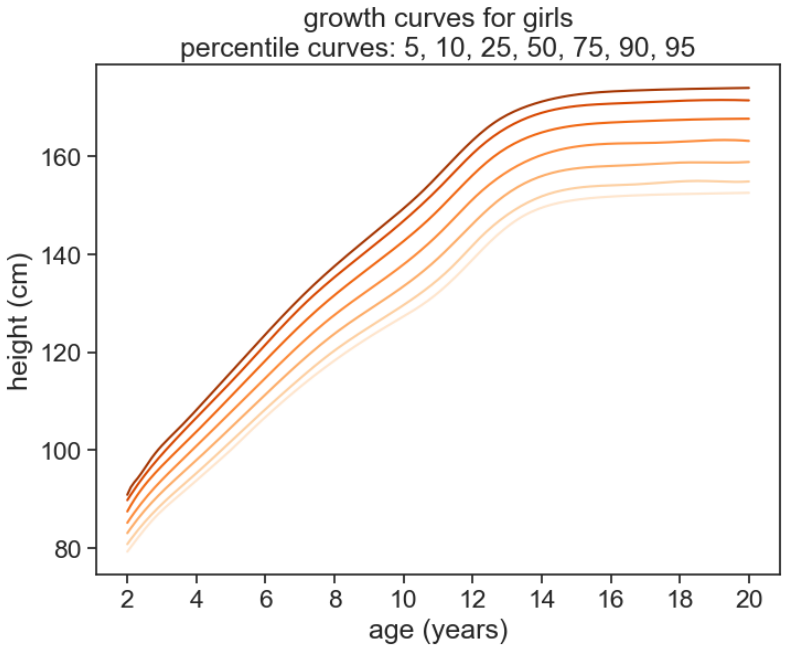
```
df_boys.index = df_boys.index.round(1)
df_boys.to_csv('../archive/data/height/boys_height_vs_age_combined.csv', index=True)
df_girls.set_index('age', inplace=True)
df_girls.index = df_girls.index.round(1)
df_girls.to_csv('../archive/data/height/girls_height_vs_age_combined.csv', index=True)
```

Let's take a look at what we just did.

```
df_girls
```

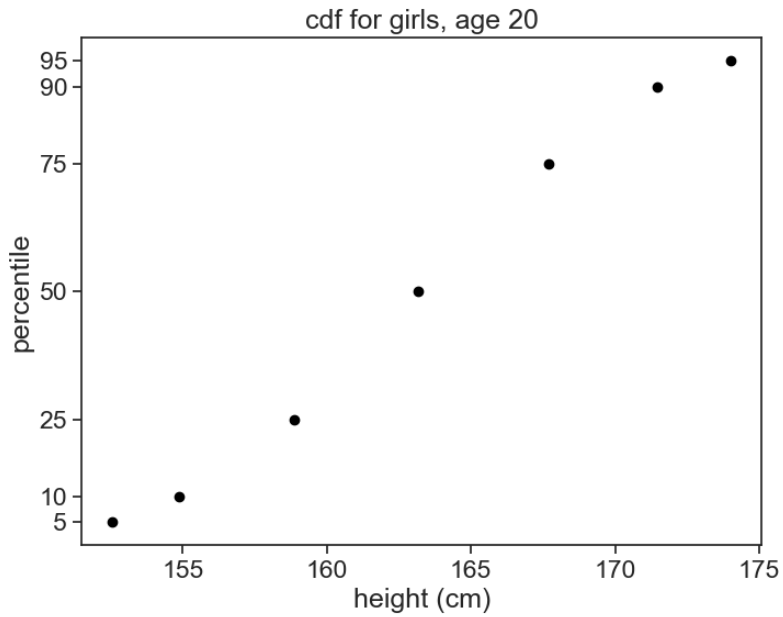
age	p05	p10	p25	p50	p75	p90	p95
2.0	79.269087	80.794167	83.049251	85.155597	87.475854	89.779822	90.882059
2.1	80.202106	81.772053	84.052858	86.207778	88.713405	90.883740	92.409913
2.2	81.130687	82.706754	85.011591	87.211543	89.856186	91.940642	93.416959
2.3	82.048325	83.601023	85.928399	88.170313	90.914093	92.953965	94.270653
2.4	82.948516	84.457612	86.806234	89.087509	91.897022	93.927147	95.226089
...	...	...	...	...	...	...	...
19.6	152.520938	154.812286	158.775277	163.337149	167.699533	171.531349	173.969235
19.7	152.534223	154.814440	158.791925	163.310864	167.704618	171.519600	173.980150
19.8	152.548001	154.827666	158.815071	163.275852	167.708562	171.504730	173.990964
19.9	152.562338	154.853760	158.845506	163.231563	167.711342	171.486629	174.001704
20.0	152.577300	154.894521	158.884019	163.177444	167.712936	171.465189	174.012396

```
fig, ax = plt.subplots(figsize=(8, 6))
# loop over col_names and plot each column
colors = sns.color_palette("Oranges", len(col_names))
for col, color in zip(col_names, colors):
    ax.plot(df_girls.index, df_girls[col], label=col, color=color)
ax.set(xlabel='age (years)',
       ylabel='height (cm)',
       xticks=np.arange(2, 21, 2),
       title="growth curves for girls\npercentile curves: 5, 10, 25, 50, 75, 90, 95",
       );
```



Let's now see the percentiles for girls age 20.

```
fig, ax = plt.subplots(figsize=(8, 6))
percentile_list = np.array([5, 10, 25, 50, 75, 90, 95])
data = df_girls.loc[20.0]
ax.plot(data, percentile_list, ls='', marker='o', markersize=6, color="black")
ax.set(xlabel='height (cm)',
         ylabel='percentile',
         yticks=percentile_list,
         title="cdf for girls, age 20"
         );
```



I suspect that the heights in the population are normally distributed. Let's check that. I'll fit the data to the integral of a gaussian, because the percentiles correspond to a cdf. If a pdf is a gaussian, its cumulative is given by

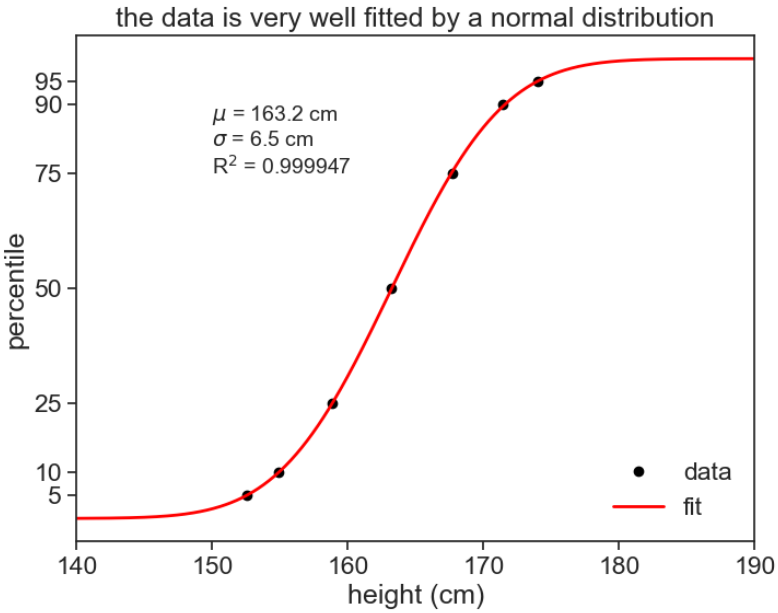
$$\Phi(x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right)$$

where μ is the mean and σ is the standard deviation of the distribution. The error function erf is a sigmoid function, which is a good approximation for the cdf of the normal distribution.

```
def erf_model(x, mu, sigma):
    return 50 * (1 + erf((x - mu) / (sigma * np.sqrt(2))) )
# initial guess for parameters: [mu, sigma]
p0 = [150, 6]
# Calculate R-squared
def calculate_r2(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
    return 1 - (ss_res / ss_tot)
```

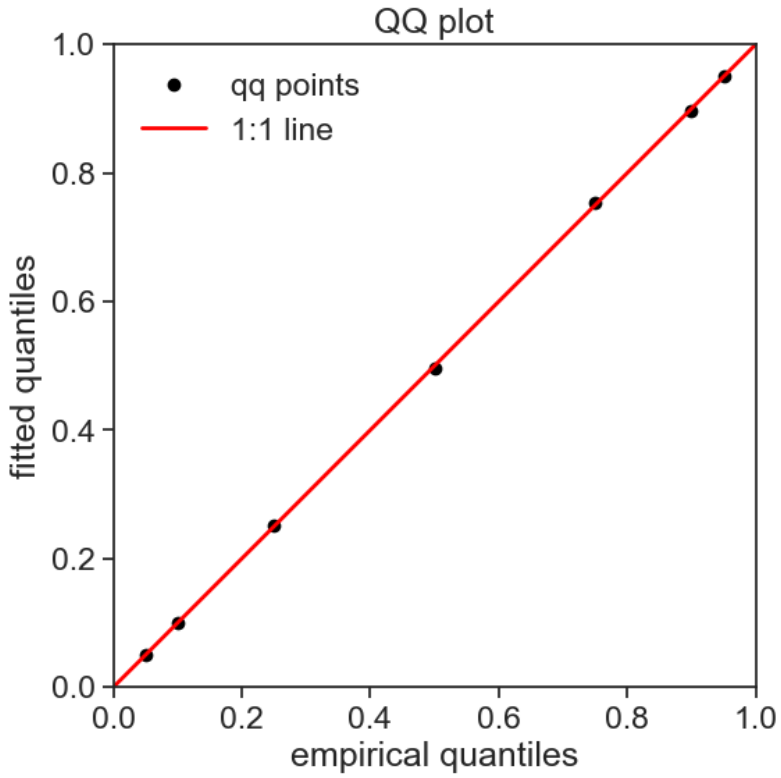
```
data = df_girls.loc[20.0]
params, _ = curve_fit(erf_model, data, percentile_list, p0=p0,
                      bounds=([100, 3],   # lower bounds for mu and sigma
                              [200, 10])  # upper bounds for mu and sigma
                      )
# store the parameters in the dataframe
percentile_predicted = erf_model(data, *params)
# R-squared value
r2 = calculate_r2(percentile_list, percentile_predicted)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
percentile_list = np.array([5, 10, 25, 50, 75, 90, 95])
data = df_girls.loc[20.0]
ax.plot(data, percentile_list, ls='', marker='o', markersize=6, color="black", label='data')
fit = erf_model(height_list, *params)
ax.plot(height_list, fit, label='fit', color="red", linewidth=2)
ax.text(150, 75, f'$\mu$ = {params[0]:.1f} cm\n$\sigma$ = {params[1]:.1f} cm\nR$^2$ = {r2:.6f}
        fontsize=14, bbox=dict(facecolor='white', alpha=0.5))
ax.legend(frameon=False)
ax.set(xlabel='height (cm)',
       xlim=(140, 190),
         ylabel='percentile',
         yticks=percentile_list,
         title="the data is very well fitted by a normal distribution"
         );
```



Another way of making sure that the model fits the data is to make a QQ plot. In this plot, the quantiles of the data are plotted against the quantiles of the normal distribution. If the data is normally distributed, the points should fall on a straight line.

```
fitted_quantiles = norm.cdf(data, loc=params[0], scale=params[1])
experimental_quantiles = percentile_list / 100
fig, ax = plt.subplots(figsize=(8, 6))
ax.set_aspect('equal', adjustable='box')
ax.plot(experimental_quantiles, fitted_quantiles,
        ls='', marker='o', markersize=6, color="black",
        label='qq points')
ax.plot([0, 1], [0, 1], color='red', linewidth=2, label="1:1 line")
ax.set(xlabel='empirical quantiles',
       ylabel='fitted quantiles',
       xlim=(0, 1),
       ylim=(0, 1),
       title="QQ plot")
ax.legend(frameon=False)
```



Great, now we just need to do exactly the same for both sexes, and all the ages. I chose to divide age from 2 to 20 into 0.1 intervals.

```
df_stats_boys = pd.DataFrame(index=age_list, columns=['mu', 'sigma', 'r2'])
df_stats_boys['mu'] = 0.0
df_stats_boys['sigma'] = 0.0
df_stats_boys['r2'] = 0.0
df_stats_girls = pd.DataFrame(index=age_list, columns=['mu', 'sigma', 'r2'])
df_stats_girls['mu'] = 0.0
df_stats_girls['sigma'] = 0.0
df_stats_girls['r2'] = 0.0
```

```
p0 = [80, 3]
# loop over ages in the index, calculate mu and sigma
for i in df_boys.index:
    # fit the model to the data
    data = df_boys.loc[i]
```

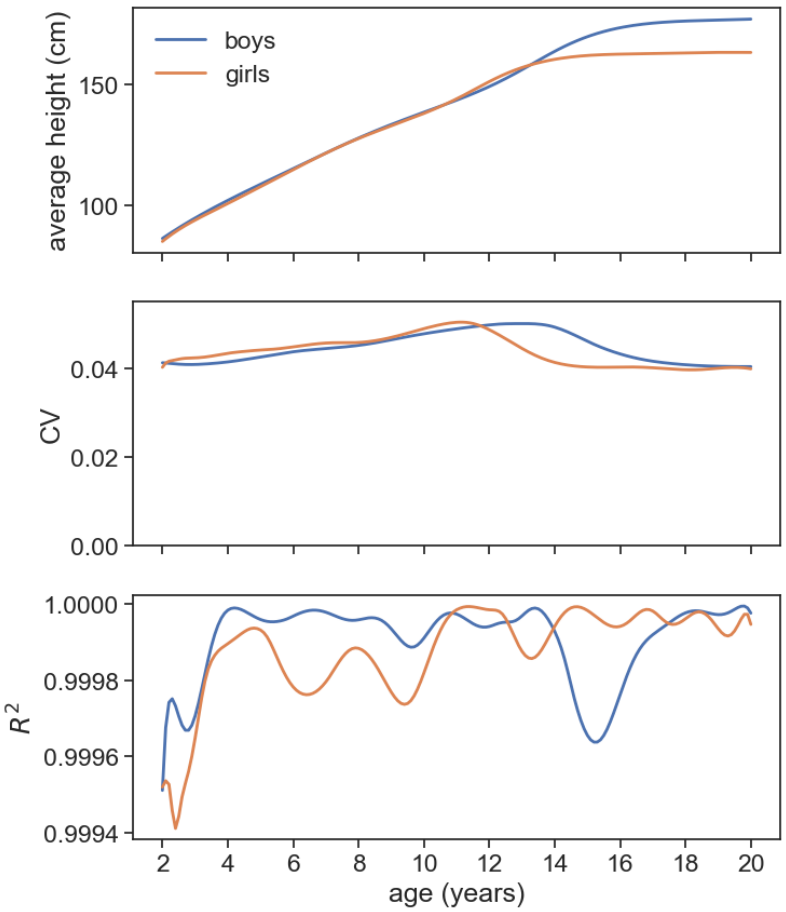
```
    params, _ = curve_fit(erf_model, data, percentile_list, p0=p0,
                          bounds=([70, 2],   # lower bounds for mu and sigma
                                  [200, 10])  # upper bounds for mu and sigma
                         )
    # store the parameters in the dataframe
    df_stats_boys.at[i, 'mu'] = params[0]
    df_stats_boys.at[i, 'sigma'] = params[1]
    percentile_predicted = erf_model(data, *params)
    # R-squared value
    r2 = calculate_r2(percentile_list, percentile_predicted)
    df_stats_boys.at[i, 'r2'] = r2
    p0 = params
# same for girls
p0 = [80, 3]
for i in df_girls.index:
    # fit the model to the data
    data = df_girls.loc[i]
    params, _ = curve_fit(erf_model, data, percentile_list, p0=p0,
                          bounds=([70, 3],   # lower bounds for mu and sigma
                                  [200, 10])  # upper bounds for mu and sigma
                         )
    # store the parameters in the dataframe
    df_stats_girls.at[i, 'mu'] = params[0]
    df_stats_girls.at[i, 'sigma'] = params[1]
    percentile_predicted = erf_model(data, *params)
    # R-squared value
    r2 = calculate_r2(percentile_list, percentile_predicted)
    df_stats_girls.at[i, 'r2'] = r2
    p0 = params

# save the dataframes to csv files
df_stats_boys.to_csv('../archive/data/height/boys_height_stats.csv', index=True)
df_stats_girls.to_csv('../archive/data/height/girls_height_stats.csv', index=True)
```

Let's see what we got. The top panel in the graph shows the average height for boys and girls, the middle panel shows the coefficient of variation (σ/μ), and the bottom panel shows the R2 of the fit (note that the range is very close to 1).

```
df_stats_boys
```

	mu	sigma	r2
2.0	86.463069	3.563785	0.999511
2.1	87.374895	3.596583	0.999676
2.2	88.269676	3.627433	0.999742
2.3	89.148086	3.657263	0.999752
2.4	90.010783	3.686764	0.999733
...	...	...	...
19.6	176.802810	7.134561	0.999991
19.7	176.845789	7.135786	0.999994
19.8	176.892196	7.137430	0.999995
19.9	176.942521	7.139466	0.999990
20.0	176.997255	7.141858	0.999976

```
fig, ax = plt.subplots(3,1, figsize=(8, 10), sharex=True)
fig.subplots_adjust(left=0.15)
ax[0].plot(df_stats_boys['mu'], label='boys', lw=2)
ax[0].plot(df_stats_girls['mu'], label='girls', lw=2)
ax[0].legend(frameon=False)

ax[1].plot(df_stats_boys['sigma'] / df_stats_boys['mu'], lw=2)
ax[1].plot(df_stats_girls['sigma'] / df_stats_girls['mu'], lw=2)

ax[2].plot(df_stats_boys.index, df_stats_boys['r2'], label=r'$r2$ boys', lw=2)
ax[2].plot(df_stats_girls.index, df_stats_girls['r2'], label=r'$r2$ girls', lw=2)

ax[0].set(ylabel='average height (cm)',)
ax[1].set(ylabel='CV',
          ylim=[0,0.055])
ax[2].set(xlabel='age (years)',
            ylabel=r'$R^2$',
            xticks=np.arange(2, 21, 2),
          );
```



Let's see how the pdfs for boys and girls move and morph as age increases.

```
age_list_string = age_list.astype(str).tolist()
df_pdf_boys = pd.DataFrame(index=height_list, columns=age_list_string)
df_pdf_girls = pd.DataFrame(index=height_list, columns=age_list_string)

for age in df_pdf_boys.columns:
    age_float = round(float(age), 1)
    df_pdf_boys[age] = norm.pdf(height_list,
                                loc=df_stats_boys.loc[age_float]['mu'],
                                scale=df_stats_boys.loc[age_float]['sigma'])
for age in df_pdf_girls.columns:
    age_float = round(float(age), 1)
    df_pdf_girls[age] = norm.pdf(height_list,
```

```
                                    loc=df_stats_girls.loc[age_float]['mu'],
                                    scale=df_stats_girls.loc[age_float]['sigma'])
```

```
df_pdf_girls
```

	2.0	2.1	2.2	2.3	2.4	2.5	2.6
70.0	0.000006	2.962419e-06	1.229580e-06	4.740717e-07	1.893495e-07	7.928033e-08	3.395629e-08
70.1	0.000007	3.369929e-06	1.401926e-06	5.423176e-07	2.172465e-07	9.118694e-08	3.914667e-08
70.2	0.000008	3.830459e-06	1.597215e-06	6.199308e-07	2.490751e-07	1.048086e-07	4.509972e-08
70.3	0.000009	4.350475e-06	1.818328e-06	7.081296e-07	2.853621e-07	1.203810e-07	5.192270e-08
70.4	0.000010	4.937172e-06	2.068480e-06	8.082806e-07	3.267014e-07	1.381707e-07	5.973725e-08
...	...	...	...	...	...	...	...
219.5	0.000000	5.214425e-307	1.377605e-289	3.568527e-277	6.457994e-266	2.232144e-255	6.340272e-24
219.6	0.000000	1.813597e-307	5.050074e-290	1.356408e-277	2.537010e-266	9.046507e-256	2.642444e-24
219.7	0.000000	6.302763e-308	1.849870e-290	5.151948e-278	9.959447e-267	3.663840e-256	1.100546e-24
219.8	0.000000	2.188653e-308	6.771033e-291	1.955386e-278	3.906942e-267	1.482823e-256	4.580523e-24
219.9	0.000000	7.594139e-309	2.476504e-291	7.416066e-279	1.531537e-267	5.997065e-257	1.905138e-24

```
import plotly.graph_objects as go
import plotly.io as pio

pio.renderers.default = 'notebook'

# create figure
fig = go.Figure()

# assume both dataframes have the same columns (ages) and index (height)
ages = df_pdf_boys.columns
x_vals = df_pdf_boys.index

# add traces: 2 per age (boys and girls), all hidden except the first pair
for i, age in enumerate(ages):
    fig.add_trace(go.Scatter(x=x_vals, y=df_pdf_boys[age], name=f'Boys {age}',
                             line=dict(color='#1f77b4'), visible=(i == 0)))
    fig.add_trace(go.Scatter(x=x_vals, y=df_pdf_girls[age], name=f'Girls {age}',
                             line=dict(color='#ff7f0e'), visible=(i == 0)))

# create slider steps
```

```
steps = []
for i, age in enumerate(ages):
    vis = [False] * (2 * len(ages))
    vis[2*i] = True      # boys trace
    vis[2*i + 1] = True  # girls trace

    steps.append(dict(
        method='update',
        args=[{'visible': vis},
              {'title': f'Height Distribution - Age: {age}'}],
        label=str(age)
    ))

# define slider
sliders = [dict(
    active=0,
    currentvalue={"prefix": "Age: "},
    pad={"t": 50},
    steps=steps
)]

# update layout
fig.update_layout(
    sliders=sliders,
    title='Height Distribution by Age',
    xaxis_title='Height (cm)',
    yaxis_title='Density',
    yaxis=dict(range=[0, 0.12]),
    showlegend=True,
    height=600,
    width=800
)

fig.show()
```

```
Unable to display output for mime type(s): text/html
```

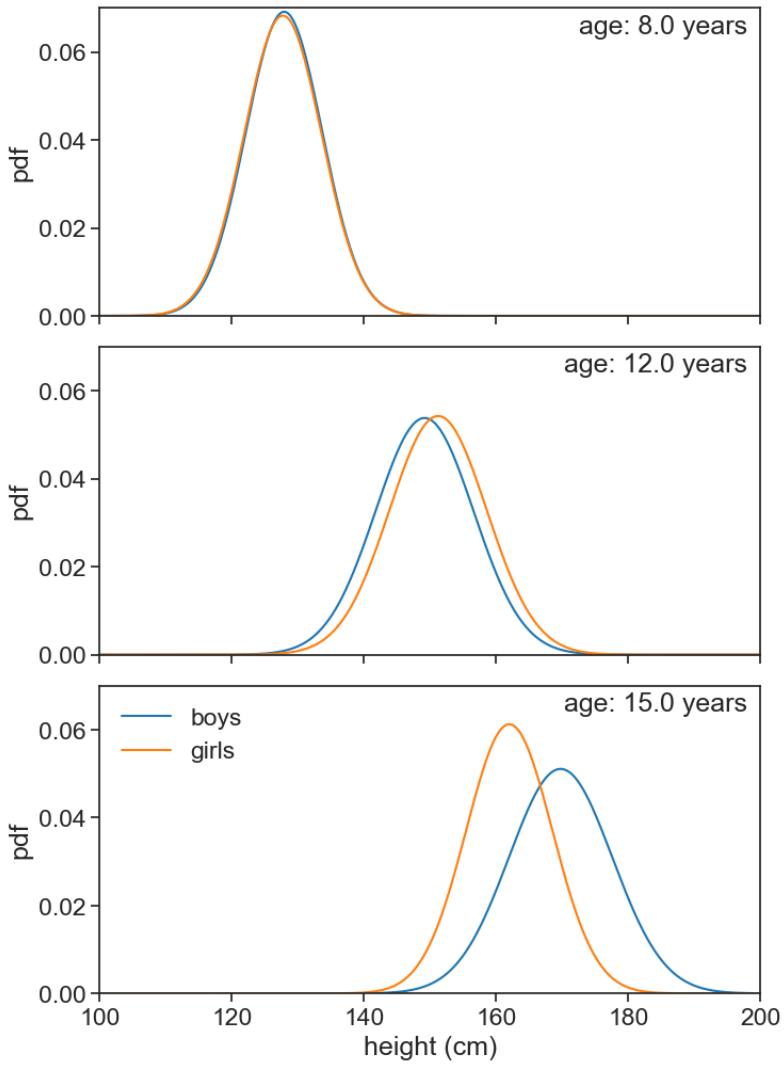
```
Unable to display output for mime type(s): text/html
```

A few notes about what we can learn from the analysis above.

- My impression that 12-year-old girls are taller than boys is indeed true.
- Boys and girls have very similar distributions up to age 11.
- From age 11 to 13 girls are on average taller than boys.
- From age 13 boys become taller than girls, on average.
- The graph showing the coefficient of variation is interesting. CV for girls peaks roughtly at age 12, and for boys it peaks around age 14. These local maxima may be explained by the wide variability in the age ofpuberty onset.
- The height distribution for each sex, across all ages, is indeed extremely well described by the normal distribution. What biological factors may account for such a fact?

I'll plot one last graph from now, let's see what we can learn from it. Let's see the pdf for boys and girls across three age groups: 8, 12, and 15 year olds.

```
fig, ax = plt.subplots(3, 1, figsize=(8, 12), sharex=True)
fig.subplots_adjust(hspace=0.1)
ages_to_plot = [8.0, 12.0, 15.0]

for i, age in enumerate(ages_to_plot):
    pdf_boys = norm.pdf(height_list, loc=df_stats_boys.loc[age]['mu'], scale=df_stats_boys.loc
    pdf_girls = norm.pdf(height_list, loc=df_stats_girls.loc[age]['mu'], scale=df_stats_girls.l
    ax[i].plot(height_list, pdf_boys, label='boys', color='tab:blue')
    ax[i].plot(height_list, pdf_girls, label='girls', color='tab:orange')
    ax[i].text(0.98, 0.98, f'age: {age} years', transform=ax[i].transAxes, verticalalignment='t
    ax[i].set(ylabel='pdf',
              ylim=(0, 0.07),
             )
ax[2].legend(frameon=False)
ax[2].set(xlabel='height (cm)',
          xlim=(100, 200),);
```



- Indeed, boys and girls age 8 have the exact same height distribution.
- 12-year-old girls are indeed taller than boys, on average. This difference is relatiely small, though.
- By age 15 boys have long surpassed girls in height, and the difference is quite large. Boys still have some growing to do, but girls are mostly done growing.

2 weight data

Now that we have height data covered, it's time we deal with weight data.

Yes, I am **VERY WELL AWARE** that weight is a force, and it is not measured in kg. Nevertheless, I will use the word weight in the colloquial sense, and for all purposes it is a synonym for mass.

This analysis is based on the CDC Growth Charts Data Files. From there I downloaded a csv for the weight of boys and girls, from age 2 to 20.

For each sex and age, the csv contains three important columns for us:

- M: median
- S: generalized coefficient of variation
- L: Box-Cox power

These three variables can be combined to give the weight W at a given Z-score (number of standard deviations from the mean):

$$W = M\left(1 + LSZ\right)^{1/L} \tag{1}$$

The website contains a different formula for the case $L = 0$, but in our data set L is never zero.

It will be useful in a little while to know the inverse of Eq. (1), which is

$$Z = \frac{(W/M)^L - 1}{LS}. \tag{1b}$$

The formulas above indicate that we're using the Box-Cox distribution (also called power-normal distribution), and they will help us compute the probability density function (pdf) for weight.

Given that the pdf of a z-scored variable is

$$f_z(Z) = \frac{1}{\sqrt{2\pi}} e^{-Z^2/2}, \tag{2}$$

we need to change variables from Z to W. To do that, we will use

$$f_w(W)dW = f_z(Z)dZ. \tag{3}$$

The rationale behind this is that the probability (area) of being in a small interval is the same, whether we measure it in terms of W or Z. See more here: Function of random variables and change of variables in the probability density function. Solving Eq. (3) for $f_w(W)$, we get

$$f_w(W) = f_z(Z)\left|\frac{dZ}{dW}\right| = f_z(Z)\left|\frac{dW}{dZ}\right|^{-1}. \tag{4}$$

Using Eq. (1), the derivative of W with respect to Z is

$$\begin{aligned}
\frac{dW}{dZ} &= MS\left(1+LSZ\right)^{\frac{1}{L}-1} \\
&= \underbrace{M\left(1+LSZ\right)^{\frac{1}{L}}}_{=W \text{ according to Eq. (1)}} \quad S\left(1+LSZ\right)^{-1} \\
&= WS\frac{1}{1+LSZ} \\
&= WS\frac{1}{1+\cancel{LS}\frac{(W/M)^L-1}{\cancel{LS}}} \\
&= WS\frac{1}{(W/M)^L} \\
&= W^{1-L}M^LS
\end{aligned} \tag{5}$$

Putting everything together [remember that we need the reciprocal of the result in Eq. (5)], we get

$$\begin{aligned} f_w(W) &= f_z(Z)\frac{W^{L-1}}{M^L S} \\ &= \frac{1}{\sqrt{2\pi}} e^{-Z^2/2}\frac{W^{L-1}}{M^L S} \end{aligned} \tag{6}$$

Of course, we could have substituted Z from Eq. (1b) into Eq. (6) to get a formula that depends only on W, it would be too messy. Later on, we will compute $f_w(W)$ numerically, so we will not need to do that.

2.1 example

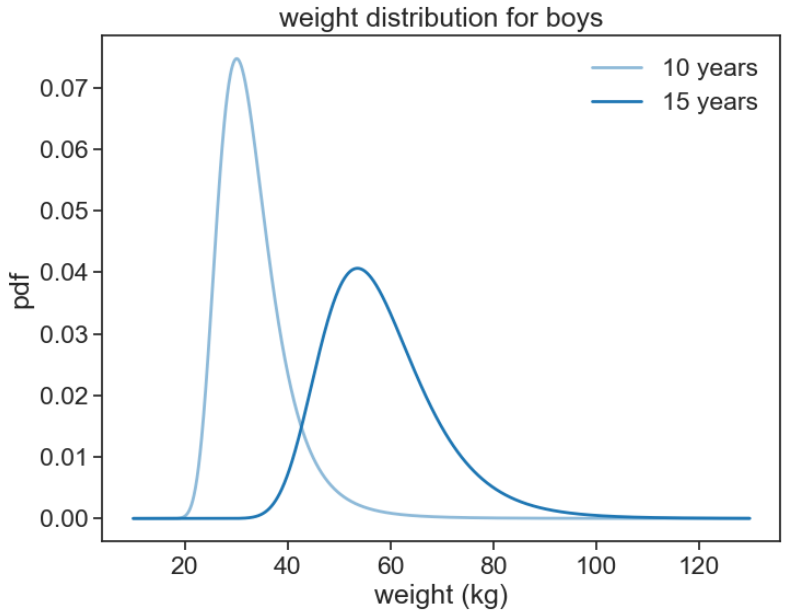
Let's see the weight probability density for boys at age 10 and 15.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import powernorm
```

```
weight_list = np.arange(10, 130, 0.1)
def power_normal_pdf(w, age, sex):
    """
    Calculates the PDF of the Power Normal distribution from the derived formula.
    This function correctly handles negative L values.
    """
    # This function is only valid for w > 0
    w = np.asarray(w)
    pdf = np.full(w.shape, np.nan)
    positive_w = w[w > 0]
    df = pd.read_csv('../archive/data/weight/wtage.csv')
    agemos = age * 12 + 0.5
    df = df[(df['Agemos'] == agemos) & (df['Sex'] == sex)]
```

```
    L = df['L'].values[0]
    M = df['M'].values[0]
    S = df['S'].values[0]

    # Calculate the z-score
    z = ((positive_w / M)**L - 1) / (L * S)

    # Calculate the two main parts of the formula
    pre_factor = positive_w**(L - 1) / (S * M**L * np.sqrt(2 * np.pi))
    exp_term = np.exp(-0.5 * z**2)

    # Store the results only for the valid (positive w) indices
    pdf[w > 0] = pre_factor * exp_term
    return pdf

pdf_boys10 = power_normal_pdf(weight_list, 10, 1)
pdf_boys15 = power_normal_pdf(weight_list, 15, 1)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(weight_list, pdf_boys10, color='tab:blue', lw=2, alpha=0.5, label='10 years')
ax.plot(weight_list, pdf_boys15, color='tab:blue', lw=2, label='15 years')
ax.set(xlabel='weight (kg)',
       ylabel='pdf',
       title="weight distribution for boys")
ax.legend(loc="upper right", frameon=False)
```

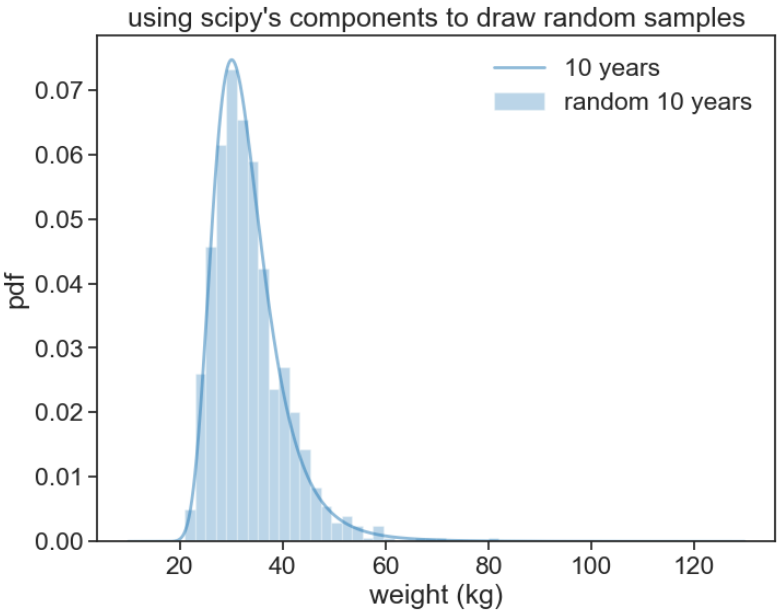


In future chapters, when we want to talk about weight, it will be more convenient to leverage scipy's capabilities, both to compute the pdf and to generate random samples.

```
def scipy_power_normal_pdf(w, age, sex):

    # Load LMS parameters from the CSV file
    df = pd.read_csv('../archive/data/weight/wtage.csv')
    agemos = age * 12 + 0.5
    df = df[(df['Agemos'] == agemos) & (df['Sex'] == sex)]
    L = df['L'].values[0]
    M = df['M'].values[0]
    S = df['S'].values[0]

    # 1. Transform weight (w) to the standard normal z-score
    z = ((w / M)**L - 1) / (L * S)

    # 2. Calculate the derivative of the transformation (dz/dw)
    # This is the Jacobian factor for the change of variables
    dz_dw = (w**(L - 1)) / (S * M**L)

    # 3. Apply the change of variables formula: pdf(w) = pdf(z) * |dz/dw|
    pdf = stats.norm.pdf(z) * dz_dw
```

```
    return pdf

def scipy_power_normal_draw_random(N, age, sex):
    # Load LMS parameters from the CSV file
    df = pd.read_csv('../archive/data/weight/wtage.csv')
    agemos = age * 12 + 0.5
    df = df[(df['Agemos'] == agemos) & (df['Sex'] == sex)]
    L = df['L'].values[0]
    M = df['M'].values[0]
    S = df['S'].values[0]

    # draw random z from standard normal distribution
    z = np.random.normal(0, 1, N)
    # transform z to w
    w = M * (1 + L * S * z)**(1 / L)

    return w

# Calculate the PDFs for 10 and 15-year-old boys using the SciPy-based function
scipy_pdf_boys10 = scipy_power_normal_pdf(weight_list, 10, 1)
draw10 = scipy_power_normal_draw_random(1000, 10, 1)
scipy_pdf_boys15 = scipy_power_normal_pdf(weight_list, 15, 1)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(weight_list, pdf_boys10, color='tab:blue', lw=2, alpha=0.5, label='10 years')
# histogram of draw10
ax.hist(draw10, bins=30, density=True, color='tab:blue', alpha=0.3, label='random 10 years')
# ax.plot(weight_list, scipy_pdf_boys10, color='tab:blue', ls=":", label='scipy 10')
ax.set(xlabel='weight (kg)',
       ylabel='pdf',
       title="using scipy's components to draw random samples")
ax.legend(loc="upper right", frameon=False)
```


using scipy's components to draw random samples
10 years
random 10 years
pdf
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
20
40
60
80
100
120
weight (kg)

Part II

hypothesis testing

3 one-sample t-test

3.1 Question

I measured the height of 10 adult men. Were they sampled from the general population of men?

3.2 Hypotheses

- Null hypothesis: The sample mean is equal to the population mean. In this case, the answer would be "yes"
- Alternative hypothesis: The sample mean is not equal to the population mean. Answer would be "no".
- Significance level: 0.05

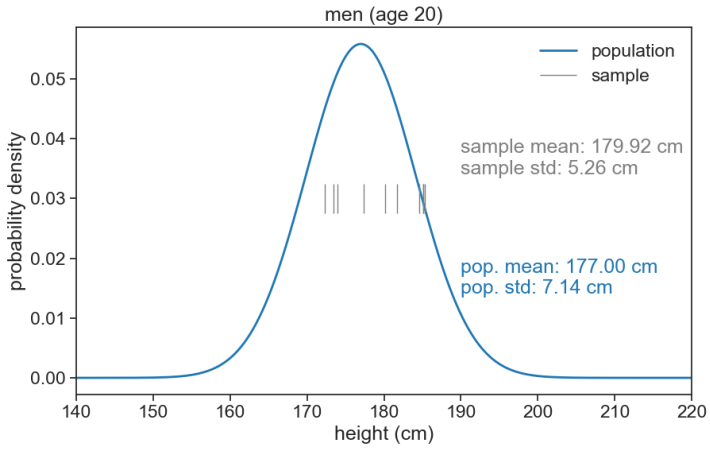
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_1samp, t
%matplotlib widget
```

```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
mu_boys = df_boys.loc[20.0, 'mu']
sigma_boys = df_boys.loc[20.0, 'sigma']
```

Let's start with a sample of 10.

```
N = 10
# set scipy seed for reproducibility
np.random.seed(314)
sample10 = norm.rvs(size=N, loc=mu_boys+2, scale=sigma_boys)
```

```
height_list = np.arange(140, 220, 0.1)
pdf_boys = norm.pdf(height_list, loc=mu_boys, scale=sigma_boys)

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(height_list, pdf_boys, lw=2, color='tab:blue', label='population')

ax.eventplot(sample10, orientation="horizontal", lineoffsets=0.03,
             linewidth=1, linelengths= 0.005,
             colors='gray', label='sample')

ax.text(190, 0.04,
       f"sample mean: {sample10.mean():.2f} cm\nsample std: {sample10.std(ddof=1):.2f} cm",
       ha='left', va='top', color='gray')

ax.text(190, 0.02,
       f"pop. mean: {mu_boys:.2f} cm\npop. std: {sigma_boys:.2f} cm",
       ha='left', va='top', color='tab:blue')

ax.legend(frameon=False)
ax.set(xlabel='height (cm)',
       ylabel='probability density',
       title="men (age 20)",
       xlim=(140, 220),
       );
```



The t value is calculated as follows:

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

where

- $\bar{x}$: sample mean
- μ: population mean
- s: sample standard deviation
- n: sample size

Let's try the formula above and compare it with scipy's ttest_1samp function.

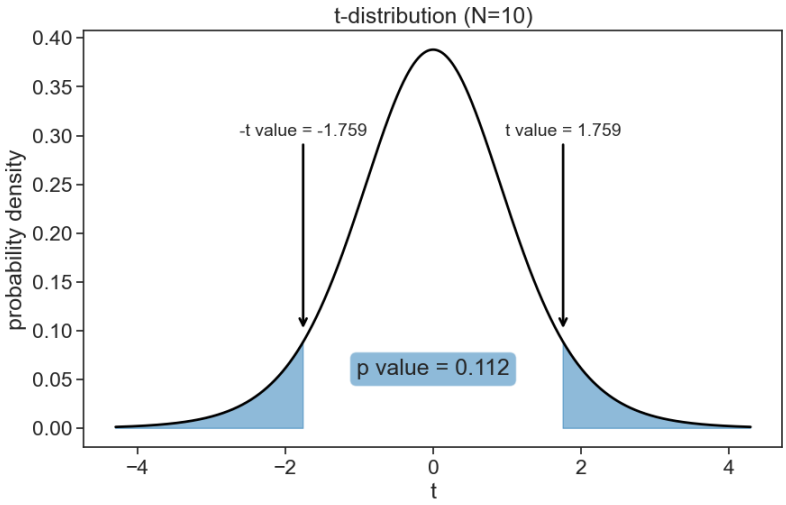
```
t_value_formula = (sample10.mean() - mu_boys) / (sample10.std(ddof=1) / np.sqrt(N))
t_value_scipy = ttest_1samp(sample10, popmean=mu_boys)
print(f"t-value (formula): {t_value_formula:.3f}")
print(f"t-value (scipy): {t_value_scipy.statistic:.3f}")
```

```
t-value (formula): 1.759
t-value (scipy): 1.759
```

Let's convert this t value to a p value. It is easy to visualize the p value by ploting the pdf for the t distribution. The p value is the area under the curve for t greater than the t value and smaller than the negative t value.

```
# degrees of freedom
dof = N - 1
fig, ax = plt.subplots(figsize=(10, 6))

t_array_min = np.round(t.ppf(0.001, dof),3)
t_array_max = np.round(t.ppf(0.999, dof),3)
t_array = np.arange(t_array_min, t_array_max, 0.001)

# annotate vertical array at t_value_scipy
ax.annotate(f"t value = {t_value_scipy.statistic:.3f}",
                        xy=(t_value_scipy.statistic, 0.10),
                        xytext=(t_value_scipy.statistic, 0.30),
                        fontsize=14,
```

```
                        arrowprops=dict(arrowstyle="->", lw=2, color='black'),
                        ha='center')
ax.annotate(f"-t value = -{t_value_scipy.statistic:.3f}",
                        xy=(-t_value_scipy.statistic, 0.10),
                        xytext=(-t_value_scipy.statistic, 0.30),
                        fontsize=14,
                        arrowprops=dict(arrowstyle="->", lw=2, color='black'),
                        ha='center')
# fill between t-distribution and normal distribution
ax.fill_between(t_array, t.pdf(t_array, dof),
                 where=(np.abs(t_array) > t_value_scipy.statistic),
                 color='tab:blue', alpha=0.5,
                 label='rejection region')

# write t_value_scipy.pvalue on the plot
ax.text(0, 0.05,
        f"p value = {t_value_scipy.pvalue:.3f}",
        ha='center', va='bottom',
        bbox=dict(facecolor='tab:blue', alpha=0.5, boxstyle="round"))

ax.plot(t_array, t.pdf(t_array, dof),
       color='black', lw=2)

ax.set(xlabel='t',
       ylabel='probability density',
       title="t-distribution (N=10)",
       );
```



The p value is the fraction of the t distribution that is more extreme than the observed t value. If the p value is less than the significance level, we reject the null hypothesis. In this case, the p value is larger than the significance level, so we fail to reject the null hypothesis. This means that we do not have enough evidence to say that the sample mean is different from the population mean. In other words, we cannot conclude that the 10 men samples were drawn from a distribution different than the general population.

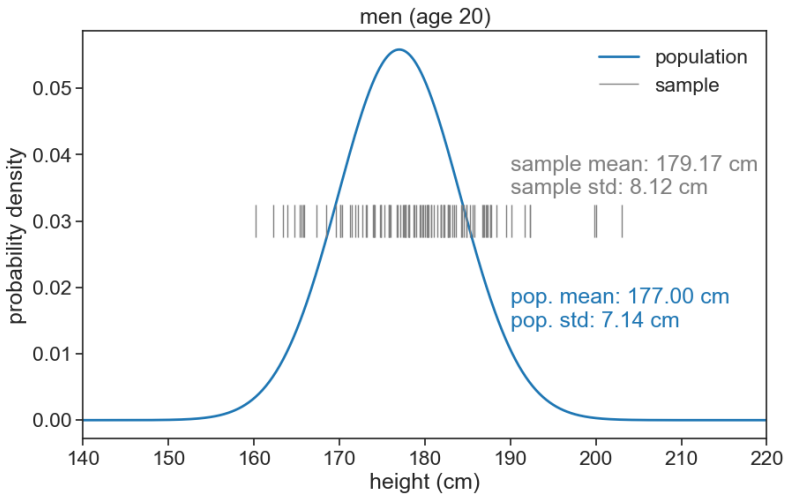
3.3 increase the sample size

Let's see what happens when we increase the sample size to 100.

```
N = 100
# set scipy seed for reproducibility
np.random.seed(628)
sample100 = norm.rvs(size=N, loc=mu_boys+2, scale=sigma_boys)
```

```
height_list = np.arange(140, 220, 0.1)
pdf_boys = norm.pdf(height_list, loc=mu_boys, scale=sigma_boys)

fig, ax = plt.subplots(figsize=(10, 6))
```

```
ax.plot(height_list, pdf_boys, lw=2, color='tab:blue', label='population')

ax.eventplot(sample100, orientation="horizontal", lineoffsets=0.03,
             linewidth=1, linelengths= 0.005,
             colors='gray', label='sample')

ax.text(190, 0.04,
       f"sample mean: {sample100.mean():.2f} cm\nsample std: {sample100.std(ddof=1):.2f} cm",
       ha='left', va='top', color='gray')

ax.text(190, 0.02,
       f"pop. mean: {mu_boys:.2f} cm\npop. std: {sigma_boys:.2f} cm",
       ha='left', va='top', color='tab:blue')

ax.legend(frameon=False)
ax.set(xlabel='height (cm)',
       ylabel='probability density',
       title="men (age 20)",
       xlim=(140, 220),
       );
```

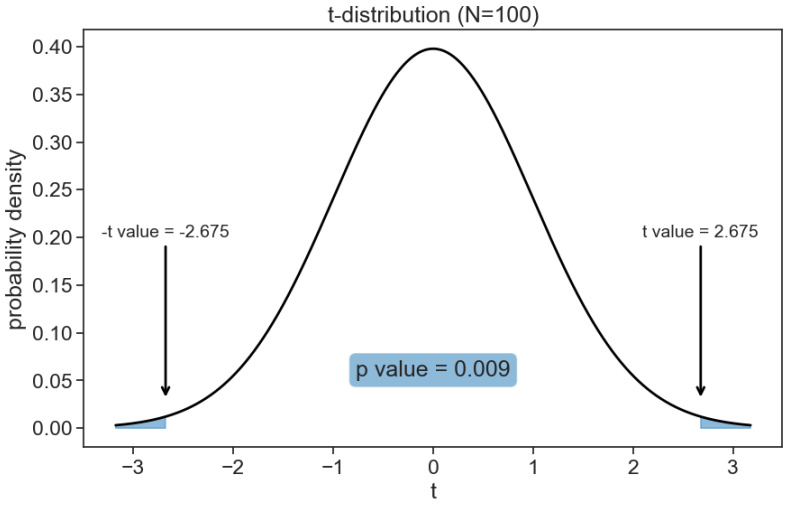



```
t_value_scipy = ttest_1samp(sample100, popmean=mu_boys)
print(f"t-value: {t_value_scipy.statistic:.3f}")
print(f"p-value: {t_value_scipy.pvalue:.3f}")
```

```
t-value: 2.675
p-value: 0.009
```

```
# degrees of freedom
dof = N - 1
fig, ax = plt.subplots(figsize=(10, 6))

t_array_min = np.round(t.ppf(0.001, dof),3)
t_array_max = np.round(t.ppf(0.999, dof),3)
t_array = np.arange(t_array_min, t_array_max, 0.001)

# annotate vertical array at t_value_scipy
ax.annotate(f"t value = {t_value_scipy.statistic:.3f}",
                        xy=(t_value_scipy.statistic, 0.03),
                        xytext=(t_value_scipy.statistic, 0.20),
                        fontsize=14,
                        arrowprops=dict(arrowstyle="->", lw=2, color='black'),
                        ha='center')
ax.annotate(f"-t value = -{t_value_scipy.statistic:.3f}",
                        xy=(-t_value_scipy.statistic, 0.03),
                        xytext=(-t_value_scipy.statistic, 0.20),
                        fontsize=14,
                        arrowprops=dict(arrowstyle="->", lw=2, color='black'),
                        ha='center')
# fill between t-distribution and normal distribution
ax.fill_between(t_array, t.pdf(t_array, dof),
                 where=(np.abs(t_array) > t_value_scipy.statistic),
                 color='tab:blue', alpha=0.5,
                 label='rejection region')

# write t_value_scipy.pvalue on the plot
ax.text(0, 0.05,
        f"p value = {t_value_scipy.pvalue:.3f}",
        ha='center', va='bottom',
        bbox=dict(facecolor='tab:blue', alpha=0.5, boxstyle="round"))

ax.plot(t_array, t.pdf(t_array, dof),
       color='black', lw=2)

ax.set(xlabel='t',
       ylabel='probability density',
```

```
        title="t-distribution (N=100)",
        );
```



3.4 Question 2

Can we say that the sampled men are taller than the general population?

3.5 Hypotheses

- Null hypothesis: The sample mean is equal to the population mean.
- Alternative hypothesis: The sample mean is higher the population mean.
- Significance level: 0.05

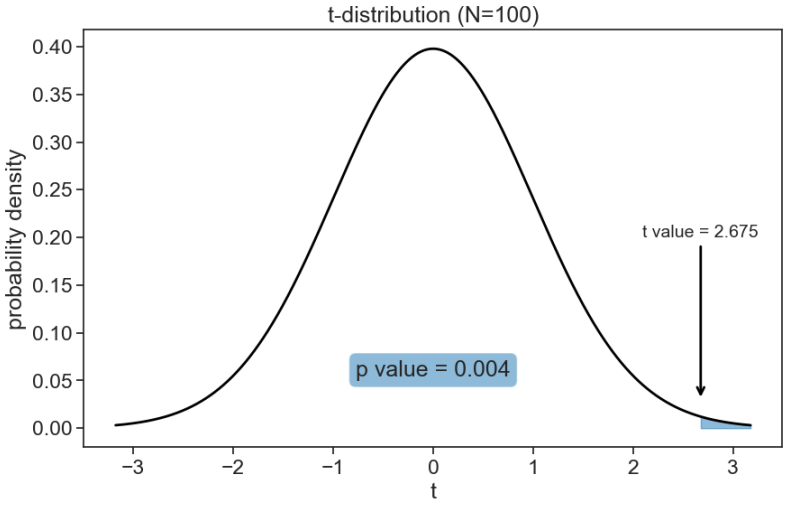
The analysis is the same as before, but we will use a one-tailed test. The t statistic is the same, but the p value is smaller, since we account for a smaller portion of the total area of the pdf.

```
t_value_scipy = ttest_1samp(sample100, popmean=mu_boys, alternative='greater')
print(f"t-value: {t_value_scipy.statistic:.3f}")
print(f"p-value: {t_value_scipy.pvalue:.3f}")
```

```
t-value: 2.675
p-value: 0.004
```

```
# degrees of freedom
dof = N - 1
fig, ax = plt.subplots(figsize=(10, 6))

t_array_min = np.round(t.ppf(0.001, dof),3)
t_array_max = np.round(t.ppf(0.999, dof),3)
t_array = np.arange(t_array_min, t_array_max, 0.001)

# annotate vertical array at t_value_scipy
ax.annotate(f"t value = {t_value_scipy.statistic:.3f}",
                        xy=(t_value_scipy.statistic, 0.03),
                        xytext=(t_value_scipy.statistic, 0.20),
                        fontsize=14,
                        arrowprops=dict(arrowstyle="->", lw=2, color='black'),
                        ha='center')
# fill between t-distribution and normal distribution
ax.fill_between(t_array, t.pdf(t_array, dof),
                 where=(t_array > t_value_scipy.statistic),
                 color='tab:blue', alpha=0.5,
                 label='rejection region')

# write t_value_scipy.pvalue on the plot
ax.text(0, 0.05,
        f"p value = {t_value_scipy.pvalue:.3f}",
        ha='center', va='bottom',
        bbox=dict(facecolor='tab:blue', alpha=0.5, boxstyle="round"))

ax.plot(t_array, t.pdf(t_array, dof),
       color='black', lw=2)

ax.set(xlabel='t',
       ylabel='probability density',
```

```
        title="t-distribution (N=100)",
        );
```



The answer is yes: the sampled men are significantly taller than the general population, since the p value is smaller than the significance level.

4 independent samples t-test

4.1 Question

Are 12-year old girls significantly taller than 12-year old boys?

4.2 Hypotheses

- Null hypothesis: Girls and boys have the same mean height.
- Alternative hypothesis: Girls are *significantly* taller.
- Significance level: 0.05

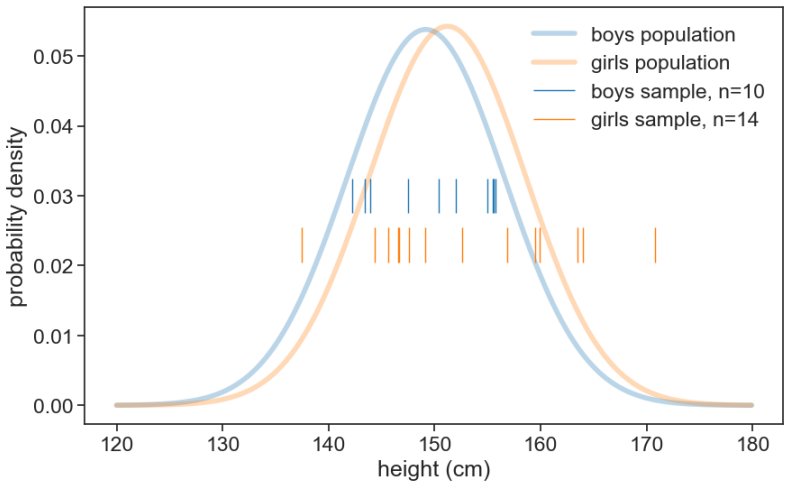
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
# %matplotlib widget
```

```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
df_girls = pd.read_csv('../archive/data/height/girls_height_stats.csv', index_col=0)
age = 12.0
mu_boys = df_boys.loc[age, 'mu']
mu_girls = df_girls.loc[age, 'mu']
sigma_boys = df_boys.loc[age, 'sigma']
sigma_girls = df_girls.loc[age, 'sigma']
```

In this example, we sampled 10 boys and 14 girls. See below the samples data and their underlying distributions.

```
N_boys = 10
N_girls = 14
# set scipy seed for reproducibility
np.random.seed(314)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
```

```
height_list = np.arange(120, 180, 0.1)
pdf_boys = norm.pdf(height_list, loc=mu_boys, scale=sigma_boys)
pdf_girls = norm.pdf(height_list, loc=mu_girls, scale=sigma_girls)
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(height_list, pdf_boys, lw=4, alpha=0.3, color='tab:blue', label='boys population')
ax.plot(height_list, pdf_girls, lw=4, alpha=0.3, color='tab:orange', label='girls population')

ax.eventplot(sample_boys, orientation="horizontal", lineoffsets=0.03,
             linewidth=1, linelengths= 0.005,
             colors='tab:blue', label=f'boys sample, n={N_boys}')
ax.eventplot(sample_girls, orientation="horizontal", lineoffsets=0.023,
             linewidth=1, linelengths= 0.005,
             colors='tab:orange', label=f'girls sample, n={N_girls}')
ax.legend(frameon=False)
ax.set(xlabel='height (cm)',
       ylabel='probability density',
    )
```



To answer the question, we will use an independent samples

t-test.

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\Theta} \tag{4.1}$$

$$\Theta = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \tag{4.2}$$

This is a generalization of the one-sample t-test. If we take one of the samples to be infinite, we get the one-sample t-test.

We can compute the t-statistic by ourselves, and compare the results with those of `scipy.stats.ttest_ind`. Because we are interested in the difference between the means, we will use the `equal_var=False` option to compute Welch's t-test. Also, because we are testing the alternative hypothesis that girls are taller, we will use the one sided test.

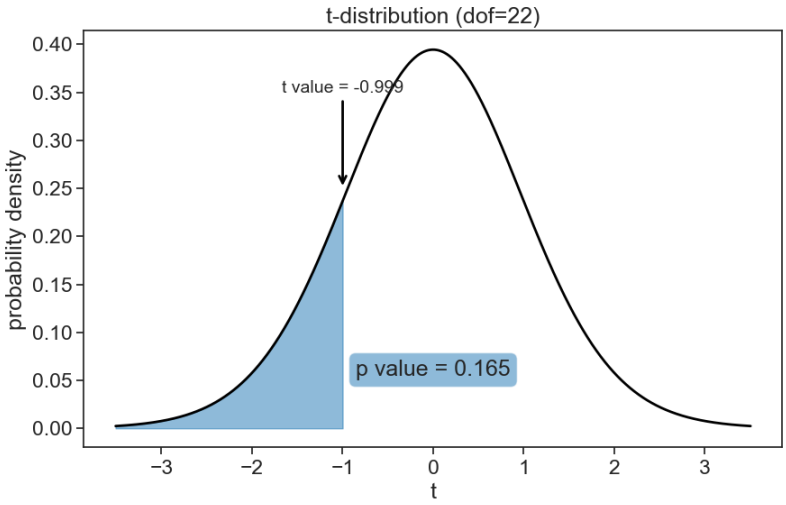
```
Theta = np.sqrt(sample_boys.std(ddof=1)**2/sample_boys.size + \
                sample_girls.std(ddof=1)**2/sample_girls.size)
t_stat = (sample_boys.mean() - sample_girls.mean()) / Theta
dof = N_boys + N_girls - 2
p_val = t.cdf(t_stat, dof)

# the option alternative="less" is used because we are testing whether the first sample (boys)
t_value_scipy = ttest_ind(sample_boys, sample_girls, equal_var=False, alternative="less")

print(f"t-statistic: {t_stat:.3f}, p-value: {p_val:.3f}")
print(f"t-statistic (scipy): {t_value_scipy.statistic:.3f}, p-value (scipy): {t_value_scipy.pva
```

```
t-statistic: -0.999, p-value: 0.164
t-statistic (scipy): -0.999, p-value (scipy): 0.165
```

We got the exact same results :)

Now let's visualize what the p-value means.

```
# degrees of freedom
fig, ax = plt.subplots(figsize=(10, 6))

t_array_min = np.round(t.ppf(0.001, dof),3)
t_array_max = np.round(t.ppf(0.999, dof),3)
t_array = np.arange(t_array_min, t_array_max, 0.001)

# annotate vertical array at t_value_scipy
ax.annotate(f"t value = {t_value_scipy.statistic:.3f}",
                        xy=(t_value_scipy.statistic, 0.25),
                        xytext=(t_value_scipy.statistic, 0.35),
                        fontsize=14,
                        arrowprops=dict(arrowstyle="->", lw=2, color='black'),
                        ha='center')
# fill between t-distribution and normal distribution
ax.fill_between(t_array, t.pdf(t_array, dof),
                 where=(t_array < t_value_scipy.statistic),
                 color='tab:blue', alpha=0.5,
                 label='rejection region')

# write t_value_scipy.pvalue on the plot
ax.text(0, 0.05,
        f"p value = {t_value_scipy.pvalue:.3f}",
        ha='center', va='bottom',
        bbox=dict(facecolor='tab:blue', alpha=0.5, boxstyle="round"))

ax.plot(t_array, t.pdf(t_array, dof),
       color='black', lw=2)

ax.set(xlabel='t',
       ylabel='probability density',
       title="t-distribution (dof=22)",
       );
```



Because the p-value is higher than the significance level, we fail to reject the null hypothesis. This means that, based on the data, we cannot conclude that girls are significantly taller than boys.

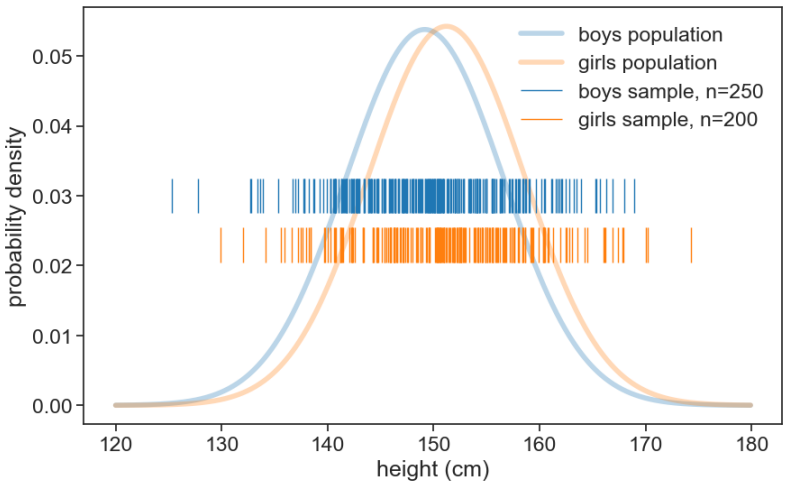
4.3 increasing sample size

Let's increase the sample size to see how it affects the p-value. We'll sample 250 boys and 200 girls now.

```
N_boys = 250
N_girls = 200
# set scipy seed for reproducibility
np.random.seed(314)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
```

```
height_list = np.arange(120, 180, 0.1)
pdf_boys = norm.pdf(height_list, loc=mu_boys, scale=sigma_boys)
pdf_girls = norm.pdf(height_list, loc=mu_girls, scale=sigma_girls)
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(height_list, pdf_boys, lw=4, alpha=0.3, color='tab:blue', label='boys population')
ax.plot(height_list, pdf_girls, lw=4, alpha=0.3, color='tab:orange', label='girls population')
```

```
ax.eventplot(sample_boys, orientation="horizontal", lineoffsets=0.03,
             linewidth=1, linelengths= 0.005,
             colors='tab:blue', label=f'boys sample, n={N_boys}')
ax.eventplot(sample_girls, orientation="horizontal", lineoffsets=0.023,
             linewidth=1, linelengths= 0.005,
             colors='tab:orange', label=f'girls sample, n={N_girls}')
ax.legend(frameon=False)
ax.set(xlabel='height (cm)',
       ylabel='probability density',
    )
```




```
Theta = np.sqrt(sample_boys.std(ddof=1)**2/sample_boys.size + \
                sample_girls.std(ddof=1)**2/sample_girls.size)
t_stat = (sample_boys.mean() - sample_girls.mean()) / Theta
dof = N_boys + N_girls - 2
p_val = t.cdf(t_stat, dof)

# the option alternative="less" is used because we are testing whether the first sample (boys)
t_value_scipy = ttest_ind(sample_boys, sample_girls, equal_var=False, alternative="less")

print(f"t-statistic: {t_stat:.3f}, p-value: {p_val:.3f}")
print(f"t-statistic (scipy): {t_value_scipy.statistic:.3f}, p-value (scipy): {t_value_scipy.pva
```

```
t-statistic: -2.639, p-value: 0.004
t-statistic (scipy): -2.639, p-value (scipy): 0.004
```

We found now a p-value lower than the significance level, so we reject the null hypothesis. This means that, based on the data, we can conclude that girls are significantly taller than boys.

Part III

confidence interval

5 basic concepts

Suppose we randomly select 30 seven-year-old boys from schools around the country and measure their heights (this is our sample). We'd like to use their average height to estimate the true average height of all seven-year-old boys nationwide (the population). Because different samples of 30 boys would yield slightly different averages, we need a way to quantify that uncertainty. A confidence interval gives us a range—based on our sample data—that expresses what we would expect to find if we were to repeat this sampling process many times.

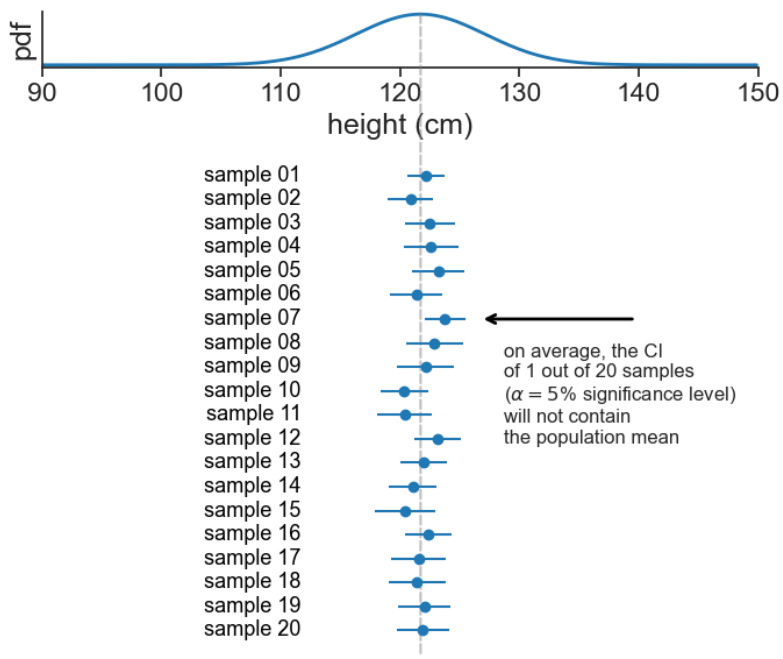
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
import scipy
from matplotlib.lines import Line2D
import matplotlib.gridspec as gridspec
# %matplotlib widget
```

```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
age = 7.0
mu_boys = df_boys.loc[age, 'mu']
sigma_boys = df_boys.loc[age, 'sigma']
```

See the height distribution for seven-year-old boys. Below it we see the means for 20 samples of groups of 30 boys. The 95% confidence interval is the range of values that, on average, 95% of the samples CI contain the true population mean. In this case, this amounts to one out of the 20 samples.

```
np.random.seed(628)
height_list = np.arange(90, 150, 0.1)
pdf_boys = norm.pdf(height_list, loc=mu_boys, scale=sigma_boys)

fig = plt.figure(figsize=(8, 6))
gs = gridspec.GridSpec(2, 1, height_ratios=[0.1, 0.9])
gs.update(left=0.09, right=0.86,top=0.98, bottom=0.06, hspace=0.30, wspace=0.05)
ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[1, 0])

ax0.plot(height_list, pdf_boys, lw=2, color='tab:blue', label='population')

N_samples = 20
N = 30

for i in range(N_samples):
    sample = norm.rvs(loc=mu_boys, scale=sigma_boys, size=N)
    sample_mean = sample.mean()
    # confidence interval
    alpha = 0.05
    z_crit = scipy.stats.t.isf(alpha/2, N-1)
    CI = z_crit * sample.std(ddof=1) / np.sqrt(N)
    ax1.errorbar(sample_mean, i, xerr=CI, fmt='o', color='tab:blue',
                 label=f'sample {i+1}' if i == 0 else "", capsize=0)


from matplotlib.patches import ConnectionPatch
line = ConnectionPatch(xyA=(mu_boys, pdf_boys.max()), xyB=(mu_boys, -1), coordsA="data", coords
                       axesA=ax0, axesB=ax1, color="gray", linestyle='--', linewidth=1.5, alpha=
ax1.add_artist(line)

ax1.annotate(
        '',
        xy=(mu_boys + 5, 13),  # tip of the arrow (first error bar, y=0)
        xytext=(mu_boys + 5 + 13, 13),  # text location
        arrowprops=dict(arrowstyle='->', lw=2, color='black'),
        fontsize=13,
        color='tab:blue',
        ha='left',
        va='center'
```

```
)

ax1.text(mu_boys + 5 + 2, 12, "on average, the CI\nof 1 out of 20 samples\n"
         r"($\alpha=5$% significance level)"
          "\nwill not contain\nthe population mean",
          va="top", fontsize=12)

# write "sample i" for each error bar
for i in range(N_samples):
    ax1.text(mu_boys -10, i, f'sample {N_samples-i:02d}',
             fontsize=13, color='black',
             ha='right', va='center')

# ax.legend(frameon=False)
ax0.spines['top'].set_visible(False)
ax0.spines['right'].set_visible(False)
ax1.spines['top'].set_visible(False)
ax1.spines['right'].set_visible(False)
ax1.spines['left'].set_visible(False)
ax1.spines['bottom'].set_visible(False)

ax0.set(xticks=np.arange(90, 151, 10),
        xlim=(90, 150),
        xlabel='height (cm)',
        # xticklabels=[],
        yticks=[],
        ylabel='pdf',
        )
ax1.set(xticks=[],
        xlim=(90, 150),
        ylim=(-1, N_samples),
        yticks=[],
       );
```


pdf
90
100
110
120
130
140
150
height (cm)
sample 01
sample 02
sample 03
sample 04
sample 05
sample 06
sample 07
sample 08
sample 09
sample 10
sample 11
sample 12
sample 13
sample 14
sample 15
sample 16
sample 17
sample 18
sample 19
sample 20
on average, the CI
of 1 out of 20 samples
($\alpha = 5\%$ significance level)
will not contain
the population mean

6 analytical confidence interval

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
import scipy
# %matplotlib widget
```

We wish to compute the confidence interval for the mean height of 7-year-old boys, for a sample of size N.

We will start our journey with a refresher of the Central Limit Theorem (CLT).

6.1 CLT

The Central Limit Theorem states that the sampling distribution of the sample mean

$$\bar{X} = \frac{1}{N} \sum_{i=1}^{N} X_i$$

approaches a normal distribution as the sample size N increases, regardless of the shape of the population distribution. This normal distribution can be expressed as:

$$\bar{X} \sim N\left(\mu, \frac{\sigma^2}{N}\right),$$

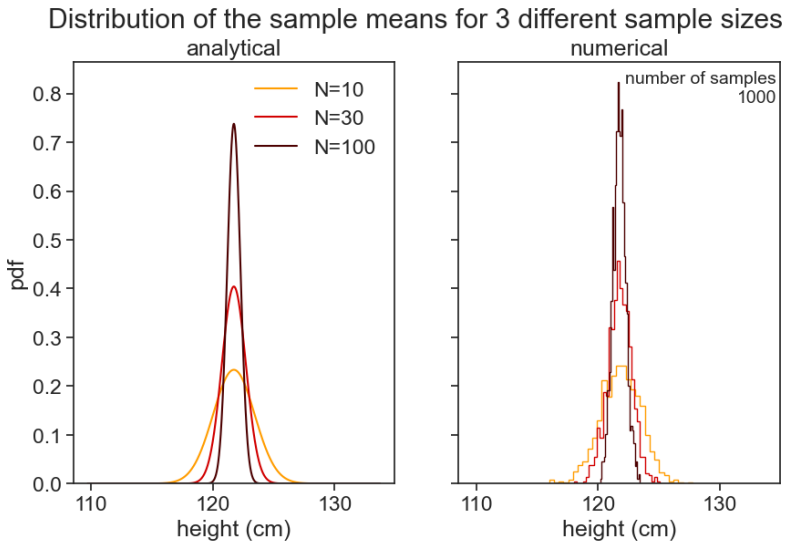
where μ and σ^2 are the population mean and variance, respectively. When talking about samples, we use $\bar{x}$ and s^2 to denote the sample mean and variance.

Let's visualize this. The graph below shows how the sample size N affects the sampling distribution of the sample mean $\bar{X}$. The higher the sample size, the more concentrated the distribution becomes around the population mean μ. If we take N to be infinity, the sampling distribution of the sample mean becomes a delta function at μ, and we will know the exact value of the population mean.

```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
mu_boys = df_boys.loc[7.0, 'mu']
sigma_boys = df_boys.loc[7.0, 'sigma']
```

```
fig, ax = plt.subplots(1,2, figsize=(10, 6), sharex=True, sharey=True)

height_list = np.arange(mu_boys-12, mu_boys+12, 0.01)
N_list = [10, 30, 100]
alpha_list = [0.4, 0.6, 1.0]

colors = plt.cm.hot([0.6, 0.3, 0.1])

N_samples = 1000
np.random.seed(628)
mean_list_10 = []
mean_list_30 = []
mean_list_100 = []
for i in range(N_samples):
    mean_list_10.append(np.mean(norm.rvs(size=10, loc=mu_boys, scale=sigma_boys)))
    mean_list_30.append(np.mean(norm.rvs(size=30, loc=mu_boys, scale=sigma_boys)))
    mean_list_100.append(np.mean(norm.rvs(size=100, loc=mu_boys, scale=sigma_boys)))

alpha = 0.05

# z_alpha_over_two = norm(loc=mu_boys, scale=SE).ppf(1 - alpha / 2)
# z_alpha_over_two = np.round(z_alpha_over_two, 2)

for i,N in enumerate(N_list):
    SE = sigma_boys / np.sqrt(N)
```

```
    ax[0].plot(height_list, norm(loc=mu_boys, scale=SE).pdf(height_list),
            color=colors[i], label=f"N={N}")

ax[1].hist(mean_list_10, bins=30, density=True, color=colors[0], label="N=10", align='mid', his
ax[1].hist(mean_list_30, bins=30, density=True, color=colors[1], label="N=10", align='mid', his
ax[1].hist(mean_list_100, bins=30, density=True, color=colors[2], label="N=10", align='mid', hi

ax[1].text(0.99, 0.98, "number of samples\n1000", ha='right', va='top', transform=ax[1].transAx

ax[0].legend(frameon=False)
ax[0].set(xlabel="height (cm)",
       ylabel="pdf",
       title="analytical"
       )
ax[1].set(xlabel="height (cm)",
          title="numerical"
          )
# title that hovers over both subplots
fig.suptitle(f"Distribution of the sample means for 3 different sample sizes");
```

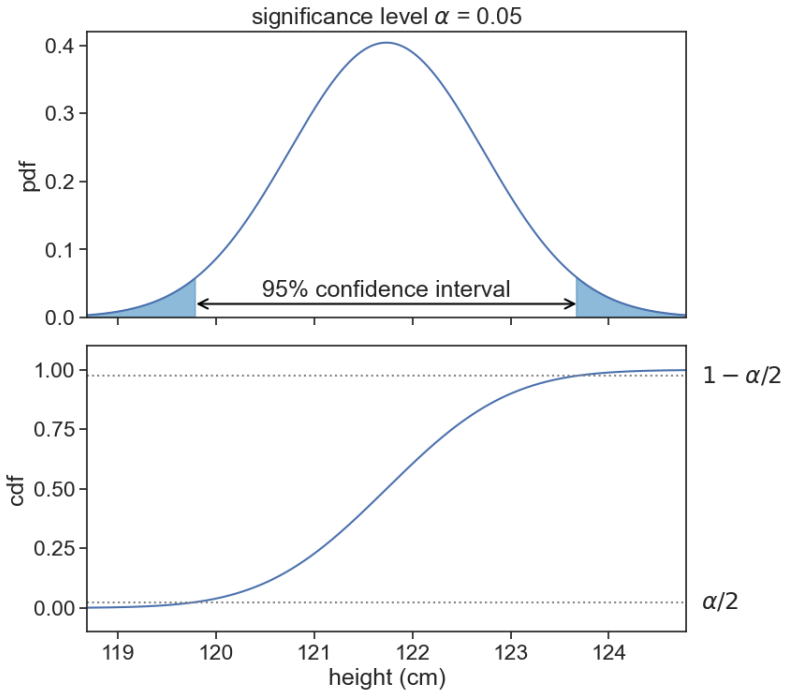


6.2 confidence interval 1

Let's use now the sample size $N = 30$. The confidence interval for a significance level $\alpha = 0.05$ is the interval that leaves $\alpha/2$ of the pdf area in each tail of the distribution.

```
fig, ax = plt.subplots(2, 1, figsize=(8, 8), sharex=True)
plt.subplots_adjust(left=0.1, bottom=0.1, right=0.9, top=0.9, wspace=0.0, hspace=0.1)
N = 30
SE = sigma_boys / np.sqrt(N)

h_min = np.round(norm(loc=mu_boys, scale=SE).ppf(0.001), 2)
h_max = np.round(norm(loc=mu_boys, scale=SE).ppf(0.999), 2)
height_list = np.arange(h_min, h_max, 0.01)

alpha = 0.05
z_alpha_over_two_hi = np.round(norm(loc=mu_boys, scale=SE).ppf(1 - alpha / 2), 2)
z_alpha_over_two_lo = np.round(norm(loc=mu_boys, scale=SE).ppf(alpha / 2), 2)


ax[0].plot(height_list, norm(loc=mu_boys, scale=SE).pdf(height_list))
ax[1].plot(height_list, norm(loc=mu_boys, scale=SE).cdf(height_list))

ax[0].fill_between(height_list, norm(loc=mu_boys, scale=SE).pdf(height_list),
                   where=((height_list > z_alpha_over_two_hi) | (height_list < z_alpha_over_two
                   color='tab:blue', alpha=0.5,
                   label='rejection region')

ax[0].annotate(f"",
               xy=(z_alpha_over_two_hi, 0.02),
               xytext=(z_alpha_over_two_lo, 0.02),
               arrowprops=dict(arrowstyle="<->", lw=1.5, color='black', shrinkA=0.0, shrinkB=0
               )
ax[1].text(h_max+0.15, norm(loc=mu_boys, scale=SE).cdf(z_alpha_over_two_lo), r"$\alpha/2$",
           ha="left", va="center")
ax[1].text(h_max+0.15, norm(loc=mu_boys, scale=SE).cdf(z_alpha_over_two_hi), r"$1-\alpha/2$",
           ha="left", va="center")
ax[1].axhline(alpha/2, color='gray', linestyle=':')
ax[1].axhline(1-alpha/2, color='gray', linestyle=':')
ax[0].text(mu_boys, 0.03, "95% confidence interval", ha="center")
```

```
ax[0].set(ylim=(0, 0.42),
          ylabel="pdf",
          title=r"significance level $\alpha$ = 0.05",
          )
ax[1].set(ylim=(-0.1, 1.1),
          xlim=(h_min, h_max),
          ylabel="cdf",
          xlabel="height (cm)",
          );
```



That's it. That's the whole story.

6.3 confidence interval 2

The rest is repackaging the above in a slightly different way. Instead of finding the top and bottom of the confidence interval according to the cdf of a normal distribution of mean μ and variance σ^2/N, we first standardize this distribution to a

standard normal distribution $Z \sim N(0,1)$, compute the confidence interval for Z, and then transform it back to the original distribution.

If the distribution of the sample mean $\bar{X}$

$$\bar{X} \sim N\left(\mu, \frac{\sigma^2}{N}\right),$$

then the standardized variable Z is defined as:

$$Z = \frac{\bar{x} - \mu}{\sigma/\sqrt{N}} \sim N(0,1).$$

Why is this useful? Because we usually use the same significance level α for all confidence intervals, and we can compute the confidence interval for Z once and use it for all confidence intervals. For $Z \sim N(0,1)$ and $\alpha = 0.05$, the top and bottom of the confidence interval are $Z_{\alpha/2} = \pm 1.96$. Now we only have to invert the expression above to get the confidence interval for $\bar{X}$:

$$X_{1,2} = \mu \pm Z_{\alpha/2} \cdot \frac{\sigma}{\sqrt{N}}.$$

The very last thing we have to account for is the fact that we don't know the population statistics μ and σ^2. Instead, we have to use the sample statistics $\bar{x}$ and s^2. Furthermore, we have to use the t-distribution instead of the normal distribution, because we are estimating the population variance from the sample variance. The t-distribution has a shape similar to the normal distribution, but it has heavier tails, which accounts for the additional uncertainty introduced by estimating the population variance. Thus, we replace μ with $\bar{x}$ and σ^2 with s^2, and we use the t-distribution with $N-1$ degrees of freedom. This gives us the final expression for the confidence interval:

$$X_{1,2} = \bar{x} \pm t^*_{N-1} \cdot \frac{s}{\sqrt{N}},$$

where t^*_{N-1} is the critical value from the t-distribution with $N-1$ degrees of freedom.

6.4 the solution

Let's say I measured the heights of 30 7-year-old boys, and this is the data I got:

```
N = 30
np.random.seed(271)
sample = norm.rvs(size=N, loc=mu_boys, scale=sigma_boys)
print(f"Sample mean: {np.mean(sample):.2f} cm")
print(sample)
```

```
Sample mean: 122.60 cm
[114.15972134 128.21581493 122.9864136  117.94247325 132.11013925
 118.69131645 123.67695468 112.03152008 121.59853424 114.8629358
 121.90458112 115.68839748 127.18043069 118.33193499 125.28525617
 124.5287395  120.72706375 113.10575734 132.229147   129.16820684
 125.94682095 126.08299475 125.95056303 125.6858065  115.07854075
 124.93539918 125.12886271 126.91366971 120.88030405 127.04777082]
```

Using the formula for the confidence interval we get:

```
alpha = 0.05
z_crit = scipy.stats.t.isf(alpha/2, N-1)
CI = z_crit * sample.std(ddof=1) / np.sqrt(N)
CI_low = np.round(sample.mean() - CI, 2)
CI_high = np.round(sample.mean() + CI, 2)
print(f"Sample mean: {np.mean(sample):.2f} cm")
print("The 95% confidence interval is [{}, {}] cm".format(CI_low, CI_high))
print(f"The true population mean is {mu_boys:.2f} cm")
```

```
Sample mean: 122.60 cm
The 95% confidence interval is [120.54, 124.67] cm
The true population mean is 121.74 cm
```

6.5 a few points to stress

It is worth commenting on a few points:

- If we were to sample a great many number of samples of size $N = 30$, and compute the confidence interval for each sample, then approximately 95% of these intervals would contain the true population mean μ.
- It is not true that the probability that the true population mean μ is in the confidence interval is 95%. The true population mean is either in the interval or not, and it does not have a probability associated with it. The 95% confidence level refers to the long-run frequency of intervals containing the true population mean if we were to repeat the sampling process many times. This is the common *frequentist* interpretation of confidence intervals.
- If you want to talk about confidence interval in the *Bayesian* framework, then first we would have to assign a prior distribution to the population mean μ, and then we would compute the posterior distribution of μ given the data. The credible interval is then the interval that contains 95% of the posterior distribution of μ.
- To sum up the difference between the frequentist and Bayesian interpretations of confidence intervals:
  - Frequentist CI: "I am 95% confident in the method" (long-run frequency).
  - Bayesian credible interval: "There is a 95% probability that lies in this interval" (degree of belief).

7 empirical confidence interval

Not always we want to compute the confidence interval of the mean. Sometimes we are interested in a different statistic, such as the median, the standard deviation, or the maximum. The equations we saw before for the confidence interval of the mean do not apply to these statistics. However, we can still compute a confidence interval for them using the empirical bootstrap method.

7.1 bootstrap confidence interval

1. Draw a sample of size N from the population. Let's assume you made an experiment and you could only afford to collect N samples. You will not have the opportunity to collect more samples, and that's all you have available.
2. Assume that the sample is representative of the population. This is a strong assumption, but we will use it to compute the confidence interval.
3. From this original sample, draw B bootstrap samples of size N with replacement. This means that you will randomly select N samples from the original sample, allowing for duplicates. This is like drawing pieces of paper from a hat, where you can put the paper back after drawing it.
4. For each bootstrap sample, compute the statistic of interest (e.g., median, standard deviation, maximum).
5. Compute the cdf of the bootstrap statistics. This will give you the empirical distribution of the statistic.
6. Compute the confidence interval using the empirical distribution. For a 95% confidence interval, you can take the 2.5th and 97.5th percentiles of the bootstrap statistics.

That's it. Now let's do it in code.

7.2 question

We have a sample of 30 7-year-old boys. What can we say about the maximum height of 7-year-olds in the general population?

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
import scipy
# %matplotlib widget
```

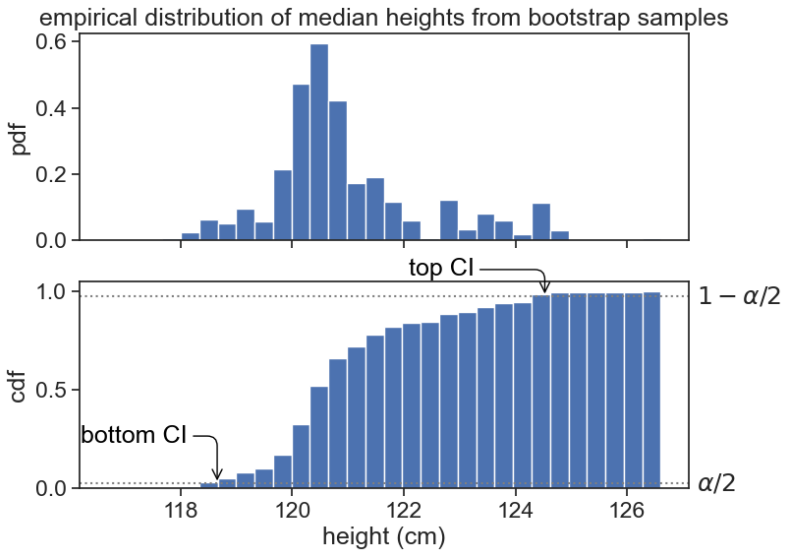
```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
mu_boys = df_boys.loc[7.0, 'mu']
sigma_boys = df_boys.loc[7.0, 'sigma']
```

```
N = 30     # bootstrap sample size equal to original sample size
B = 10000  # number of bootstrap samples
np.random.seed(1)
sample = norm.rvs(size=N, loc=mu_boys, scale=sigma_boys)
median_list = []
for i in range(B):
    sample_bootstrap = np.random.choice(sample, size=N, replace=True)
    median_list.append(np.median(sample_bootstrap))
median_list = np.array(median_list)

alpha = 0.05
ci_bottom = np.quantile(median_list,alpha/2)
ci_top = np.quantile(median_list, 1-alpha/2)
print(f"Bootstrap CI for median: {ci_bottom:.2f} - {ci_top:.2f} cm")
```

```
Bootstrap CI for median: 118.65 - 124.53 cm
```

```
fig, ax = plt.subplots(2,1, figsize=(8, 6), sharex=True)
ax[0].hist(median_list, bins=30, density=True, align='mid')
ax[1].hist(median_list, bins=30, density=True, cumulative=True, align='mid')
```

```
ax[1].axhline(alpha/2, color='gray', linestyle=':')
ax[1].axhline(1-alpha/2, color='gray', linestyle=':')

xlim = ax[1].get_xlim()
ax[1].text(xlim[1]+0.15, alpha/2, r"$\alpha/2$",
           ha="left", va="center")
ax[1].text(xlim[1]+0.15, 1-alpha/2, r"$1-\alpha/2$",
           ha="left", va="center")

ax[1].annotate(
     'bottom CI',
     xy=(ci_bottom, alpha/2), xycoords='data',
     xytext=(-100, 30), textcoords='offset points',
     color='black',
     arrowprops=dict(arrowstyle="->", color='black',
                     connectionstyle="angle,angleA=0,angleB=90,rad=10"))
ax[1].annotate(
     'top CI',
     xy=(ci_top, 1-alpha/2), xycoords='data',
     xytext=(-100, 15), textcoords='offset points',
     color='black',
     arrowprops=dict(arrowstyle="->", color='black',
                     connectionstyle="angle,angleA=0,angleB=90,rad=10"))

ax[0].set(ylabel="pdf",
          title="empirical distribution of median heights from bootstrap samples")
ax[1].set(ylabel="cdf",
          xlabel="height (cm)")
```



Clearly, the distribution of median height is not normal. The bootstrap method gives us a way to compute the confidence interval of the median height (or any other statistic of your choosing) without assuming normality.

Part IV

permutation

8 the problem with t-test

Let's go back to the example of the independent samples t-test.

We sampled 10 boys and 14 girls, age 12, and asked:

> Are 12-year old girls significantly taller than 12-year old boys?

We then went about answering this question by talking about the means of each sample, and if the differences between the means were large enough to be considered significant.

The whole machinery behind the t-test is based on the normality assumption.

8.1 the normality assumption

Two possible interpretations come to mind.

1. The assumption is that the height of men and women in the *population* is normally distributed. From these idealized populations we draw samples.
2. The t-test effectively compares the difference between the means of the two samples, and the variability within each sample. Because of the Central Limit Theorem, the means of the samples will approach a normal distribution as the sample size increases. In this interpretation, the normality assumption is about the distribution of the means of the samples, and not the distribution of the population.

In the context of the t-test, the above is a distinction without a difference. Even if the population is not normally distributed, the means of the samples will be normally distributed as long as the sample size is large enough. We then use the t-test and go on with our lives.

8.2 other statistical tests

The Central Limit Theorem dictates that the means will be normally distributed, but it does not apply to other statistics, such as:

- the median
- the variance
- the skewness
- the maximum
- the Interquartile Range (IQR)
- etc.

In this case, the t-test can't be relied upon, and we need another solution.

9 permutation test

We wish to compare two samples, and see if they significantly differ regarding some statistic of interest (median, mean, etc.). To make things concrete, let's talk about the heights of 12-year-old boys and girls. Are girls significantly taller than boys?

9.1 hypotheses

- **Null hypothesis (H0):** The two samples come from the same distribution.
- **Alternative hypothesis (H1):** Girls are taller than boys.

The basic idea behind the permutation test is that, *if the null hypothesis is correct*, then it wouldn't matter if we relabelled the samples. If we randomly permute the labels "girls" and "boys" of the two samples, the statistic of interest should not change significantly. However, if by permuting the labels we get a significantly different statistic, then we can reject the null hypothesis.

That's beautiful, right?

9.2 steps

1. Compute the statistic of interest (e.g., the difference in medians) for the original samples.
2. Randomly permute the labels of the two samples.
3. Compute the statistic of interest for the permuted samples.

4. Repeat steps 2 and 3 many times (e.g., 1000 times) to create a distribution of the statistic under the null hypothesis.
5. Compare the original statistic to the distribution of permuted statistics to see if it is significantly different (e.g., by checking if it falls in the top 5% of the distribution). From this, we can numerically compute a p-value.

9.3 example

Let's use the very same example as in the independent samples t-test.

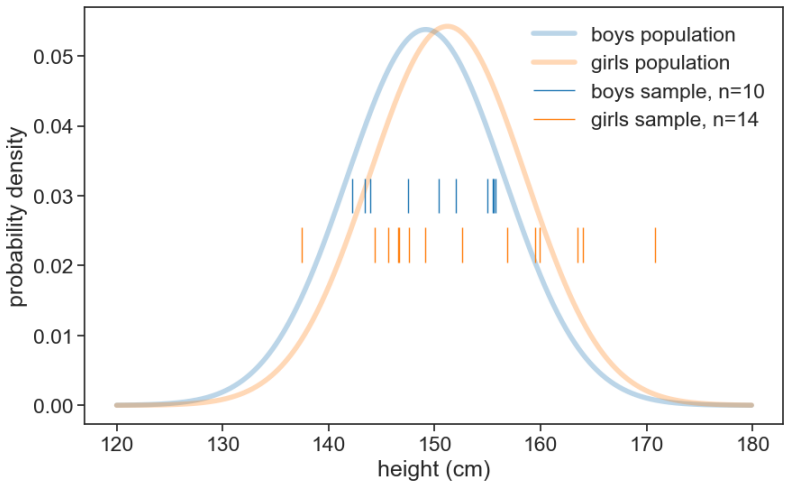
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
# %matplotlib widget
```

```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
df_girls = pd.read_csv('../archive/data/height/girls_height_stats.csv', index_col=0)
age = 12.0
mu_boys = df_boys.loc[age, 'mu']
mu_girls = df_girls.loc[age, 'mu']
sigma_boys = df_boys.loc[age, 'sigma']
sigma_girls = df_girls.loc[age, 'sigma']
```

```
N_boys = 10
N_girls = 14
# set scipy seed for reproducibility
np.random.seed(314)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
```

```
height_list = np.arange(120, 180, 0.1)
pdf_boys = norm.pdf(height_list, loc=mu_boys, scale=sigma_boys)
```

```
pdf_girls = norm.pdf(height_list, loc=mu_girls, scale=sigma_girls)
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(height_list, pdf_boys, lw=4, alpha=0.3, color='tab:blue', label='boys population')
ax.plot(height_list, pdf_girls, lw=4, alpha=0.3, color='tab:orange', label='girls population')

ax.eventplot(sample_boys, orientation="horizontal", lineoffsets=0.03,
             linewidth=1, linelengths= 0.005,
             colors='tab:blue', label=f'boys sample, n={N_boys}')
ax.eventplot(sample_girls, orientation="horizontal", lineoffsets=0.023,
             linewidth=1, linelengths= 0.005,
             colors='tab:orange', label=f'girls sample, n={N_girls}')
ax.legend(frameon=False)
ax.set(xlabel='height (cm)',
       ylabel='probability density',
    );
```



The statistic of interest now is the difference in medians between the two samples.

```
# define the desired statistic.
# in can be anything you want, you can even write your own function.
statistic = np.median
# compute the median for each sample and the difference
median_girls = statistic(sample_girls)
median_boys = statistic(sample_boys)
observed_diff = median_girls - median_boys
```

```
print(f"median height for girls: {median_girls:.2f} cm")
print(f"median height for boys: {median_boys:.2f} cm")
print(f"median difference (girls minus boys): {observed_diff:.2f} cm")
```

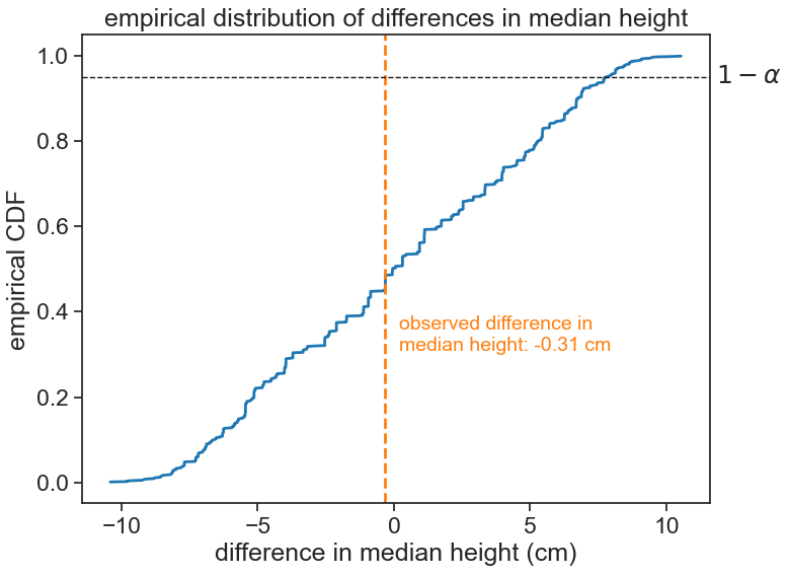
```
median height for girls: 150.88 cm
median height for boys: 151.19 cm
median difference (girls minus boys): -0.31 cm
```

```
N_permutations = 1000
# combine all values in one array
all_data = np.concatenate([sample_girls, sample_boys])
# create an array to store the differences
diffs = np.empty(N_permutations-1)

for i in range(N_permutations - 1):    # this "minus 1" will be explained later
    # permute the labels
    permuted = np.random.permutation(all_data)
    new_girls = permuted[:N_girls]  # first 14 values are girls
    new_boys = permuted[N_girls:]   # remaining values are boys
    diffs[i] = statistic(new_girls) - statistic(new_boys)
# add the observed difference to the array of differences
diffs = np.append(diffs, observed_diff)
```

Now let's see the empirical cdf of the permuted statistics, and where the original statistic falls in that distribution.

```
fig, ax = plt.subplots(figsize=(8, 6))

# compute the empirical CDF
rank = np.arange(len(diffs)) + 1
cdf = rank / (len(diffs) + 1)
sorted_diffs = np.sort(diffs)

ax.plot(sorted_diffs, cdf, lw=2, color='tab:blue', label='empirical CDF')
ax.axvline(observed_diff, color='tab:orange', lw=2, ls='--',
           label=f'obs. median diff. = {observed_diff:.2f} cm')
ax.text(observed_diff + 0.5, 0.3, f'observed difference in\nmedian height: {observed_diff:.2f}
        color='tab:orange', fontsize=14, ha='left', va='bottom')
```

```
alpha = 0.05
# for a one-tailed test
ax.axhline(1-alpha, color='k', lw=1, ls='--')
ax.annotate(r"$1-\alpha$", xy=(1.01, 1-alpha), xycoords=('axes fraction', 'data'),
            ha="left", va="center")
# for a two-tailed test
# ax.axhline(alpha/2, color='k', lw=1, ls='--')
# ax.axhline(1-alpha/2, color='k', lw=1, ls='--')
# ax.annotate(r"$\alpha/2$", xy=(1.01, alpha/2), xycoords=('axes fraction', 'data'),
#             ha="left", va="center")
# ax.annotate(r"$1-\alpha/2$", xy=(1.01, 1-alpha/2), xycoords=('axes fraction', 'data'),
#             ha="left", va="center")
ax.set(xlabel='difference in median height (cm)',
         ylabel='empirical CDF',
         title=f'empirical distribution of differences in median height');
```



The observed statistic is well within the boundaries set by the significance level of 5%. Therefore, we cannot reject the null hypothesis. We conclude that, based on this data, the most probable interpretation is that girls and boys have the same underlying distribution.

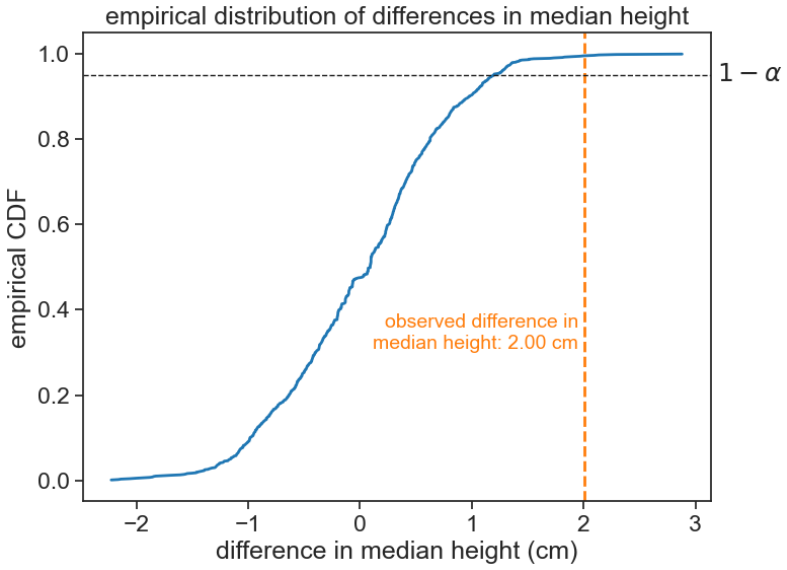
9.4 increase sample size

Let's increase the sample size to 200 girls and 300 boys.

```
# take samples
N_boys = 300
N_girls = 200
# set scipy seed for reproducibility
np.random.seed(314)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)

# compute the observed difference in medians
statistic = np.median
# compute the median for each sample and the difference
median_girls = statistic(sample_girls)
median_boys = statistic(sample_boys)
observed_diff = median_girls - median_boys

# permutation algorithm
N_permutations = 1000
# combine all values in one array
all_data = np.concatenate([sample_girls, sample_boys])
# create an array to store the differences
diffs = np.empty(N_permutations-1)
for i in range(N_permutations - 1):    # this "minus 1" will be explained later
    # permute the labels
    permuted = np.random.permutation(all_data)
    new_girls = permuted[:N_girls]  # first 200 values are girls
    new_boys = permuted[N_girls:]   # remaining values are boys
    diffs[i] = statistic(new_girls) - statistic(new_boys)
# add the observed difference to the array of differences
diffs = np.append(diffs, observed_diff)
```

```
fig, ax = plt.subplots(figsize=(8, 6))

# compute the empirical CDF
rank = np.arange(len(diffs)) + 1
cdf = rank / (len(diffs) + 1)
sorted_diffs = np.sort(diffs)
```

```
ax.plot(sorted_diffs, cdf, lw=2, color='tab:blue', label='empirical CDF')
ax.axvline(observed_diff, color='tab:orange', lw=2, ls='--',
           label=f'obs. median diff. = {observed_diff:.2f} cm')
ax.text(observed_diff - 0.05, 0.3, f'observed difference in\nmedian height: {observed_diff:.2f}
        color='tab:orange', fontsize=14, ha='right', va='bottom')

alpha = 0.05
# for a one-tailed test
ax.axhline(1-alpha, color='k', lw=1, ls='--')
ax.annotate(r"$1-\alpha$", xy=(1.01, 1-alpha), xycoords=('axes fraction', 'data'),
            ha="left", va="center")
# for a two-tailed test
# ax.axhline(alpha/2, color='k', lw=1, ls='--')
# ax.axhline(1-alpha/2, color='k', lw=1, ls='--')
# ax.annotate(r"$\alpha/2$", xy=(1.01, alpha/2), xycoords=('axes fraction', 'data'),
#             ha="left", va="center")
# ax.annotate(r"$1-\alpha/2$", xy=(1.01, 1-alpha/2), xycoords=('axes fraction', 'data'),
#             ha="left", va="center")

ax.set(xlabel='difference in median height (cm)',
         ylabel='empirical CDF',
         title=f'empirical distribution of differences in median height');
```



Now the observed statistic is well outside the right boundary set by the significance level of 5%. Therefore, we can reject the null hypothesis. We conclude that, based on this data, girls are significantly taller than boys.

9.5 p-value

It is quite easy to compute the p-value from the permutation test. It is simply the fraction of permuted statistics that are more extreme than the observed statistic. In this case, since we are testing whether girls are taller than boys, we have a one-tailed test, and we only consider the right tail of the distribution. If we were testing whether girls are significantly different from boys in their height, we would have a two-tailed test, and we would consider both tails of the distribution.

```
# one-tailed p-value
p_value = np.mean(diffs >= observed_diff)
# two-tailed p-value
# p_value = np.mean(np.abs(diffs) >= np.abs(observed_diff))
print(f"observed difference: {observed_diff:.3f}")
print(f"p-value (one-tailed): {p_value:.4f}")
```

```
observed difference: 2.004
p-value (one-tailed): 0.0050
```

We can now address the fact that we ran only 999 permutations, although I intended to run 1000. See in the code that after the permutation algorithm, I inserted the original statistic in the list of permuted statistics. This is because I want to compute the p-value as the fraction of permuted statistics that are more extreme than the original statistic, and I want to include the original statistic in the distribution. If I had not done this, for a truly extreme observed statistic, we would get that the p-value equals 0, that is, the fraction of permuted statistics that are more extreme than the observed statistic is zero. To avoid this behavior, we include the original statistic in the distribution of permuted statistics.

A corollary of this is that the smallest p-value we can get is 0.001 (for our example with 1000 permutations).

10 numpy vs pandas

10.1 numpy

In the previous chapter, we computed the permutation test using `numpy`. We had two samples of different sizes, and before the permutation test we concatenated the two samples into one array. Then we shuffled the concatenated array and split it back into two samples, according to the original sizes. See a sketch of the code below:

Store the two samples in numpy arrays:

```
boys = np.array([121, 123, 124, 125])
girls = np.array([120, 121, 121, 122, 123, 123, 128, 129])
N_boys = len(boys)
N_girls = len(girls)
```

Define the statistic and compute the observed difference:

```
statistic = np.median
# compute the median for each sample and the difference
median_girls = statistic(sample_girls)
median_boys = statistic(sample_boys)
observed_diff = median_girls - median_boys
```

Run the permutation test:

```
N_permutations = 1000
# combine all values in one array
all_data = np.concatenate([sample_girls, sample_boys])
# create an array to store the differences
diffs = np.empty(N_permutations - 1)
```

```
for i in range(N_permutations - 1):    # this "minus 1" will be explained later
    # permute the labels
    permuted = np.random.permutation(all_data)
    new_girls = permuted[:N_girls]  # first N_girls values are girls
    new_boys = permuted[N_girls:]   # remaining values are boys
    diffs[i] = statistic(new_girls) - statistic(new_boys)
# add the observed difference to the array of differences
diffs = np.append(diffs, observed_diff)
```

All this works great if this is how your data looks like. Sometimes, however, you have structured data with more information, such as a DataFrame with multiple columns. In this case, you can leverage the capabilities of `pandas`.

10.2 pandas

Let's give an example of structured data. Suppose we have a DataFrame with the following columns: `sex`, `height`, and `weight`.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
# %matplotlib widget
```

```
N_total = 20
np.random.seed(3)
height_list = norm.rvs(size=N_total, loc=150, scale=7)
weight_list = norm.rvs(size=N_total, loc=42, scale=5)
sex_list = np.random.choice(['M', 'F'], size=N_total, replace=True)
df = pd.DataFrame({
    'sex': sex_list,
    'height (cm)': height_list,
    'weight (kg)': weight_list
})
df
```

	sex	height (cm)	weight (kg)
0	M	162.520399	36.074767
1	M	153.055569	40.971751
2	M	150.675482	49.430742
3	F	136.955551	43.183581
4	F	148.058283	36.881074
5	M	147.516687	38.435034
6	F	149.420810	45.126225
7	F	145.610995	41.197433
8	F	149.693273	38.155818
9	M	146.659474	40.849846
10	M	140.802947	45.725281
11	F	156.192357	51.880554
12	F	156.169226	35.779383
13	M	161.967011	38.867915
14	F	150.350235	37.981170
15	M	147.167258	29.904584
16	M	146.182480	37.381040
17	M	139.174659	36.880621
18	M	156.876572	47.619890
19	M	142.292527	41.340429

Calculate sample statistics using `groupby`:

```
sample_stats = df.groupby('sex')['height (cm)'].median()
observed_diff = sample_stats['F'] - sample_stats['M']
```

We can now leverage the `pandas.DataFrame.sample` method to sample from the DataFrame. Here, we use the following options:

- `frac=1` means we want to sample 100% of rows, but shuffled.
- `replace=False` means we want to sample without replacement, that is, no duplicate rows.

We will shuffle the `sex` column and store the result in a new column called `sex_shuffled`. Then we can use `groupby` to compute the median.

```
N_permutations = 1000
diffs = np.empty(N_permutations - 1)
for i in range(N_permutations - 1):
    # shuffle dataframe 'sex' colunn, store it in 'sex_shuffled'
    df['sex_shuffled'] = df['sex'].sample(frac=1, replace=False).reset_index(drop=True)
    shuffled_stats = df.groupby('sex_shuffled')['height (cm)'].median()
    diffs[i] = shuffled_stats['F'] - shuffled_stats['M']  # median(F) - median(M)
# add the observed difference to the array of differences
diffs = np.append(diffs, observed_diff)
```

11 exact vs. Monte Carlo permutation tests

The permutation tests from before do not sample from the full distribution of the test statistic under the null hypothesis. This would be impppractical if the total number of permutations is large, as it would require computing the test statistic for every possible permutation of the data.

For example, if we have 10 boys and 14 girls, the total number of permutations is almost two million:

$$\binom{24}{14} = \frac{24!}{14! \cdot (24-14)!} = 1961256$$

The expression above is the binomial coefficient, which counts the number of ways to choose 14 samples from a total of 24, without regard to the order of selection. This is why we say "24 choose 14" to refer to the parenthesis above.

There is no preference in "24 choose 14" over "24 choose 10", as both expressions yield the same result. You can verify this on your own.

11.1 Monte Carlo permutation tests

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to obtain numerical results. In the context of permutation tests, Monte Carlo methods do not compute the test statistic for every possible permutation of the data. In the examples from before, we computed 1000 permutations only, and from that we estimated the p-value of the test statistic. If we had run the test more than once, we

would have obtained a different p-value each time, as the test statistic is computed from a random sample of permutations.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
# %matplotlib widget
```

```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
df_girls = pd.read_csv('../archive/data/height/girls_height_stats.csv', index_col=0)
age = 12.0
mu_boys = df_boys.loc[age, 'mu']
mu_girls = df_girls.loc[age, 'mu']
sigma_boys = df_boys.loc[age, 'sigma']
sigma_girls = df_girls.loc[age, 'sigma']
```

```
N_boys = 10
N_girls = 14
# set scipy seed for reproducibility
np.random.seed(314)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
```

```
# define the desired statistic.
# in can be anything you want, you can even write your own function.
statistic = np.median
# compute the median for each sample and the difference
median_girls = statistic(sample_girls)
median_boys = statistic(sample_boys)
observed_diff = median_girls - median_boys
```

```
N_permutations = 1000
# combine all values in one array
all_data = np.concatenate([sample_girls, sample_boys])
# create an array to store the differences
diffs = np.empty(N_permutations-1)
```

```
for i in range(N_permutations - 1):    # this "minus 1" will be explained later
    # permute the labels
    permuted = np.random.permutation(all_data)
    new_girls = permuted[:N_girls]  # first 14 values are girls
    new_boys = permuted[N_girls:]   # remaining values are boys
    diffs[i] = statistic(new_girls) - statistic(new_boys)
# add the observed difference to the array of differences
diffs = np.append(diffs, observed_diff)

p_value = np.mean(diffs >= observed_diff)
# two-tailed p-value
# p_value = np.mean(np.abs(diffs) >= np.abs(observed_diff))
print("Monte Carlo permutation test 1")
print(f"observed difference: {observed_diff:.3f}")
print(f"p-value (one-tailed): {p_value:.4f}")
```

```
Monte Carlo permutation test 1
observed difference: -0.314
p-value (one-tailed): 0.5450
```

```
N_permutations = 1000
# combine all values in one array
all_data = np.concatenate([sample_girls, sample_boys])
# create an array to store the differences
diffs = np.empty(N_permutations-1)

for i in range(N_permutations - 1):    # this "minus 1" will be explained later
    # permute the labels
    permuted = np.random.permutation(all_data)
    new_girls = permuted[:N_girls]  # first 14 values are girls
    new_boys = permuted[N_girls:]   # remaining values are boys
    diffs[i] = statistic(new_girls) - statistic(new_boys)
# add the observed difference to the array of differences
diffs = np.append(diffs, observed_diff)

p_value = np.mean(diffs >= observed_diff)
# two-tailed p-value
# p_value = np.mean(np.abs(diffs) >= np.abs(observed_diff))
print("Monte Carlo permutation test 2")
```

```
print(f"observed difference: {observed_diff:.3f}")
print(f"p-value (one-tailed): {p_value:.4f}")
```

```
Monte Carlo permutation test 2
observed difference: -0.314
p-value (one-tailed): 0.5340
```

As you can see, the p-value in not exactly the same, but the difference is negligible. This is because both times we sampled 1000 permutations that are representative of the full distribution of the test statistic under the null hypothesis.

One more thing. The example above with 10 boys and 14 girls is usually considered small. It is often the case that one has a lot more samples, and the number of permutations can be astronomically large, much much larger than two million.

11.2 exact permutation test

If the total number of permutations is small, we can compute the **exact** p-value by sampling from the full distribution of the test statistic under the null hypothesis. That is to say, we compute the test statistic for every possible permutation of the data.

If we had height measurements of 7 boys and 6 girls, the total number of permutations is:

$$\binom{13}{7} = 1716$$

Any computer can easily handle this number of permutations. How to do it in practice? We will use the `itertools.combinations` function.

```python
import numpy as np
from itertools import combinations

#| code-summary: "generate data"
N_girls = 6
N_boys = 7
# set scipy seed for reproducibility
np.random.seed(314)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)

combined = np.concatenate([sample_girls, sample_boys])
n_total = len(combined)

# observed difference in means
observed_diff = np.median(sample_girls) - np.median(sample_boys)

# generate all combinations of indices for group "girls"
indices = np.arange(n_total)
all_combos = list(combinations(indices, N_girls))

# compute all permutations
diffs = []
for idx_a in all_combos:
    mask = np.zeros(n_total, dtype=bool)
    mask[list(idx_a)] = True
    sample_g = combined[mask]
    sample_b = combined[~mask]
    diffs.append(np.median(sample_g) - np.median(sample_b))

diffs = np.array(diffs)

# exact one-tailed p-value
p_value = np.mean(diffs >= observed_diff)
print(f"Observed difference: {observed_diff:.3f} cm")
print(f"Exact p-value (one-tailed): {p_value:.4f}")
print(f"Total permutations: {len(diffs)}")
```

```
Observed difference: 7.620 cm
Exact p-value (one-tailed): 0.0944
```

```
Total permutations: 1716
```

Attention!

If you read the documentation of the `itertools` library, you might be tempted to use `itertools.permutations` instead of `itertools.combinations`.

Don't do that.

Although we are conductiong a permutation test, we are not interested in the order of the samples, and that is what the `permutations` cares about. For instance, if we have 10 people called

[Alice, Bob, Charlie, David, Eve, Frank, Grace, Heidi, Ivan, Judy]

and we want to randomly assign the label "girl" to 4 of them, we do not care about the order in which we assign the labels. We just want to know which 4 people are assigned the label "girl". The permutation function does care about the order, and that is why we should not use it. Instead, we use the `combinations` function, which return all possible combinations of the data, without regard to the order of selection.

Part V

regression

12 the geometry of regression

12.1 a very simple example

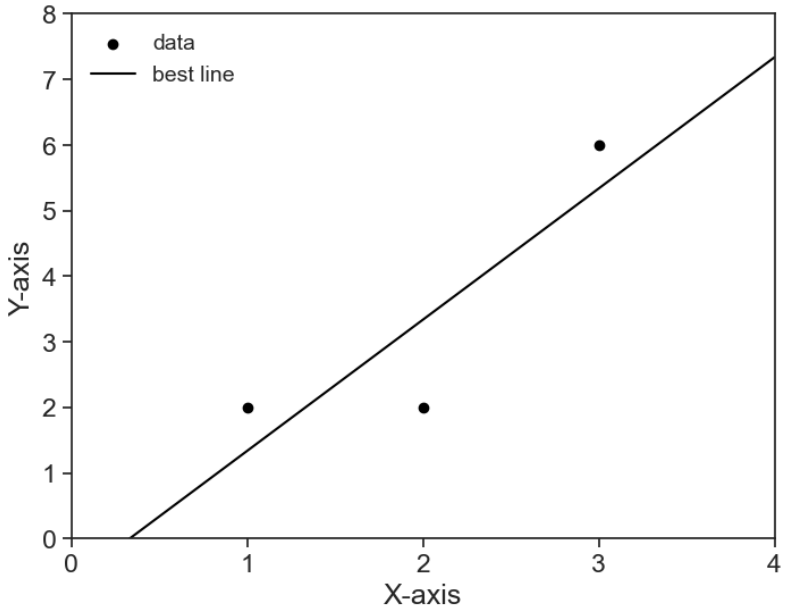
It's almost always best to start with a simple and concrete example.

Goal: We wish to find the best straight line that describes the following data points:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
import scipy

# %matplotlib widget
```

```
x = np.array([1, 2, 3])
y = np.array([2, 2, 6])

fig, ax = plt.subplots(figsize=(8, 6))

ax.scatter(x, y, label='data', facecolors='black', edgecolors='black')
# linear regression
slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x, y)
x_domain = np.linspace(0, 4, 101)
ax.plot(x_domain, intercept + slope * x_domain, color='black', label='best line')

ax.legend(loc='upper left', fontsize=14, frameon=False)
ax.set(xlim=(0, 4),
       ylim=(0, 7),
       xticks=np.arange(0, 5, 1),
```

```
        yticks=np.arange(0, 9, 1),
        xlabel='X-axis',
        ylabel='Y-axis');
```



12.2 formalizing the problem

Let's translate this problem into the language of linear algebra.

The independent variable x is the column vector

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

and the dependent variable y is the column vector

$$y = \begin{pmatrix} 2 \\ 2 \\ 6 \end{pmatrix}.$$

Because we are looking for a straight line, we can express the relationship between x and y as

$$\tilde{y} = \beta_0 + \beta_1 x.$$

Here we introduced the notation $\tilde{y}$ to denote the predicted values of y based on the linear model. It is different from the actual values of y because the straight line usually does not pass exactly on top of y.

The parameter β_0 is the intercept and β_1 is the slope of the line.

Which values of β_0, β_1 will give us the very best line?

12.3 higher dimensions

It is very informative to think about this problem not as a scatter plot in the $X-Y$ plane, but as taking place in a higher-dimensional space. Because we have three data points, we can think of the problem in a three-dimensional space. We want to explain the vector y as a linear combination of the vector x and a constant vector (this is what our linear model states).

In three dimensions, our building blocks are the vectors c, the intercept, and x, the data points.

$$c = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \qquad x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

We can combine these c and x as column vectors in a matrix called **design matrix**:

$$X = \begin{pmatrix} 1 & x_0 \\ | & | \\ 1 & x_i \\ | & | \\ 1 & x_n \end{pmatrix} = \begin{pmatrix} | & | \\ 1 & x \\ | & | \end{pmatrix}$$

Why is this convenient? Because now the linear combination of $\vec{1}$ and x can be expressed as a matrix multiplication:

$$\begin{pmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \end{pmatrix} = \begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ 1 & x_2 \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} = \begin{pmatrix} 1 \cdot \beta_0 + x_0 \cdot \beta_1 \\ 1 \cdot \beta_0 + x_1 \cdot \beta_1 \\ 1 \cdot \beta_0 + x_2 \cdot \beta_1 \end{pmatrix}$$

In short, the linear combination of our two building blocks yields a prediction vector $\hat{y}$:

$$\hat{y} = X\beta,$$

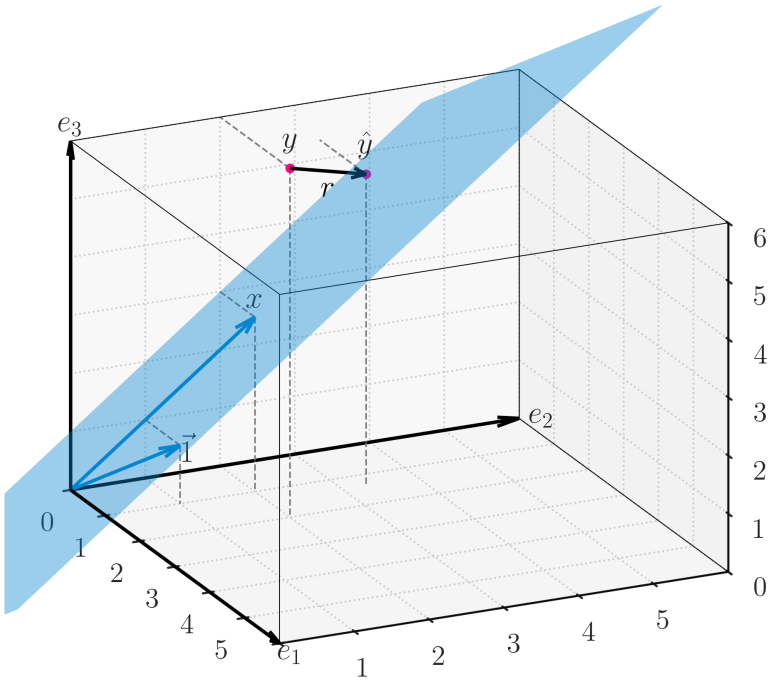
where β is the column vector $(\beta_0, \beta_1)^T$.

This prediction vector $\hat{y}$ lies on a plane in the 3d space, it cannot be anywhere in this 3d space. Mathematically, we say that the vector $\hat{y}$ is in the subspace spanned by the columns of the design matrix X.

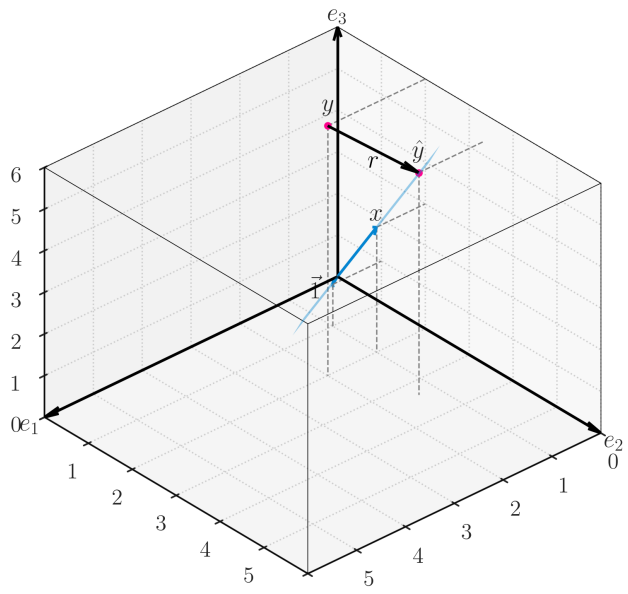
It will be extremely improbable that the vector y will also lie on this plane, so we will have to find the best prediction $\hat{y}$ that lies on this plane. Geometrically, our goal is to find the point $\hat{y}$ on the plane that is **closest** to the point y in the 3d space.

- When the distance $r = y - \hat{y}$ is minimized, the vector r is orthogonal to the plane spanned by the columns of the design matrix X.
- We call this vector r the **residual vector**.
- The residual is orthogonal to each of the columns of X, that is, $\vec{1} \cdot r = 0$ and $x \cdot r = 0$.

I tried to summarize all the above in the 3d image below. This is, for me, the *geometry of regression*. If you have that in your head, you'll never forget it.



Another angle of the image above. This time, because the view direction is within the plane, we see that the residual vector r is orthogonal to the plane spanned by the columns of the design



matrix X.

For a fully interactive version, see this Geogebra applet.

Taking advantage of the matrix notation, we can express the orthogonality condition as follows:

$$\begin{pmatrix} - & 1 & - \\ - & x & - \end{pmatrix} r = X^T r = 0$$

Let's substitute $r = y - \hat{y} = y - X\beta$ into the equation above.

$$X^T(y - X\beta) = 0$$

Distributing yields

$$X^T y - X^T X\beta = 0,$$

and then

$$X^T X\beta = X^T y.$$

We need to solve this equation for β, so we left-multiply both sides by the inverse of $X^T X$,

$$\beta = (X^T X)^{-1} X^T y.$$

That's it. We did it. Given the data points x and y, we can compute the parameters β_0 and β_1 that bring $\hat{y}$ as close as possible to y. These parameters are the best fit of the straight line to the data points.

12.4 overdetermined system

The *design matrix* X is a tall and skinny matrix, meaning that it has more rows (n) than columns (m). This is called an **overdetermined system**, because we have more equations (rows) than unknowns (columns), so we have no hope in finding an exact solution β.

This is to say that, almost certainly, the vector y does not lie on the plane spanned by the columns of the design matrix X. No combination of the parameters β will yield a vector $\hat{y}$ that is exactly equal to y.

12.5 least squares

The method above for finding the best parameters β is called **least squares**. The name comes from the fact that we are trying to minimize the length of the residual vector

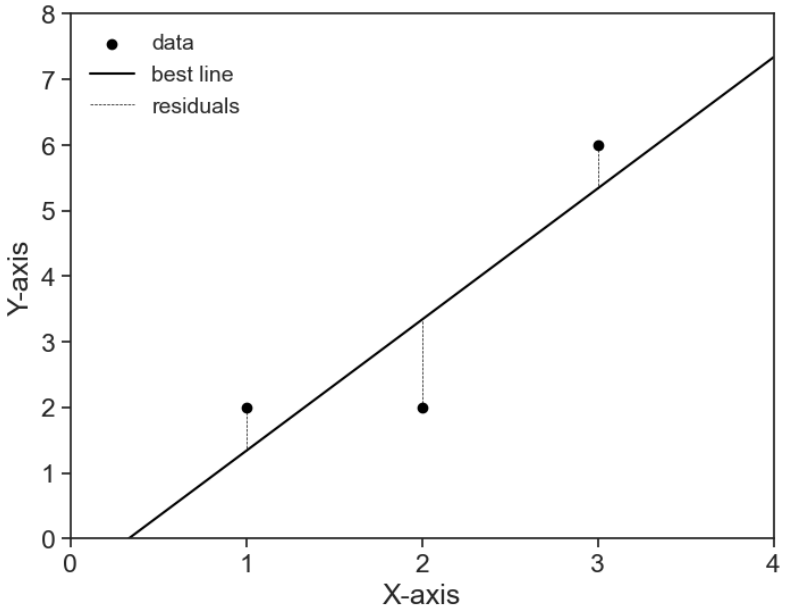
$$r = y - \hat{y}.$$

The length of the residual is given by the Euclidean norm (or L^2 norm), which is a direct generalization of the Pythagorean theorem for many dimensions.

$$\|r\|^2 = \|y - \hat{y}\|^2 \tag{12.1}$$
$$= (y_0 - \hat{y}_0)^2 + (y_1 - \hat{y}_1)^2 + \cdots + (y_{n-1} - \hat{y}_{n-1})^2 \tag{12.2}$$
$$= r_0^2 + r_1^2 + \cdots + r_{n-1}^2 \tag{12.3}$$

The length (squared) of the residual vector is the sum of the squares of all residuals. The best parameters β are those that yield the **least squares**, thus the name.

```
fig, ax = plt.subplots(figsize=(8, 6))

ax.scatter(x, y, label='data', facecolors='black', edgecolors='black')
x_domain = np.linspace(0, 4, 101)
ax.plot(x_domain, intercept + slope * x_domain, color='black', label='best line')

def linear(x, slope, intercept):
    return intercept + slope * x

for i, xi in enumerate(x):
    ax.plot([xi, xi],
            [y[i], linear(xi, slope, intercept)],
            color='black', linestyle='--', linewidth=0.5,
            label='residuals' if i == 0 else None)

ax.legend(loc='upper left', fontsize=14, frameon=False)
ax.set(xlim=(0, 4),
       ylim=(0, 7),
       xticks=np.arange(0, 5, 1),
       yticks=np.arange(0, 9, 1),
       xlabel='X-axis',
       ylabel='Y-axis');
```



12.6 many more dimensions

The concrete example here dealt with only three data points, therefore we could visualize the problem in a three-dimensional space. However, the same reasoning applies to any number of data points and any number of independent variables.

- **any number of data points**: we call the number of data points n, and that makes y be a vector in an n-dimensional space.
- **any number of independent variables**: we calculated a regression for a straight line, and thus we had only two building blocks, the intercept $\vec{1}$ and the independent variable x. However, we can have any number of independent variables, say m of them. For example, we might want to predict the data using a polynomial of degree m, or we might have any arbitrary m functions that we wish to use: $\exp(x)$, $\tanh(x^2)$, whatever we want. All this will work as long as the parameters β multiply these building blocks. That's the topic of the next chapter.

13 least squares

13.1 ordinary least squares (OLS) regression

Let's go over a few things that appear in this notebook, statsmodels, Ordinary Least Squares

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
np.random.seed(9876789)
```

13.2 polynomial regression

Let's start with a simple polynomial regression example. We will start by generating synthetic data for a quadratic equation plus some noise.

```
# number of points
nsample = 100
# create independent variable x
x = np.linspace(0, 10, 100)
# create design matrix with linear and quadratic terms
X = np.column_stack((x, x ** 2))
# create coefficients array
beta = np.array([5, -2, 0.5])
# create random error term
e = np.random.normal(size=nsample)
```

x and e can be understood as column vectors of length n, while X and β are:

$$X = \begin{pmatrix} x_0 & x_0^2 \\ | & | \\ x_i & x_i^2 \\ | & | \\ x_n & x_n^2 \end{pmatrix}, \qquad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}.$$

Oops, there is no intercept column $\vec{1}$ in the design matrix X. Let's add it:

```
X = sm.add_constant(X)
print(X[:5, :])  # print first 5 rows of design matrix
```

```
[[1.         0.         0.        ]
 [1.         0.1010101  0.01020304]
 [1.         0.2020202  0.04081216]
 [1.         0.3030303  0.09182736]
 [1.         0.4040404  0.16324865]]
```

This `add_constant` function is smart, it has as default a `prepend=True` argument, meaning that the intercept is added as the first column, and a `has_constant='skip'` argument, meaning that it will not add a constant if one is already present in the matrix.

The matrix X is now a **design matrix** for a polynomial regression of degree 2.

$$X = \begin{pmatrix} 1 & x_0 & x_0^2 \\ | & | & | \\ 1 & x_i & x_i^2 \\ | & | & | \\ 1 & x_n & x_n^2 \end{pmatrix}$$

We now put everything together in the following equation:

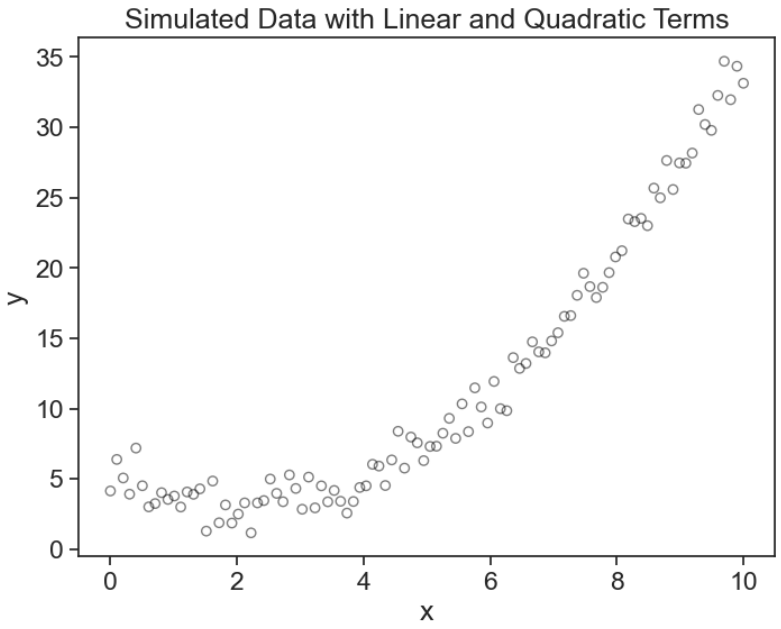
$$y = X\beta + e$$

This creates the dependend variable y as a linear combination of the independent variables in X and the coefficients in β, plus an error term e.

```
y = np.dot(X, beta) + e
```

Let's visualize this:

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(x, y, label='data', facecolors='none', edgecolors='black', alpha=0.5)
ax.set(xlabel='x',
       ylabel='y',
       title='Simulated Data with Linear and Quadratic Terms'
       );
```



13.3 solving the "hard way"

I'm going to do something that nobody does. I will use the formula we derived in the previous chapter to find the coefficients β of the polynomial regression.

$$\beta = (X^T X)^{-1} X^T y.$$

Translating this into code, and keeping in mind that matrix multiplication in Python is done with the `@` operator, we get:

```
beta_opt = np.linalg.inv(X.T@X)@X.T@y
print(f"beta = {beta_opt}")
```

```
beta = [ 5.34233516 -2.14024948  0.51025357]
```

That's it. We did it (again).

Let's take a look at the matrix $X^T X$. Because X is a tall and skinny matrix of shape $(n, 3)$, the matrix X^T is a wide and short matrix of shape $(3, n)$. This is because we have many more data points n than the number of predictors $(\vec{1}, x, x^2)$, which is of course equal to the number of coefficients $(\beta_0, \beta_1, \beta_2)$.

```
print(X.T@X)
```

```
[[1.00000000e+02 5.00000000e+02 3.35016835e+03]
 [5.00000000e+02 3.35016835e+03 2.52525253e+04]
 [3.35016835e+03 2.52525253e+04 2.03033670e+05]]
```

When we multiply the matrices $X^T_{3\times n}$ and $X_{n\times 3}$, we get a square matrix of shape $(3, 3)$, because the inner dimensions match (the number of columns in X^T is equal to the number of rows in X). The product $X^T X$ is a square matrix of shape $(3, 3)$, which is quite easy to invert. If it were the other way around $(X\, X^T)$, we would have a matrix of shape (n, n), which is much harder to invert, especially if n is large. Lucky us.

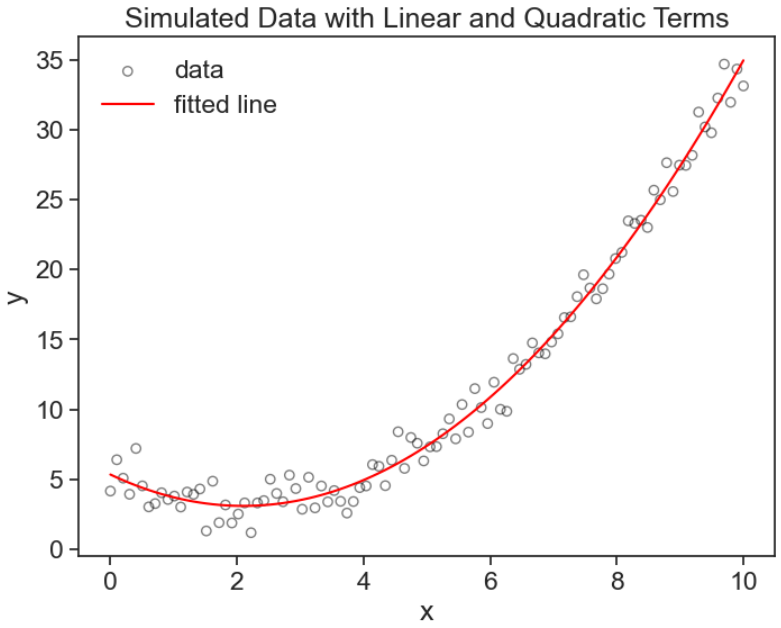
Now let's see if the parameters we found are any good.

```
print("beta parameters used to generate data:")
print(beta)
print("beta parameters estimated from data:")
print(beta_opt)
```

```
beta parameters used to generate data:
[ 5.  -2.   0.5]
beta parameters estimated from data:
[ 5.34233516 -2.14024948  0.51025357]
```

Pretty good, right? Now let's see the best fit polynomial on the graph.

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(x, y, label='data', facecolors='none', edgecolors='black', alpha=0.5)
ax.plot(x, np.dot(X, beta_opt), color='red', label='fitted line')
ax.legend(frameon=False)
ax.set(xlabel='x',
       ylabel='y',
       title='Simulated Data with Linear and Quadratic Terms'
       );
```



Why did I call it the "hard way"? Because these operations are so common that of course there are libraries that do this for us. We don't need to remember the equation, we can just use, for example, `statsmodels` library's `OLS` function, which does exactly this. Let's see how it works.

```
model = sm.OLS(y, X)
results = model.fit()
```

Now we can compare the results of our manual calculation with the results from `statsmodels`. We should get the same coefficients, and we do.

```
print("beta parameters used to generate data:")
print(beta)
print("beta parameters from our calculation:")
print(beta_opt)
print("beta parameters from statsmodels:")
print(results.params)
```

```
beta parameters used to generate data:
[ 5.  -2.   0.5]
beta parameters from our calculation:
[ 5.34233516 -2.14024948  0.51025357]
beta parameters from statsmodels:
[ 5.34233516 -2.14024948  0.51025357]
```

13.4 statmodels.OLS and the summary

Statmodels provides us a lot more information than just the coefficients. Let's take a look at the summary of the OLS regression.

```
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.988
Model:                            OLS   Adj. R-squared:                  0.988
Method:                 Least Squares   F-statistic:                     3965.
Date:                Mon, 23 Jun 2025   Prob (F-statistic):           9.77e-94
Time:                        12:50:31   Log-Likelihood:                -146.51
No. Observations:                 100   AIC:                             299.0
```

```
Df Residuals:                       97   BIC:                             306.8
Df Model:                            2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          5.3423      0.313     17.083      0.000       4.722       5.963
x1            -2.1402      0.145    -14.808      0.000      -2.427      -1.853
x2             0.5103      0.014     36.484      0.000       0.482       0.538
==============================================================================
Omnibus:                        2.042   Durbin-Watson:                   2.274
Prob(Omnibus):                  0.360   Jarque-Bera (JB):                1.875
Skew:                           0.234   Prob(JB):                        0.392
Kurtosis:                       2.519   Cond. No.                         144.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

I won't go into the details of the summary, but I encourage you to take a look at it and see if you can make sense of it.

13.5 R-squared

R-squared is a measure of how well the model fits the data. It is defined as the proportion of the variance in the dependent variable that is predictable from the independent variables. It can be computed as follows:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where SS_{res} and SS_{tot} are defined as follows:

$$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

$$SS_{tot} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

The letters SS mean "sum of squares", and $\bar{y}$ is the mean of the dependent variable. Let's compute it manually, and then compare it with the value from the `statsmodels` summary.

```
y_hat = np.dot(X, beta_opt)
SS_res = np.sum((y - y_hat) ** 2)
SS_tot = np.sum((y - np.mean(y)) ** 2)
R2 = 1 - (SS_res / SS_tot)
print("R-squared (manual calculation): ", R2)
print("R-squared (from statsmodels): ", results.rsquared)
```

```
R-squared (manual calculation):  0.9879144521349076
R-squared (from statsmodels):  0.9879144521349076
```

This high R^2 value tells us that the model explains a very large proportion of the variance in the dependent variable.

How can we know that the variance has anything to do with the R^2? If we divide both the SS_{res} and SS_{tot} by $n-1$, we get the sample variances of the residuals and the dependent variable, respectively.

$$s^2_{res} = \frac{SS_{res}}{n-1} = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n-1}$$
$$s^2_{tot} = \frac{SS_{tot}}{n-1} = \frac{\sum_{i=1}^{n}(y_i - \bar{y})^2}{n-1}$$

Then the R^2 can then be expressed as:

$$R^2 = 1 - \frac{s^2_{res}}{s^2_{tot}}.$$

I prefer this equation over the first, because it makes it clear that R^2 is the ratio of the variances, which is a more intuitive way to think about it.

13.6 any function will do

The formula we derived the the previous chapter works for predictors (independent variables) of any kind, not only polynomials. The formula will work as long as the parameters β are linear in the predictors. For exammple, we could have a nonlinear function like this:

$$y = \beta_0 + \beta_1 e^x + \beta_2 \sin(x^2)$$

because each beta multiplies a predictor. On the other hand, the following function would not work, because the parameters are not linear in the predictors:

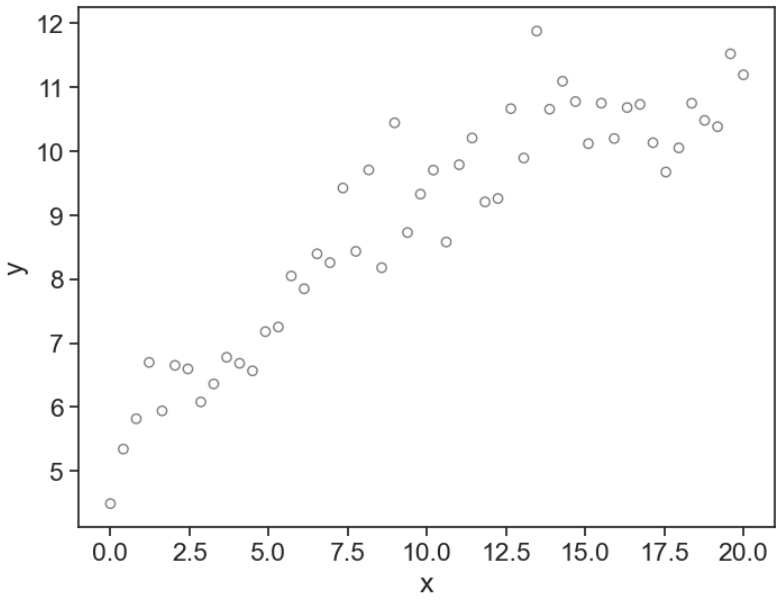
$$y = \beta_0 + e^{\beta_1 x} + \sin(\beta_2 x^2)$$

Let's this this in action, I'll use the same example provided by `statsmodels` documentation, which is a nonlinear function of the form:

$$y = \beta_0 x + \beta_1 \sin(x) + \beta_2 (x - 5)^2 + \beta_3$$

```
nsample = 50
sig = 0.5
x = np.linspace(0, 20, nsample)
X = np.column_stack((
    x,
    np.sin(x),
    (x - 5) ** 2,
    np.ones(nsample)
    ))
beta = [0.5, 0.5, -0.02, 5.0]
y_true = np.dot(X, beta)
y = y_true + sig * np.random.normal(size=nsample)
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(x, y, label='data', facecolors='none', edgecolors='black', alpha=0.5)
ax.set(xlabel='x',
       ylabel='y',
       )
```




```
result = sm.OLS(y, X).fit()
print(result.summary())
```

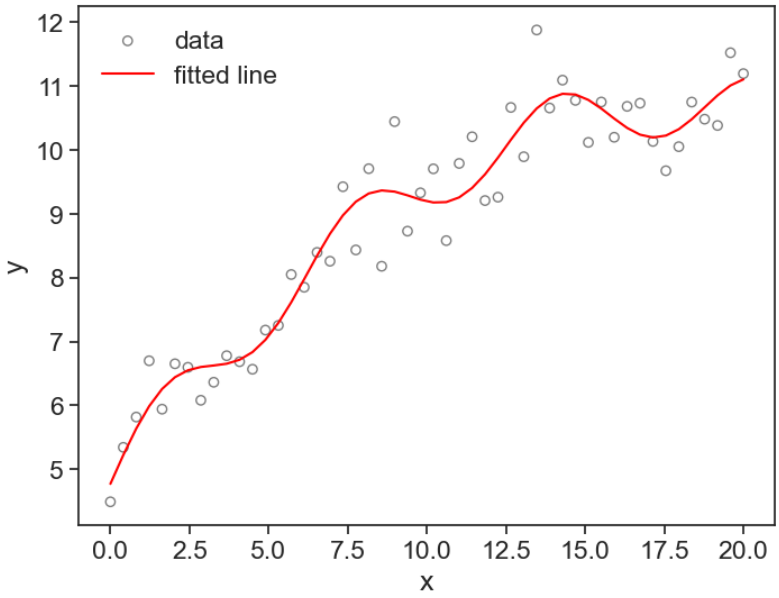
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.933
Model:                            OLS   Adj. R-squared:                  0.928
Method:                 Least Squares   F-statistic:                     211.8
Date:                Mon, 23 Jun 2025   Prob (F-statistic):           6.30e-27
Time:                        12:51:08   Log-Likelihood:                -34.438
No. Observations:                  50   AIC:                             76.88
Df Residuals:                      46   BIC:                             84.52
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
```

```
------------------------------------------------------------------------------
x1             0.4687      0.026     17.751      0.000       0.416       0.522
x2             0.4836      0.104      4.659      0.000       0.275       0.693
x3            -0.0174      0.002     -7.507      0.000      -0.022      -0.013
const          5.2058      0.171     30.405      0.000       4.861       5.550
==============================================================================
Omnibus:                        0.655   Durbin-Watson:                   2.896
Prob(Omnibus):                  0.721   Jarque-Bera (JB):                0.360
Skew:                           0.207   Prob(JB):                        0.835
Kurtosis:                       3.026   Cond. No.                         221.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Note something interesting: in our design matrix X, we encoded the intercept column as the last column, there is no reason why it should be the first column (although first column is a common choice). The function 'statsmodels.OLS' sees this, and when we print the summary, it will show the intercept as the last coefficient. Nice!

Let's see a graph of the data and the fitted model.

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(x, y, label='data', facecolors='none', edgecolors='black', alpha=0.5)
ax.plot(x, np.dot(X, result.params), color='red', label='fitted line')
ax.legend(frameon=False)
ax.set(xlabel='x',
       ylabel='y',
       )
```


data
fitted line
y
x
12
11
10
9
8
7
6
5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

14 equivalence

14.1 orthogonality

In the context of linear regression, the orthogonality condition states that the residual vector r is orthogonal to the column space of the design matrix X:

$$X^T r = 0$$

Let's substitute $r = y - \hat{y} = y - X\beta$ into the equation above.

$$X^T(y - X\beta) = 0$$

Distributing yields

$$X^T y - X^T X\beta = 0,$$

and then

$$X^T X\beta = X^T y.$$

We need to solve this equation for β, so we left-multiply both sides by the inverse of $X^T X$,

$$\beta = (X^T X)^{-1} X^T y.$$

We already did that in a previous chapter. Now let's get exactly the same result using another approach.

14.2 optimization

We wish to minimize the sum of squared errors, which is given by

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - X_{ij}\beta_j)^2.$$

I'm not exactly sure, but I think we call this L because of the lagrangian function. Later on when we talk about regularization, we will see that the lagrangian function can be constrained by lagrange multipliers. In any case, let's keep going with the optimization.

It is useful to express L in matrix notation. The sum of squared errors can be thought as the dot product of the residual vector with itself:

$$\begin{aligned} L &= (y - X\beta)^T (y - X\beta) \\ &= y^T y - y^T X\beta - \beta^T X^T y + \beta^T X^T X\beta, \end{aligned}$$

where we used the following properties of matrix algebra: 1. The dot product of a vector with itself is the sum of the squares of its components, i.e., $a^T a = \sum_{i=1}^{n} a_i^2$. We used this to express the sum of squared errors in matrix notation. 2. The dot product is bilinear, i.e., $(a-b)^T(c-d) = a^T c - a^T d - b^T c + b^T d$. We used this to expand the expression for the sum of squared errors. 3. The transpose of a product of matrices is the product of their transposes in reverse order, i.e., $(AB)^T = B^T A^T$. We used this to compute $(X\beta)^T$.

Let's use one more property to join the two middle terms, $-y^T X\beta - \beta^T X^T y$:

4. The dot product is symmetric, i.e., $a^T b = b^T a$. This is evident we express the dot product as a summation:

$$a^T b = \sum_{i=1}^{n} a_i b_i = \sum_{i=1}^{n} b_i a_i = b^T a.$$

Joining the two middle terms results in the following L:

$$L = y^T y - 2y^T X\beta + \beta^T X^T X\beta,$$

The set of parameters β that minimizes L is that which satisfies the extreme condition of the function L (either maximum or minimum). This means that the gradient of L with respect to β must be zero:

$$\frac{\partial L}{\partial \beta} = 0.$$

Let's plug in the expression for L:

$$\frac{\partial}{\partial \beta}\left(y^T y - 2\beta^T X^T y + \beta^T X^T X\beta\right) = 0$$

The quantity L is a scalar, and also each of the three terms that we are differentiating is a scalar. Let's differentiate them one by one.

The first term, $y^T y$, is a constant with respect to β, so its derivative is zero.

The second term

$$\frac{\partial}{\partial \beta}\left(-2\beta^T X^T y\right) = -2\frac{\partial}{\partial \beta}\left(\beta^T X^T y\right).$$

The quantity being differentiated is a scalar, it's the product of the row vector β^T and the column vector $X^T y$. Right now we don't care much about $X^T y$, it could be any column vector, so let's call it c. The derivative of dot product $\beta^T c$ with respect to a specific element β_k can be written explicitly as

$$\frac{\partial(\beta^T c)}{\partial\beta} = \frac{\partial}{\partial\beta_k}\left(\sum_{i=1}^{p}\beta_i c_i\right) = \frac{\partial}{\partial\beta_k}\left(\beta_1 c_1 + \beta_2 c_2 + ... + \beta_p c_p\right) = c_k.$$

Whatever value for the index k we choose, the derivative will be zero for all indices except for k, and that explain the result above.

Since the gradient $\nabla_\beta(\beta^T c)$ is a vector,

$$\nabla_\beta(\beta^T c) = \frac{\partial(\beta^T c)}{\partial\beta} = \begin{pmatrix} \frac{\partial(\beta^T c)}{\partial\beta_1} \\ \frac{\partial(\beta^T c)}{\partial\beta_2} \\ \vdots \\ \frac{\partial(\beta^T c)}{\partial\beta_p} \end{pmatrix}$$

and we have just figured out what each component is, we can write the solution as

$$\frac{\partial(\beta^T c)}{\partial\beta} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_p \end{pmatrix} = c = X^T y.$$

So the derivative of the second term is simply $-2X^T y$.

To solve the derivatie of the third term, $\beta^T X^T X\beta$, we use the following property:

$$\frac{\partial}{\partial\beta}\left(\beta^T A\beta\right) = 2A\beta,$$

when A is a symmetric matrix. In our case, $A = X^T X$, which is symmetric because $(X^T X)^T = X^T (X^T)^T = X^T X$. Therefore we have:

$$\frac{\partial}{\partial\beta}\left(\beta^T X^T X\beta\right) = 2X^T X\beta,$$

and that is the derivative of the third term.

We still have to prove why the derivative of $\beta^T A\beta$ is $2A\beta$. First, let's use the summation notation to express the term $\beta^T A\beta$:

$$\beta^T A\beta = \sum_{i=1}^{p}\sum_{j=1}^{p} \beta_i A_{ij}\beta_j.$$

Now, let's differentiate this expression with respect to β_k, using the chain rule:

$$\frac{\partial}{\partial\beta_k}\left(\sum_{i=1}^{p}\sum_{j=1}^{p} \beta_i A_{ij}\beta_j\right) = \sum_{i=1}^{p} A_{ik}\beta_i + \sum_{j=1}^{p} \beta_j A_{kj}.$$

Using the symmetry of A ($A_{ij} = A_{ji}$):

$$\frac{\partial}{\partial\beta_k}\left(\sum_{i=1}^{p}\sum_{j=1}^{p} \beta_i A_{ij}\beta_j\right) = 2\sum_{i=1}^{p} A_{ik}\beta_i = 2(A\beta)_k.$$

This is the element k of the vector $2A\beta$. Since this is true for any index k, we can write the gradient as

$$\nabla_\beta(\beta^T A\beta) = 2A\beta.$$

Now that we have the derivatives of all three terms, we can write the gradient of L:

$$\frac{\partial L}{\partial\beta} = 0 - 2X^T y + 2X^T X\beta = 0.$$

Rearranging…

$$X^T X\beta = X^T y$$

…and solving for β:

$$\beta = (X^T X)^{-1} X^T y$$

14.3 discussion

Using two completely different approaches, we arrived at the same result for the least squares solution:

$$\beta = (X^T X)^{-1} X^T y$$

.

- **Approach 1**: We used the orthogonality condition, which states that the residual vector is orthogonal to the column space of the design matrix.
- **Approach 2**: We applied the optimization method, minimizing the sum of squared errors—which corresponds to minimizing the squared length of the residual vector.

There is a deep connection here. The requirement that the residual vector is orthogonal to the column space of the design matrix is equivalent to minimizing the sum of squared errors. We can see this visually: if the projection of the response vector y onto the column space of X were anywhere else, the residual vector would be not only not orthogonal, but also longer!

$$\text{orthogonality} \iff \text{optimization}$$

This result even tranfers to other contexts, as long as there is a vector space with a well defined inner product (dot product) and an orthogonal basis. In these cases, the least squares solution can be interpreted as finding the projection of a vector onto a subspace spanned by an orthogonal basis. Some examples include:

- Fourier series: the Fourier coefficients are the least squares solution to the problem of approximating a function by a sum of sines and cosines, where these functions are an orthogonal basis.
- SVD (Singular Value Decomposition): A matrix can be decomposed into orthogonal matrices, and the singular values can be interpreted as the least squares solution to the problem of approximating a matrix by a sum of outer products of orthogonal vectors.

14.4 other properties

14.4.1 the sum of residuals is zero

As long as the model includes an intercept term, the sum of the residuals is zero. The model could be anything, not necessarily linear regression. Let's prove this property.

$$\hat{y}_i = \beta_0 + f(x_i; \beta_1, \beta_2, \dots, \beta_p)$$

The Ordinary Least Squares (OLS) estimates of the parameters β minimize the sum of squared residuals L. The equation for β_0 reads:

$$\begin{aligned}
\frac{\partial L}{\partial \beta_0} &= \frac{\partial}{\partial \beta_0} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\
&= \frac{\partial}{\partial \beta_0} \sum_{i=1}^{n} \left[y_i - \beta_0 - f(x_i; \beta_1, \beta_2, \dots, \beta_p) \right]^2 \\
&= -2 \sum_{i=1}^{n} \left[y_i - \beta_0 - f(x_i; \beta_1, \beta_2, \dots, \beta_p) \right] \\
&= -2 \sum_{i=1}^{n} (y_i - \hat{y}_i) = 0
\end{aligned}$$

From the last line it follows that the sum of the residuals is zero.

15 partitioning of the sum of squares

One of the most important equations in regression analysis is the partitioning of the sum of squares (SS).

$$SS_{\text{Total}} = SS_{\text{Model}} + SS_{\text{Error}}$$

This equation states that the total variability in the response variable can be partitioned into two components: the variability explained by the regression model and the variability that is not explained by the model (the residuals).

In a more precise mathematical language, the equation states that:

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 + \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where:

- y_i are the observed values,
- $\hat{y}_i$ are the predicted values from the regression model,
- $\bar{y}$ is the mean of the observed values,
- n is the number of observations.

15.1 high-dimensional Pythagorean theorem

Why is this equation true? It's easiest to understand this equation by thinking of it as a high-dimensional version of the Pythagorean theorem:

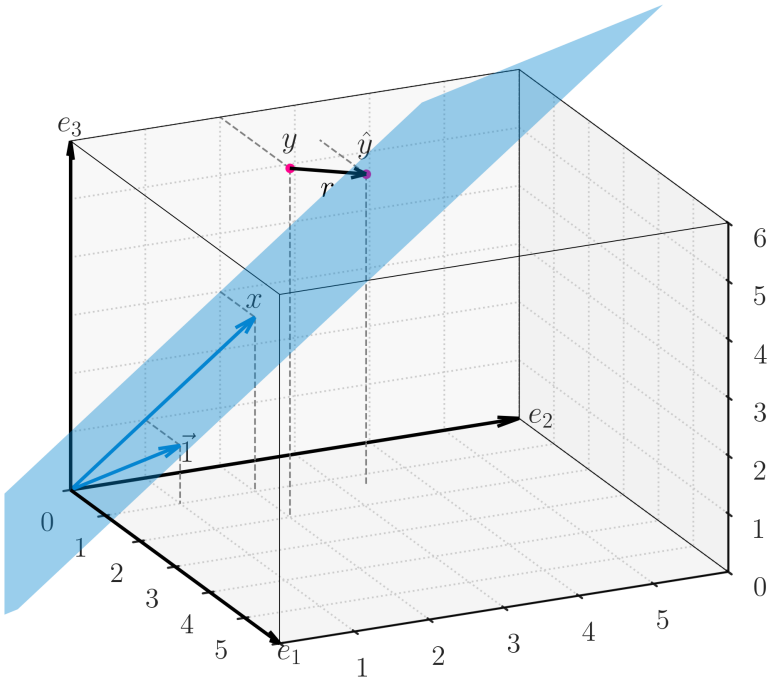
$$\|T\|^2 = \|M\|^2 + \|E\|^2,$$

where:

- $T = y - \bar{y}$ is the total vector, the vector of deviations of the observed values from their mean;
- $M = \hat{y} - \bar{y}$ is the model vector, the vector of deviations of the predicted values from the mean of the observed values;
- $E = y - \hat{y}$ is the error vector, the vector of residuals, or deviations of the observed values from the predicted values.

I omitted the subscript i to emphasize that these are vectors, not scalars.

Make sure you understand the figure below, already presented in a previous chapter.



The vector $r = E$ in black is the residual or error vector. It is orthogonal to the subspace spanned by the column vector of the design matrix, represented in this image by the blue plane.

The vector M is not shown, but it necessarily lies in the blue plane. How do we know that? Because the predicted values $\hat{y}$ are a linear combination of the columns of the design matrix, and therefore $\hat{y}$ lies in the column space of the design matrix. Since $\bar{y}$ is a constant vector (a multiple of the vector of ones), it also lies in the column space of the design matrix. Therefore, $M = \hat{y} - \bar{y}$ lies in the column space of the design matrix.

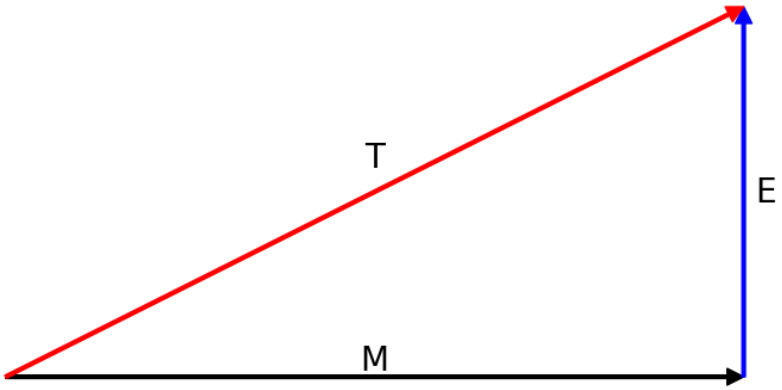
From the above, we can already conclude that E is orthogonal to M. These are the two legs of a right triangle. We now need a hypotenuse. The hypotenuse is the total vector $T = y - \bar{y}$, which is the sum of the other two vectors:

$$T = M + E.$$

This is easy to see by substituting the definitions of M and E:

$$T = y - \bar{y} = (\hat{y} - \bar{y}) + (y - \hat{y}) = M + E.$$

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(8, 6))
ax.set_aspect('equal', adjustable='box')
ax.annotate('', xy=(2, 0), xytext=(0, 0), arrowprops=dict(facecolor='black', edgecolor='black'
ax.annotate('', xy=(2, 1), xytext=(2, 0), arrowprops=dict(facecolor='blue', edgecolor='blue', w
ax.annotate('', xy=(2, 1), xytext=(0, 0), arrowprops=dict(facecolor='red', edgecolor='red', wi

ax.text(1.0, 0.0, 'M', fontsize=20, ha='center', va='bottom')
ax.text(2.03, 0.5, 'E', fontsize=20, ha='left', va='center')
ax.text(1.0, 0.55, 'T', fontsize=20, ha='center', va='bottom')
ax.set_xticks([])
ax.set_yticks([])
ax.set_frame_on(False)
ax.set_xlim(0, 2)
ax.set_ylim(0, 1);
```


T
E
M

16 linear mixed effect model

A mixed effect model is an expansion of the ordinary linear regression model that includes both fixed effects and random effects. The fixed effects are the same as in a standard linear regression (could be with or without interactions), while the random effects account for variability across different groups or clusters in the data.

16.1 practical example

We are given a dataset of annual income (independent variable) and years of education (independent variable) for individuals that studied different majors in university (categorical variable). We want to predict the annual income based on years of education and the major studied, including an interaction term between years of education and major. One more thing: each individual appears more than once in the dataset, so we can assume that there is a random effect associated with each individual.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
import statsmodels.formula.api as smf
```

```
# set seed for reproducibility
np.random.seed(42)
# define parameters
majors = ['Juggling', 'Magic', 'Dragon Taming']
n_individuals = 90  # 30 per major
```

```
years_per_person = np.random.randint(1, 5, size=n_individuals)  # 1 to 4 time points

# assign majors and person IDs
person_ids = [f'P{i+1:03d}' for i in range(n_individuals)]
major_assignment = np.repeat(majors, n_individuals // len(majors))

# simulate data
records = []
for i, pid in enumerate(person_ids):
    major = major_assignment[i]
    n_years = years_per_person[i]
    years = np.sort(np.random.choice(np.arange(1, 21), size=n_years, replace=False))

    # base intercept and slope by major
    if major == 'Juggling':
        base_income = 25_000
        growth = 800
    elif major == 'Magic':
        base_income = 20_000
        growth = 1500
    elif major == 'Dragon Taming':
        base_income = 30_000
        growth = 400  # slower growth

    # add person-specific deviation
    personal_offset = np.random.normal(0, 5000)
    slope_offset = np.random.normal(0, 200)

    for y in years:
        income = base_income + personal_offset + (growth + slope_offset) * y + np.random.normal
        records.append({
            'person': pid,
            'major': major,
            'years_after_grad': y,
            'income': income
        })

df = pd.DataFrame(records)
```

Let's take a look at the dataset. There are many data points,

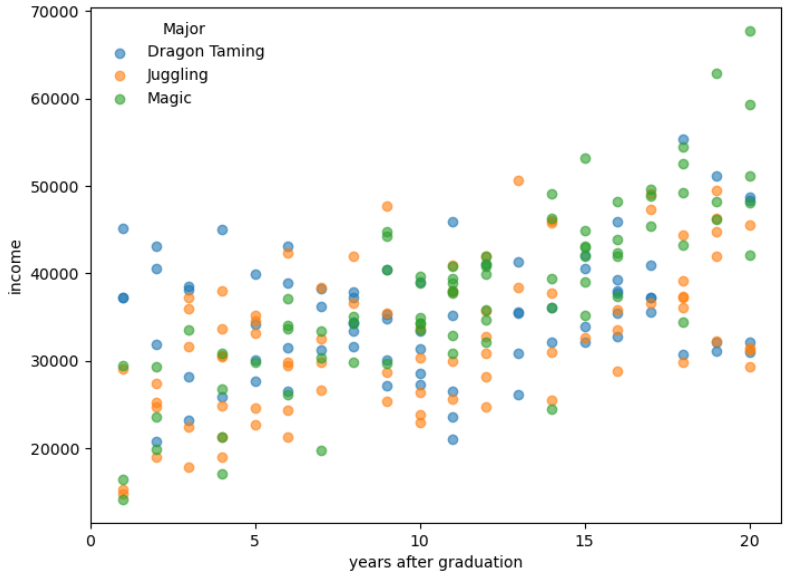
so we will only see 15 points in three different places.

```
print(df[:5])
print(df[90:95])
print(df[190:195])
```

```
  person     major  years_after_grad        income
0   P001  Juggling                 3  37183.719609
1   P001  Juggling                 5  35238.112407
2   P001  Juggling                11  37905.435001
3   P002  Juggling                 2  27432.186391
4   P002  Juggling                 4  30617.926804
   person  major  years_after_grad        income
90   P034  Magic                 1  14151.072305
91   P034  Magic                 7  19716.656861
92   P035  Magic                12  41056.576643
93   P035  Magic                14  46339.987229
94   P036  Magic                16  41981.131518
    person          major  years_after_grad        income
190   P072  Dragon Taming                 7  36173.437735
191   P073  Dragon Taming                 8  33450.564557
192   P074  Dragon Taming                 9  35276.927416
193   P074  Dragon Taming                17  37271.203018
194   P075  Dragon Taming                 2  31819.051946
```

Now let's see the data in a plot.

```
fig, ax = plt.subplots(figsize=(8, 6))

gb = df.groupby('major')
for major, group in gb:
    ax.scatter(group['years_after_grad'], group['income'], label=major, alpha=0.6)

ax.legend(title='Major', frameon=False)
ax.set(xlabel='years after graduation',
       ylabel='income',
       xticks=np.arange(0, 21, 5)
       );
```



The model we will use is

$$y = \underbrace{X\beta}_{\text{fixed effects}} + \underbrace{Zb}_{\text{random effects}} + \underbrace{\varepsilon}_{\text{residuals}}$$

The only new term here is Zb, the random effects, where Z is the design matrix for the random effects and b is the vector of random effects coefficients. We will discuss that a bit later. Let's start with the fixed effects part:

$$X\beta = \beta_0 + \beta_1 \cdot \text{years} + \beta_2 \cdot \text{major} + \beta_3 \cdot (\text{years} \cdot \text{major})$$

The "years" variable is continuous, while the "major" variable is categorical. How to include categorical variables in a linear regression model? We can use dummy coding, where we create binary variables for each category of the categorical variable (except one category, which serves as the reference group). In our case, we have three majors: Juggling, Magic, and Dragon Taming. Let's use "Juggling" as the reference group. We can create two dummy variables that function as toggles.

- `major_Magic`: 1 if the major is Magic, 0 otherwise

- `major_DragonTaming`: 1 if the major is Dragon Taming, 0 otherwise

16.2 visualizing categories as toggles

In the equation above, we have only one parameter for "major" (β_2), and only one parameter for the interaction terms (β_3). In reality we have more, see:

$$\begin{aligned}\text{income} = \beta_0 &+ \beta_1 \cdot \text{years} \\ &+ \beta_2 \cdot \text{major\_Magic} + \beta_3 \cdot \text{major\_DragonTaming} \\ &+ \beta_4 \cdot (\text{years} \cdot \text{major\_Magic}) + \beta_5 \cdot (\text{years} \cdot \text{major\_DragonTaming}) \\ &+ \epsilon\end{aligned}$$

The first line represents the linear relationship between income and education of the reference group (Juggling). The second line adds the effects on the intercept of having studied Magic or Dragon Taming instead, and the third line adds the the effects on the slope of these two majors.

Let's see for a few data points how this works. Below, dummy variables represent the pair (major_Magic, major_DragonTaming).

years_after_grad	major	Dummy variables	income
3	Juggling	(0, 0)	37183.72
5	Magic	(1, 0)	35101.07
7	Dragon Taming	(0, 1)	27179.77
10	Juggling	(0, 0)	26366.80
12	Magic	(1, 0)	26101.53
16	Dragon Taming	(0, 1)	39252.76

The design matrix X would look like this:

$$X = \begin{matrix} \beta_0 & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 \end{matrix} \begin{pmatrix} 1 & 3 & 0 & 0 & 0 & 0 \\ 1 & 5 & 1 & 0 & 5 & 0 \\ 1 & 7 & 0 & 1 & 0 & 7 \\ 1 & 10 & 0 & 0 & 0 & 0 \\ 1 & 12 & 1 & 0 & 12 & 0 \\ 1 & 16 & 0 & 1 & 0 & 16 \end{pmatrix} .$$

The betas above the matrix are there just to label the columns, they are not really part of the matrix. The 3rd and 4th columns are the dummy variables for the majors, and the 5th and 6th columns are the interaction terms between education and the majors.

If we were not interested in the random effects, we could stop here, and just use the ordinary least squares (OLS) method already discussed to estimate the coefficients β.

16.3 random effects

The name "mixed effects" comes from the fact that we have both fixed effects and random effects.

Conceptually, the random effects function in a very similar way to the fixed effects. Instead of a small number of categories, now each person in the dataset is a category. In our example we have 90 different people represented in the dataset, so the quantity Z in Zb is the design matrix for the random effects, which is a matrix with 90 columns, one for each person, and as many rows as there are data points in the dataset. Each row has a 1 in the column corresponding to the person, and 0s elsewhere. The vector b is a vector of random effects coefficients, one for each person.

16.4 implementation

We can use `statsmodels` function `smf.mixedlm` to do everything for us. We just need to specify the formula, which includes the interaction term, and the data.

If you don't mind which category is the reference group, you can skip the cell below. If you want to make sure a give one is the reference group (Juggling in our case), then you should run it.

```
from pandas.api.types import CategoricalDtype
# define the desired order: Juggling as reference
major_order = CategoricalDtype(categories=["Juggling", "Magic", "Dragon Taming"], ordered=True)
df["major"] = df["major"].astype(major_order)
```

The syntax is fairly economic. The formula

`income ~ years_after_grad * major`

specifies a linear model where both the baseline income (intercept) and the effect of time since graduation (slope) can vary by major. The * operator includes both the main effects (years after graduation and major) and their interaction, allowing the model to fit a different intercept and slope for each major.

In the line

`model = smf.mixedlm(formula, data=df, groups=df["person"])`

the `groups` argument specifies that the random effects are associated with the "person" variable, meaning that each person can have their own random intercept.

```
# formula with interaction
formula = "income ~ years_after_grad * major"

# fit mixed model with random intercept for person
model = smf.mixedlm(formula, data=df, groups=df["person"])
result = model.fit()
```

Let's see the results

```
print(result.summary())
```

```
                          Mixed Linear Model Regression Results
=============================================================================================
Model:                          MixedLM            Dependent Variable:            income
```

```
No. Observations:              239                Method:                           REML
No. Groups:                    90                 Scale:                            10690821.7105
Min. group size:               1                  Log-Likelihood:                   -2327.5068
Max. group size:               4                  Converged:                        Yes
Mean group size:               2.7
-------------------------------------------------------------------------------------------------
                                          Coef.     Std.Err.   z    P>|z|   [0.025    0.975]
-------------------------------------------------------------------------------------------------
Intercept                                25206.095 1349.760 18.675 0.000 22560.615 27851.575
major[T.Magic]                           -2999.754 1995.748 -1.503 0.133 -6911.348   911.840
major[T.Dragon Taming]                    5579.198 1954.661  2.854 0.004  1748.133  9410.263
years_after_grad                           723.745   72.028 10.048 0.000   582.573   864.917
years_after_grad:major[T.Magic]            635.180  109.599  5.795 0.000   420.370   849.989
years_after_grad:major[T.Dragon Taming]   -295.862  106.315 -2.783 0.005  -504.235   -87.488
Group Var                              33814137.626 2268.953
=================================================================================================
```

16.5 interpreting the results

To interpret the coefficients, start with the reference group, which in this model is someone who studied Juggling. Their predicted income is:

$$\text{income} = 25206.10 + 723.75 \times \text{years}$$

Now, for a person who studied Magic, the model **adjusts** both the intercept and the slope:

Intercept shift: -2999.75 Slope shift: +635.18 So for Magic, the predicted income becomes:

$$\begin{aligned}\text{income} &= (25206.10 - 2999.75) + (723.75 + 635.18) \times \text{years} \\ &= 22206.35 + 1358.93 \times \text{years}\end{aligned}$$

This means that compared to Jugglers, Magicians start with a lower baseline salary, but their income grows much faster with each year after graduation.

The `Coef.` column shows the estimated value of each parameter (e.g., intercepts, slopes, interactions). The `Std.Err.` column reports the standard error of the estimate, reflecting its uncertainty. The `z` column is the test statistic (estimate divided by standard error), and `P>|z|` gives the p-value, which helps assess whether the effect is statistically significant. The final two columns, `[0.025 and 0.975]`, show the 95% confidence interval for the coefficient — if this interval does not include zero, the effect is likely meaningful.

The line labeled `Group Var` shows the estimated variance of the random intercepts — in this case, variation in baseline income between individuals. The second number reported is the standard error associated with this estimate, which indicates how much uncertainty there is in the estimate of the variance.

If you like, you can print out all the variances for the random effects. They are not explicity shown in the summary, but you can access them through the model's `random_effects` attribute:

```
result.random_effects
```

Finally, the model as is does not include random slopes, meaning that the effect of years after graduation is assumed to be the same for all individuals. If you want to allow for different slopes for each individual, you can modify the model to include random slopes as well. This would require changing the formula and the `groups` argument accordingly. Also, `result.random_effects` will then contain not only the random intercepts, but also the random slopes for each individual.

```
model = smf.mixedlm(
    "income ~ years_after_grad * major",
    data=df,
    groups=df["person"],
    re_formula="~years_after_grad"
)
result = model.fit()
print(result.summary())
```

```
/Users/yairmau/miniforge3/envs/olympus/lib/python3.11/site-packages/statsmodels/base/model.py:6
  warnings.warn("Maximum Likelihood optimization failed to "
/Users/yairmau/miniforge3/envs/olympus/lib/python3.11/site-packages/statsmodels/regression/mixe
  warnings.warn(
```

Mixed Linear Model Regression Results

Model:	MixedLM	Dependent Variable:	income
No. Observations:	239	Method:	REML
No. Groups:	90	Scale:	10125672.1682
Min. group size:	1	Log-Likelihood:	-2323.7559
Max. group size:	4	Converged:	Yes
Mean group size:	2.7		

	Coef.	Std.Err.	z	P>|z|	[0.025	0.975]
Intercept	25133.841	1208.050	20.805	0.000	22766.106	27501.576
major[T.Magic]	-2805.540	1811.051	-1.549	0.121	-6355.135	744.055
major[T.Dragon Taming]	5980.367	1767.166	3.384	0.001	2516.786	9443.949
years_after_grad	731.399	84.211	8.685	0.000	566.349	896.450
years_after_grad:major[T.Magic]	611.065	126.072	4.847	0.000	363.969	858.161
years_after_grad:major[T.Dragon Taming]	-329.530	122.977	-2.680	0.007	-570.561	-88.498
Group Var	22392488.656	1835.422				
Group x years_after_grad Cov	90328.607	75.664				
years_after_grad Var	39074.487	7.401				

16.6 back to OLS

If you went this far, and now realized you don't care about random effects, you can just use the `statsmodels` function `smf.ols` to fit an ordinary least squares regression model. The syntax is similar, but without the `groups` argument.

```
import statsmodels.formula.api as smf

# formula with main effects and interaction
formula = "income ~ years_after_grad * major"
```

```python
# fit the model with OLS (no random effects)
ols_model = smf.ols(formula, data=df)
ols_result = ols_model.fit()

# print summary
print(ols_result.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 income   R-squared:                       0.455
Model:                            OLS   Adj. R-squared:                  0.443
Method:                 Least Squares   F-statistic:                     38.85
Date:                Tue, 24 Jun 2025   Prob (F-statistic):           6.27e-29
Time:                        16:16:38   Log-Likelihood:                -2437.0
No. Observations:                 239   AIC:                             4886.
Df Residuals:                     233   BIC:                             4907.
Df Model:                           5
Covariance Type:            nonrobust
================================================================================================
                                           coef    std err          t      P>|t|      [0.025
------------------------------------------------------------------------------------------------
Intercept                                2.486e+04   1450.267     17.141      0.000     2.2e+04
major[T.Magic]                          -4402.0846   2281.475     -1.929      0.055   -8897.041
major[T.Dragon Taming]                   7696.8705   2167.061      3.552      0.000    3427.332
years_after_grad                          778.4674    123.280      6.315      0.000     535.582
years_after_grad:major[T.Magic]           758.4393    185.397      4.091      0.000     393.170
years_after_grad:major[T.Dragon Taming]  -510.1096    183.456     -2.781      0.006    -871.553
==============================================================================
Omnibus:                        2.143   Durbin-Watson:                   1.088
Prob(Omnibus):                  0.343   Jarque-Bera (JB):                2.132
Skew:                           0.176   Prob(JB):                        0.344
Kurtosis:                       2.699   Cond. No.                         93.1
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

17 logistic regression

17.1 question

We are given a list of heights for men and women. Given one more data point (180 cm), could we assign a probability that it belongs to either class?

17.2 discriminative model

The idea behind the logistic regression is to find a boundary between our two classes (here men and women). The logistic regression models the probability of a class given a data point, i.e. $P(y|x)$. We can use the logistic function to model this probability:

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

where y is the class (0=women, 1=men), x is the data point (height), and β_0 and β_1 are the parameters of the model.

Our goal is to find the best s-shaped curve that describes the data. This is done by finding the parameters β_0 and β_1 that maximize the likelihood of the data.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from sklearn.linear_model import LogisticRegression
from scipy.stats import norm
```

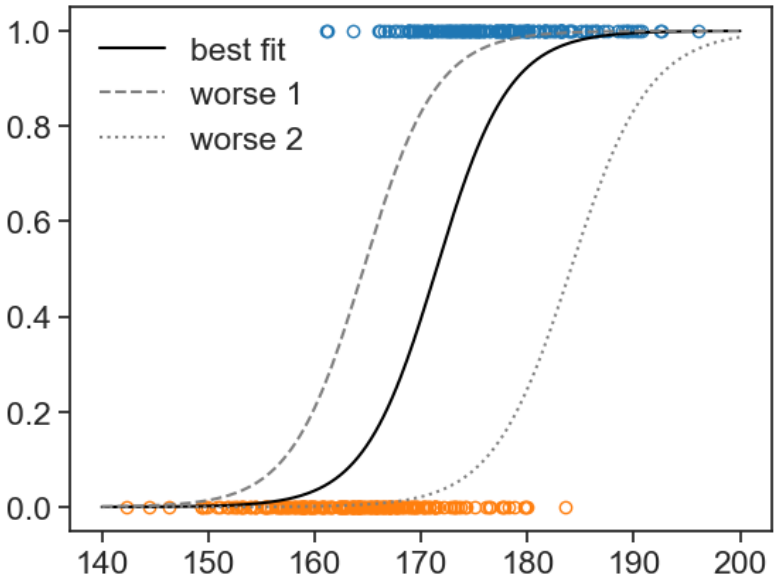
```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
df_girls = pd.read_csv('../archive/data/height/girls_height_stats.csv', index_col=0)
age = 20.0
mu_boys = df_boys.loc[age, 'mu']
mu_girls = df_girls.loc[age, 'mu']
sigma_boys = df_boys.loc[age, 'sigma']
sigma_girls = df_girls.loc[age, 'sigma']

N_boys = 150
N_girls = 200
np.random.seed(314)  # set scipy seed for reproducibility
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
# pandas dataframe with the two samples in it
df = pd.DataFrame({
    'height (cm)': np.concatenate([sample_boys, sample_girls]),
    'sex': ['M'] * N_boys + ['F'] * N_girls
})
df = df.sample(frac=1, random_state=314).reset_index(drop=True)
df
```

	height (cm)	sex
0	178.558416	M
1	173.334306	M
2	183.084154	M
3	178.236047	F
4	175.868642	M
...	...	...
345	177.387837	M
346	157.122325	F
347	166.891746	F
348	181.090312	M
349	171.479631	M

```
X = df['height (cm)'].values.reshape(-1, 1)
y = df['sex'].map({'M': 1, 'F': 0}).values
log_reg_2 = LogisticRegression(penalty=None, solver = 'newton-cg', max_iter= 150).fit(X,y)
beta1 = log_reg_2.coef_[0][0]
beta0 = log_reg_2.intercept_[0]
```

```
def logistic_function(x, beta0, beta1):
    return 1 / (1 + np.exp(-(beta0 + beta1 * x)))
```

```
fig, ax = plt.subplots()
ax.plot(sample_girls, np.zeros_like(sample_girls),
        linestyle='None', marker='o',
        markerfacecolor='none', markeredgecolor='tab:orange')
ax.plot(sample_boys, np.ones_like(sample_boys),
        linestyle='None', marker='o',
        markerfacecolor='none', markeredgecolor='tab:blue')

x_array = np.linspace(140, 200, 300).reshape(-1, 1)
y_proba = log_reg_2.predict_proba(x_array)[:, 1]
# ax.plot(x_array, y_proba, color='black')
ax.plot(x_array, logistic_function(x_array, beta0, beta1), color='black', label="best fit")
ax.plot(x_array, logistic_function(x_array, beta0+2, beta1), color='gray', linestyle='--', labe
ax.plot(x_array, logistic_function(x_array, beta0, beta1-0.02), color='gray', linestyle=':', la
ax.legend(frameon=False)
```



17.3 likelihood

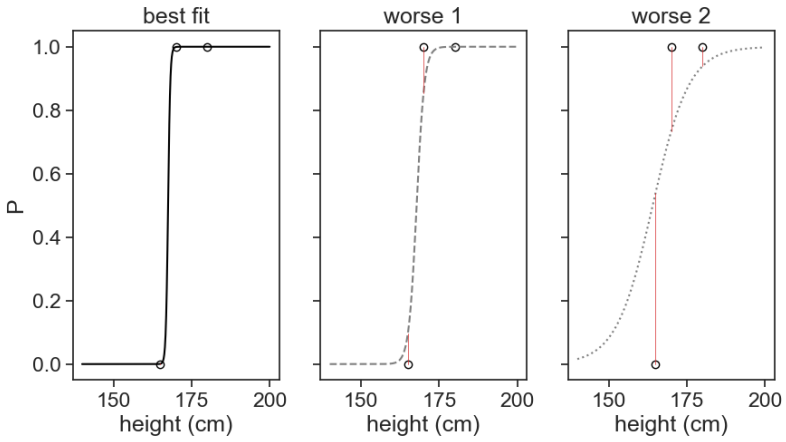
How do we know the parameters of the **best** s-shaped curve?
Let's pretend we have only three data points:

- Man, 180 cm. Data point $(180, 1)$.
- Man, 170 cm. Data point $(170, 1)$.
- Woman, 165 cm. Data point $(165, 0)$.

```
x3 = np.array([180.0, 170.0, 165.0])
y3 = np.array([1, 1, 0])
log_reg_3 = LogisticRegression(penalty=None, solver = 'newton-cg', max_iter= 150).fit(x3.reshap
beta1 = log_reg_3.coef_[0][0]
beta0 = log_reg_3.intercept_[0]
```

```
fig, ax = plt.subplots(1, 3, figsize=(10, 5), sharex=True, sharey=True)
x_array = np.linspace(140, 200, 300)

ya = logistic_function(x_array, beta0, beta1)
yb = logistic_function(x_array, beta0+344, beta1*0.28)
yc = logistic_function(x_array, beta0+450, beta1*0.06)
yhat3a = logistic_function(x3, beta0, beta1)
yhat3b = logistic_function(x3, beta0+344, beta1*0.28)
yhat3c = logistic_function(x3, beta0+450, beta1*0.06)

for axi in ax:
    axi.plot(x3, y3,
             linestyle='None', marker='o',
             markerfacecolor='none', markeredgecolor='black')

ax[0].plot(x_array, ya, color='black')
ax[1].plot(x_array, yb, color='gray', linestyle='--')
ax[2].plot(x_array, yc, color='gray', linestyle=':')

for i in range(len(x3)):
    ax[0].plot([x3[i], x3[i]], [y3[i], yhat3a[i]], color='tab:red', linestyle='-', lw=0.5)
    ax[1].plot([x3[i], x3[i]], [y3[i], yhat3b[i]], color='tab:red', linestyle='-', lw=0.5)
    ax[2].plot([x3[i], x3[i]], [y3[i], yhat3c[i]], color='tab:red', linestyle='-', lw=0.5)

ax[0].set(xlabel="height (cm)",
```

```
          ylabel="P",
          title="best fit")
ax[1].set(xlabel="height (cm)",
          title="worse 1")
ax[2].set(xlabel="height (cm)",
            title="worse 2")
```



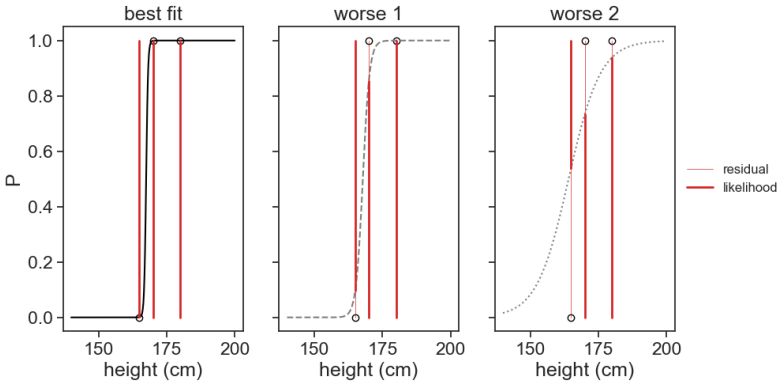
As usual, our task in performing the regression is to find the parameters β_0 and β_1 that minimize the distance between the model and the data (the residual). See the figure above, we plotted the same three data points, and in each panel we see a different s-shaped curve (black) and the distance between the model and the data (red lines).

[Note: this time, because we have only three data points, the best fit gave us an extremely sharp logistic function, that neatly discriminates between the data points. In the first example, the function was much more "shallow", because of the overlap between the Men and Women datasets.]

In the logistic regression, instead of **minimizing the residual**, we **maximize the likelihood** of the data given the model parameters. The likelihood is the complement of the residual, see the thick red bars in the figure below.

```
fig, ax = plt.subplots(1, 3, figsize=(10, 5), sharex=True, sharey=True)

for axi in ax:
```

```
    axi.plot(x3, y3,
            linestyle='None', marker='o',
            markerfacecolor='none', markeredgecolor='black')

ax[0].plot(x_array, ya, color='black')
ax[1].plot(x_array, yb, color='gray', linestyle='--')
ax[2].plot(x_array, yc, color='gray', linestyle=':')

for i in range(len(x3)):
    ax[0].plot([x3[i], x3[i]], [y3[i], yhat3a[i]], color='tab:red', linestyle='-', lw=0.5)
    ax[0].plot([x3[i], x3[i]], [not bool(y3[i]), yhat3a[i]], color='tab:red', linestyle='-', lw
    ax[1].plot([x3[i], x3[i]], [y3[i], yhat3b[i]], color='tab:red', linestyle='-', lw=0.5)
    ax[1].plot([x3[i], x3[i]], [not bool(y3[i]), yhat3b[i]], color='tab:red', linestyle='-', lw
    ax[2].plot([x3[i], x3[i]], [y3[i], yhat3c[i]], color='tab:red', linestyle='-', lw=0.5)
    ax[2].plot([x3[i], x3[i]], [not bool(y3[i]), yhat3c[i]], color='tab:red', linestyle='-', lw

ax[2].plot([], [], color='tab:red', linestyle='-', lw=0.5, label="residual")
ax[2].plot([], [], color='tab:red', linestyle='-', lw=2, label="likelihood")

ax[0].set(xlabel="height (cm)",
          ylabel="P",
          title="best fit")
ax[1].set(xlabel="height (cm)",
          title="worse 1")
ax[2].set(xlabel="height (cm)",
           title="worse 2")
ax[2].legend(frameon=False, loc='center left', bbox_to_anchor=(1, 0.5), fontsize=12)
```



What is the probability that we would measure the observed

data, given the model parameters? For a single data point (height, class), the length of the red bars can be described by the formula below:

$$L(y_i|P_i) = P_i^{y_i}(1-P_i)^{1-y_i}.$$

For instance, if the data point corresponds to a man ($y_i = 1$), we have $L = P_i$. The likelihood (thick red bars) for men is just the value of the logistic function for that value of x. For women ($y_i = 0$), we have $L = 1 - P_i$. The likelihood for women is just the complement (one minus) of the logistic function.

Assuming that each data point is independent, the likelihood of the entire dataset is the product of the likelihoods of each data point:

$$L(\beta_0, \beta_1) = \prod_{i=1}^{N} P_i^{y_i}(1-P_i)^{1-y_i}.$$

In the example below, the likelihood of the entire dataset for each panel is as follows:

```
La = 1.0
Lb = 1.0
Lc = 1.0
for i in range(len(x3)):
    La = La * yhat3a[i]**y3[i] * (1-yhat3a[i])**(1-y3[i])
    Lb = Lb * yhat3b[i]**y3[i] * (1-yhat3b[i])**(1-y3[i])
    Lc = Lc * yhat3c[i]**y3[i] * (1-yhat3c[i])**(1-y3[i])
print("Likelihood for best fit: ", La)
print("Likelihood for worse 1: ", Lb)
print("Likelihood for worse 2: ", Lc)
```

```
Likelihood for best fit:  0.9984099891897481
Likelihood for worse 1:  0.7711068593851899
Likelihood for worse 2:  0.3173593316343797
```

As we increase the number of data points, the likelihood becomes very small (because we are multiplying many numbers between 0 and 1). To avoid numerical issues, we usually work with the log-likelihood.

17.4 log-likelihood

The log-likelihood is the logarithm of the likelihood:

$$\ell(\beta_0, \beta_1) = \log L(\beta_0, \beta_1) = \sum_{i=1}^{N} P_i^{y_i} (1 - P_i)^{1-y_i}$$

Using the properties of logarithms, we can rewrite the log-likelihood as follows:

$$\ell(\beta_0, \beta_1) = \sum_{i=1}^{N} \left(y_i \log P_i + (1 - y_i) \log(1 - P_i) \right).$$

17.5 binary cross-entropy, or log loss

We can use gradient descent to find the parameters that maximize the log-likelihood. Most implementations of gradient descent are designed to minimize a cost function. Therefore, instead of maximizing the log-likelihood, we can minimize the negative log-likelihood:

$$J(\beta_0, \beta_1) = -\ell(\beta_0, \beta_1) = -\sum_{i=1}^{N} \left(y_i \log P_i + (1 - y_i) \log(1 - P_i) \right).$$

This cost function is also known as binary cross-entropy or log loss.

It turns out that taking the log of the likelihood is very convenient. What was before only a trick to avoid numerical issues, now has a nice interpretation. The cross-entropy can be thought of as a measure of "surprise". The more the model is surprised by the data, the higher the cross-entropy, and the poorer the fit. The less surprised the model is by the data, the lower the cross-entropy, and the better the fit.

17.6 wrapping up

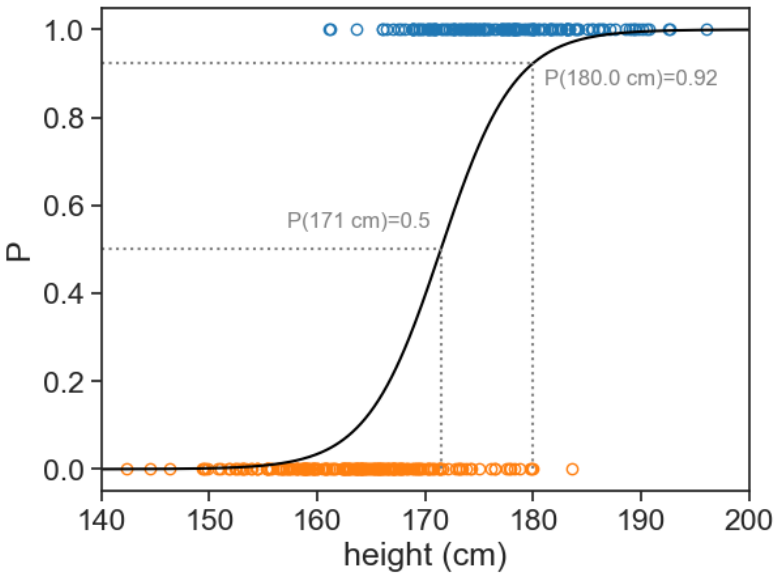
From the provided data:

- What is the probability that a person whose height is 180 cm is a man?
- If we had to choose one height to discriminate between men and women, what would it be?

Let's run the code for the logistic regression again:

```
X = df['height (cm)'].values.reshape(-1, 1)
y = df['sex'].map({'M': 1, 'F': 0}).values
log_reg_2 = LogisticRegression(penalty=None, solver = 'newton-cg', max_iter= 150).fit(X,y)
beta1 = log_reg_2.coef_[0][0]
beta0 = log_reg_2.intercept_[0]

#| code-summary: "plot "
fig, ax = plt.subplots()
ax.plot(sample_girls, np.zeros_like(sample_girls),
        linestyle='None', marker='o',
        markerfacecolor='none', markeredgecolor='tab:orange')
ax.plot(sample_boys, np.ones_like(sample_boys),
        linestyle='None', marker='o',
        markerfacecolor='none', markeredgecolor='tab:blue')

x_array = np.linspace(140, 200, 300).reshape(-1, 1)
y_proba = log_reg_2.predict_proba(x_array)[:, 1]
# ax.plot(x_array, y_proba, color='black')
ax.plot(x_array, logistic_function(x_array, beta0, beta1), color='black', label="best fit")

h = 180.0
p180 = log_reg_2.predict_proba(np.array([[h]]))[0, 1]
ax.plot([h, h], [0, p180], color='gray', linestyle=':')
ax.plot([np.min(x_array), h], [p180, p180], color='gray', linestyle=':')
ax.text(h+1, p180-0.05, f"P({h} cm)={p180:.2f}", color='gray', fontsize=12)

p_50percent = -beta0 / beta1
ax.plot([p_50percent, p_50percent], [0, 0.5], color='gray', linestyle=':')
ax.plot([np.min(x_array), p_50percent], [0.5, 0.5], color='gray', linestyle=':')
ax.text(p_50percent-1, 0.5+0.05, f"P({p_50percent:.0f} cm)=0.5", color='gray', fontsize=12, ha=
```

```
ax.set(xlim=(140, 200),
       xlabel="height (cm)",
       ylabel="P",)
```



Answers:

- The probability that a person 180 cm tall is a man is 92%.
- The height that best discriminates between men (above) and women (below) is 171 cm.

This last result follows directly from:

$$P(x) = \frac{1}{1 + \exp[-(\beta_0 + \beta_1 x)]} = \frac{1}{2}$$

$$\text{therefore}$$

$$1 + \exp[-(\beta_0 + \beta_1 x)] = 2$$

$$\exp[-(\beta_0 + \beta_1 x)] = 1$$

$$-(\beta_0 + \beta_1 x) = 0$$

$$x = -\frac{\beta_0}{\beta_1}$$

Compare this result with the one we obtained with the parametric generative model discussed in the Bayes' theorem section.

If you want to see a nice tutorial, see Dr. Roi Yehoshua's "Mastering Logistic Regression".

17.7 connection to neural networks

The logistic regression can be understood as a single-layer perceptron neural network model. This is to say, a neural network with no hidden layers, and a single output neuron that uses the logistic (sigmoid) activation function.

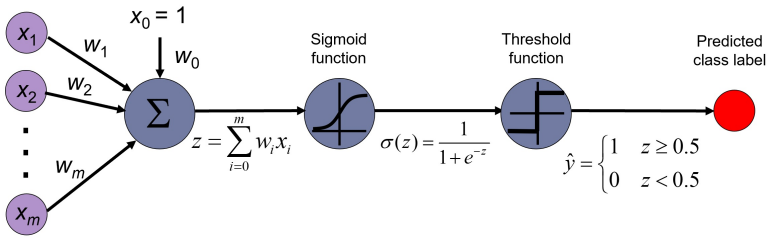


Figure 17.1: "source: https://towardsdatascience.com/mastering-logistic-regression-3e502686f0ae/"

18 logistic 2d

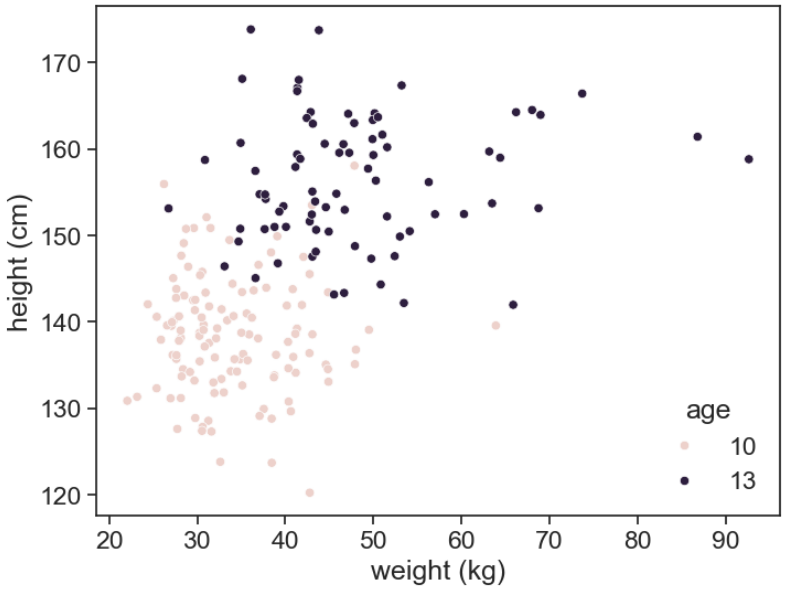
The figure below represents the distribution of height and weight for boys aged 10 and 13 years old.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from sklearn.linear_model import LogisticRegression
from scipy.stats import norm
```

```
def scipy_power_normal_draw_random(N, age, sex):
    # Load LMS parameters from the CSV file
    df = pd.read_csv('../archive/data/weight/wtage.csv')
    agemos = age * 12 + 0.5
    df = df[(df['Agemos'] == agemos) & (df['Sex'] == sex)]
    L = df['L'].values[0]
    M = df['M'].values[0]
    S = df['S'].values[0]

    # draw random z from standard normal distribution
    z = np.random.normal(0, 1, N)
    # transform z to w
    w = M * (1 + L * S * z)**(1 / L)

    return w

# 10-year-old boys
N_10 = 120
w_10 = scipy_power_normal_draw_random(N_10, 10, 1)
df_height_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
mu_boys_10 = df_height_boys.loc[10.0, 'mu']
sigma_boys_10 = df_height_boys.loc[10.0, 'sigma']
```

```
h_10 = norm.rvs(size=N_10, loc=mu_boys_10, scale=sigma_boys_10)

# 15-year-old boys
N_13 = 80
w_13 = scipy_power_normal_draw_random(N_13, 13, 1)
mu_boys_13 = df_height_boys.loc[13.0, 'mu']
sigma_boys_13 = df_height_boys.loc[13.0, 'sigma']
h_13 = norm.rvs(size=N_13, loc=mu_boys_13, scale=sigma_boys_13)

# Combine data into a single DataFrame
df = pd.DataFrame({
    'weight': np.concatenate([w_10, w_13]),
    'height': np.concatenate([h_10, h_13]),
    'age': [10] * N_10 + [13] * N_13
})
```

```
fig, ax = plt.subplots(figsize=(8, 6))
sns.scatterplot(data=df, x='weight', y='height', hue='age', ax=ax)
ax.legend(title='age', loc='lower right', frameon=False)
ax.set(xlabel='weight (kg)',
       ylabel='height (cm)')
```



Our job is to find a decision boundary that separates the two

classes. Fundamentally, this is the same as the logistic regression we saw in the previous chapter, but now we have two features instead of one. There are two ways to equivalent ways to describe this, the statisics and the machine learning way.

18.1 statistics

We call our two features, height and weight, x_1 and x_2. We can write the logistic regression model as

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

where p is the probability of being 13 years old. The left hand side is called the log-odds or logit. The right hand side is a linear combination of the features.

This is, of course, equivalent to the expression with the sigmoid function:

$$p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$

Speaking a "statistics language", this linear relationship is expressed by writing everything in matrix form:

$$z = X\beta,$$

where X is the design matrix, β is the vector of coefficients, and z is the linear predictor.

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix} = \begin{pmatrix} \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} \\ \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} \\ \vdots \\ \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} \end{pmatrix}$$

18.2 machine learning

In machine learning, we often call the intercept term the bias, and we call the coefficients weights. We can write the linear predictor as

$$z = w_1 x_1 + w_2 x_2 + b$$

where w_1 and w_2 are the weights, and b is the bias. In matrix form:

$$z = w^T x + b,$$

which expands to

$$\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{pmatrix} = \begin{pmatrix} w_1 & w_2 \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ \vdots & \vdots \\ x_{n1} & x_{n2} \end{pmatrix} + \begin{pmatrix} b \\ b \\ \vdots \\ b \end{pmatrix} = \begin{pmatrix} w_1 x_{11} + w_2 x_{12} + b \\ w_1 x_{21} + w_2 x_{22} + b \\ \vdots \\ w_1 x_{n1} + w_2 x_{n2} + b \end{pmatrix}$$

This is the same as the statistics formulation, just with different names for the parameters.

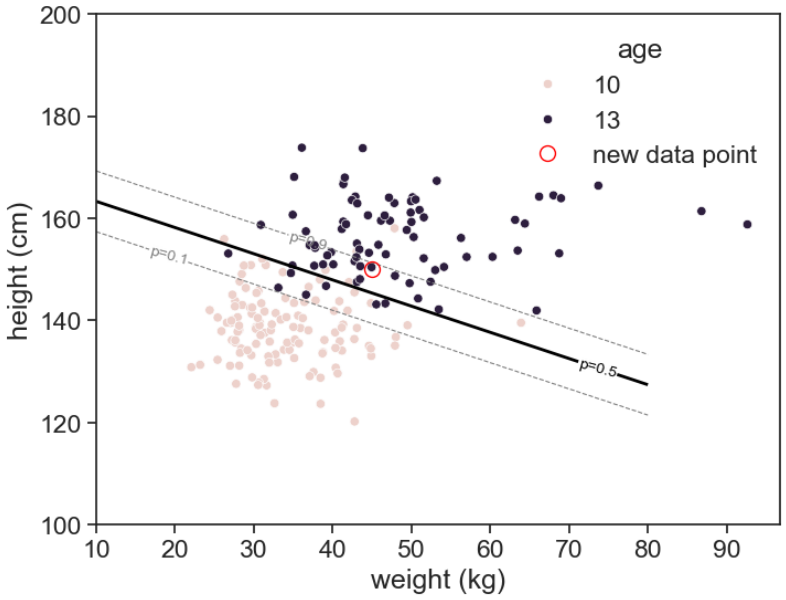
18.3 solving

I boy from either 7th grade (13 years old) or 5th grade (10 years old) is randomly selected. Given his height (150 cm) and weight (45 kg), we want to predict his age group.

```
fig, ax = plt.subplots(figsize=(8, 6))
X = df[['weight', 'height']]
y = df['age']
model = LogisticRegression()
model.fit(X, y)
xx, yy = np.meshgrid(np.linspace(10, 80, 100), np.linspace(100, 200, 100))
grid_points = np.c_[xx.ravel(), yy.ravel()]
predict_df = pd.DataFrame(grid_points, columns=['weight', 'height'])
Z = model.predict(predict_df)
Z = Z.reshape(xx.shape)
Z_prob = model.predict_proba(predict_df)[:, 1]
Z = Z_prob.reshape(xx.shape)
# contour_fill = ax.contourf(xx, yy, Z, levels=np.arange(0, 1.1, 0.2), cmap='RdBu_r', alpha=0.8
```

```
cont = ax.contour(xx, yy, Z, levels=[0.1, 0.5, 0.9], colors=['gray', 'black', 'gray'], linesty
ax.clabel(cont, fmt='p=%1.1f', inline=True, fontsize=10)

sns.scatterplot(data=df, x='weight', y='height', hue='age', ax=ax)
ax.set(xlabel='weight (kg)',
       ylabel='height (cm)');

h1 = 150
w1 = 45
df1 = pd.DataFrame([[w1, h1]], columns=['weight', 'height'])
ax.plot([w1], [h1], ls='None', marker='o', markersize=10, markerfacecolor='None', markeredgeco
ax.legend(title='age', loc='upper right', frameon=False)
p1 = model.predict_proba(df1)[0, 1]
print(f"Predicted probability of being 13 years old: {p1:.3f}")
```

```
Predicted probability of being 13 years old: 0.848
```



The thick line in the figure above is the decision boundary, where the probability of being 13 years old is 0.5. The equation of the line is

$$0 = w_1 x_1 + w_2 x_2 + b,$$

were feature x_1 is weight and feature x_2 is height. The weights w_1, w_2 and bias b are

```
model.coef_, model.intercept_
```

```
(array([[0.19047627, 0.37131303]]), array([-62.54777199]))
```

Part VI

correlation

19 correlation

19.1 variance

To understand correlation, we need to start with variance. Let's say X is a random variable with mean μ. The variance of X, denoted $\text{var}(X) = \sigma^2$, is defined as the expected value of the squared deviation from the mean:

$$\sigma^2 = \text{var}(X) = E[(X - \mu)^2] = \frac{1}{N}\sum_{i=1}^{N}(X_i - \mu)^2.$$

I should point out that the formula above is for the *population* variance. If we are working with a *sample*, we would use $N-1$ in the denominator instead of N to get an unbiased estimate of the population variance. Also, the mean and variance for the sample are denoted $\bar{X}$ and s^2 respectively. In any case, let's continue with the population variance for simplicity.

In simple words, the variance measures how much the values of X deviate from the mean μ. A high variance indicates that the data points are spread out over a wider range of values, while a low variance indicates that they are closer to the mean.

19.2 covariance

Now, let's consider two random variables, X and Y, with means μ_X and μ_Y. The covariance between X and Y, denoted $\text{cov}(X, Y)$, is defined as:

$$\text{cov}(X, Y) = E[(X - \mu_X)(Y - \mu_Y)] = \frac{1}{N}\sum_{i=1}^{N}(X_i - \mu_X)(Y_i - \mu_Y).$$

A high covariance indicates that when X is above its mean, Y tends to be above its mean as well (and vice versa). A low (or negative) covariance indicates that when X is above its mean, Y tends to be below its mean.

19.3 correlation

The covariance can be any value, making it difficult to interpret. To standardize the measure, we use the correlation coefficient, denoted ρ (for population) or r (for sample). The correlation coefficient is defined as:

$$\begin{aligned}\rho_{X,Y} &= \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} \\ &= E\left[\frac{(X-\mu_X)}{\sigma_X}\frac{(Y-\mu_Y)}{\sigma_Y}\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\frac{X_i-\mu_X}{\sigma_x}\frac{Y_i-\mu_Y}{\sigma_Y}.\end{aligned}$$

where σ_X and σ_Y are the standard deviations of X and Y, respectively.

Something becomes clear now. If we calculate the correlation of X with itself, we get:

$$\rho_{X,X} = \frac{\text{cov}(X,X)}{\sigma_X \sigma_X} = \frac{\text{var}(X)}{\sigma_X^2} = 1.$$

The highest possible correlation is 1, which indicates a perfect positive linear relationship between the two variables. The lowest possible correlation is -1, which indicates a perfect negative linear relationship. A correlation of 0 indicates no linear relationship between the variables.

19.4 Pearson correlation coefficient

When we say "correlation", we usually mean the Pearson correlation coefficient, which is the formula given above. Pearson invented this, so it's named after him. There are other types of correlation coefficients, such as Spearman's rank correlation coefficient and Kendall's tau coefficient, which are used for non-parametric data or ordinal data.

19.5 covariance of z-scored variables

Notice that the correlation formula can be interpreted as the covariance of the z-scored variables. The z-score of a variable X is defined as:

$$Z_X = \frac{X - \mu_X}{\sigma_X}$$

Thus, the correlation can be rewritten as:

$$\rho_{X,Y} = \text{cov}(Z_X, Z_Y) = \frac{1}{N} \sum_{i=1}^{N} Z_{X_i} Z_{Y_i}$$

It is quite easy to compute the correlation on the computer. If X and Y are two arrays, first we standardize (z-score) them, then we compute their dot product (sum of piecewise multiplication), and finally we divide by N (or $N - 1$ for sample correlation).

19.6 linearity

The Pearson correlation coefficient measures the strength and direction of a **linear** relationship between two variables. It does not capture non-linear relationships. For example, if $Y = X^2$, the correlation between X and Y may be low or even zero, despite a clear non-linear relationship. When we say "correlation", it is usually implicit that we are referring to linear correlation.

20 correlation and linear regression

The correlation coefficient is closely related to linear regression. We will see below a few instances of this relationship.

20.1 prelude: finding the intercept and slope

Let's derive the formulas for the intercept and slope of the regression line. We want to minimize the sum of squared residuals L:

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2, \tag{1}$$

where

$$\hat{y}_i = \beta_0 + \beta_1 x_i. \tag{2}$$

To find the optimal values of β_0 and β_1, we take the partial derivatives of L with respect to β_0 and β_1, set them to zero, and solve the resulting equations.

$$\frac{\partial L}{\partial \beta_0} = 0 \tag{3a}$$

$$\frac{\partial L}{\partial \beta_1} = 0 \tag{3b}$$

We already did that for a general case, but the calculation had the variables in vector/matrix form. Here we will do it for the simple case of one predictor variable, so that we can see the relationship with correlation more clearly.

20.1.1 intercept

Let's start with Eq. (3a), and substitute into it Eq. (2):

$$\frac{\partial L}{\partial \beta_0} = \frac{\partial}{\partial \beta_0}(y_i - \hat{y}_i)^2 \tag{4a}$$

$$= \frac{\partial}{\partial \beta_0}(y_i - \beta_0 - \beta_1 x_i)^2 \tag{4b}$$

$$= -2\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i) = 0 \tag{4c}$$

Eliminating the constant factor -2 and expanding the summation, we get:

$$\sum_{i=1}^{n} y_i - n\beta_0 - \beta_1 \sum_{i=1}^{n} x_i = 0 \tag{5}$$

We now divide by n and rearrange to isolate β_0:

$$\beta_0 = \bar{y} - \beta_1 \bar{x} \tag{6}$$

Note: we can rewrite equation (4c) as

$$\sum_{i=1}^{n}(y_i - \hat{y}_i) = \sum_{i=1}^{n} \text{residuals} = 0,$$

which is a nice thing to know.

20.1.2 slope

Now let's move on to Eq. (3b), and substitute the result of Eq. (6) into it:

$$\frac{\partial L}{\partial \beta_1} = \frac{\partial}{\partial \beta_1}(y_i - \hat{y}_i)^2 \tag{7a}$$

$$= \frac{\partial}{\partial \beta_1}(y_i - \beta_0 - \beta_1 x_i)^2 \tag{7b}$$

$$= -2\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)x_i = 0 \tag{7c}$$

$$= -2\sum_{i=1}^{n}(y_i - \bar{y} + \beta_1 \bar{x} - \beta_1 x_i)x_i = 0 \tag{7d}$$

Eliminating the constant factor 2 and expanding the summation, we get:

$$-\sum_{i=1}^{n} x_i y_i + \sum_{i=1}^{n} x_i \bar{y} - \beta_1 \sum_{i=1}^{n} x_i \bar{x} + \beta_1 \sum_{i=1}^{n} x_i^2 = 0 \tag{8}$$

Let's group the terms involving β_1 on one side and the rest on the other side:

$$\beta_1 \left(\sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i \bar{x} \right) = \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \bar{y} \tag{9}$$

Isolating β_1, we have:

$$\beta_1 = \frac{\sum x_i y_i - \sum x_i \bar{y}}{\sum x_i^2 - \sum x_i \bar{x}} = \frac{\text{numerator}}{\text{denominator}} \tag{10}$$

It's easier to interpret the numerator and denominator separately. To each we will add and subtract a term that will allow us to express them in simpler forms.

Numerator:

$$\text{numerator} = \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \bar{y} + \sum_{i=1}^{n} \bar{x} y_i - \sum_{i=1}^{n} \bar{x} y_i \tag{11}$$

We express the third term thus:

$$\text{third term} = \sum_{i=1}^{n} \bar{x} y_i = \bar{x} \sum_{i=1}^{n} y_i = n\bar{x}\bar{y} = \sum_{i=1}^{n} \bar{x}\bar{y} \qquad (12)$$

The numerator now becomes:

$$\text{numerator} = \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \bar{y} + \sum_{i=1}^{n} \bar{x}\bar{y} - \sum_{i=1}^{n} \bar{x} y_i \qquad (13a)$$

$$= \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \qquad (13b)$$

Now the denominator:

$$\text{denominator} = \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i \bar{x} + \sum_{i=1}^{n} x_i \bar{x} - \sum_{i=1}^{n} x_i \bar{x} \qquad (14)$$

We group the second and fourth terms, and express the third term thus:

$$\text{third term} = \sum_{i=1}^{n} x_i \bar{x} = \bar{x} \sum_{i=1}^{n} x_i = n\bar{x}^2 = \sum_{i=1}^{n} \bar{x}^2 \qquad (15)$$

The denominator now becomes:

$$\text{denominator} = \sum_{i=1}^{n} x_i^2 - 2 \sum_{i=1}^{n} x_i \bar{x} + \sum_{i=1}^{n} \bar{x}^2 \qquad (16a)$$

$$= \sum_{i=1}^{n} (x_i - \bar{x})^2 \qquad (16b)$$

Putting it all together, we have:

$$\beta_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \qquad (17)$$

20.2 slope and correlation

Let's divide both the numerator and denominator of Eq. (17) by $n-1$:

$$\beta_1 = \frac{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{18}$$

The numerator is the sample covariance $\text{cov}(X, Y)$, and the denominator is the sample variance $\text{var}(X)$:

$$\beta_1 = \frac{\text{cov}(X, Y)}{\text{var}(X)} \tag{19}$$

Now, we can express the covariance in terms of the correlation coefficient $\rho_{X,Y}$ and the standard deviations σ_X and σ_Y (see here):

$$\text{cov}(X, Y) = \rho_{X,Y}\sigma_X\sigma_Y \tag{20}$$

Substituting Eq. (20) into Eq. (19), we get:

$$\beta_1 = \frac{\rho_{X,Y}\sigma_X\sigma_Y}{\sigma_X^2} \tag{21}$$

And finally, we have:

$$\beta_1 = \rho_{X,Y}\frac{\sigma_Y}{\sigma_X} \tag{22}$$

This shows that the slope of the regression line is directly proportional to the correlation coefficient. A higher absolute value of the correlation coefficient indicates a steeper slope, while a lower absolute value indicates a flatter slope.

20.3 R^2 = square of the correlation coefficient r

Let's start by saying that the correlation coefficient is called ρ when referring to the population, and r when referring to a sample. I'm playing loose with this convention here, but I hope it's clear from the context.

Let's show now that the coefficient of determination R^2 is equal to the square of the correlation coefficient r.

We start with the definition of R^2:

$$R^2 = 1 - \frac{\text{SS}_{\text{Error}}}{\text{SS}_{\text{Total}}} = \frac{\text{SS}_{\text{Total}} + \text{SS}_{\text{Error}}}{\text{SS}_{\text{Total}}} = \frac{\text{SS}_{\text{Model}}}{\text{SS}_{\text{Total}}} \tag{23}$$

where we used the fact that $\text{SS}_{\text{Total}} = \text{SS}_{\text{Model}} + \text{SS}_{\text{Error}}$, already seen before.

We now substitute into Eq. (23) the definitions of SS_{Model} and SS_{Total}:

$$R^2 = \frac{\sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{24}$$

We now substitute into Eq. (24) the expression of $\hat{y}_i$ from Eq. (2):

$$R^2 = \frac{\sum_{i=1}^{n}(\beta_0 + \beta_1 x_i - \bar{y})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{25}$$

Now we substitute into Eq. (25) the expression of β_0 from Eq. (6):

$$R^2 = \frac{\sum_{i=1}^{n}\left(\bar{y} - \beta_1\bar{x} + \beta_1 x_i - \bar{y}\right)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = \frac{\sum_{i=1}^{n}\left(\beta_1(x_i - \bar{x})\right)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{26}$$

β_1 is a number, so we can take it out of the summation in the numerator:

$$R^2 = \frac{\beta_1^2 \sum_{i=1}^n (x_i - \bar{x})^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \tag{26b}$$

Now, let's substitute into Eq. (26) the expression of β_1 from Eq. (17):

$$R^2 = \frac{\left(\frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}\right)^2 \sum_{i=1}^n (x_i - \bar{x})^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \tag{27}$$

We can simplify Eq. (27) by canceling one instance of the denominator in the squared term with the factor outside the squared term:

$$R^2 = \frac{\left[\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})\right]^2}{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2} \tag{28}$$

Now, let's multiply both the numerator and denominator of Eq. (28) by $\frac{1}{(n-1)^2}$:

$$R^2 = \frac{\left[\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})\right]^2}{\left[\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2\right] \left[\frac{1}{n-1} \sum_{i=1}^n (y_i - \bar{y})^2\right]} \tag{29}$$

The numerator is the square of the sample covariance $\text{cov}(X, Y)$, and the denominator is the product of the sample variances $\text{var}(X) = \sigma_X^2$ and $\text{var}(Y) = \sigma_Y^2$:

$$R^2 = \frac{\text{cov}(X, Y)^2}{\sigma_X^2 \sigma_Y^2} = \left(\frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y}\right)^2 \tag{30}$$

This is exactly the square of the correlation coefficient r (see here):

$$R^2 = r^2 \tag{31}$$

21 cosine

In the spirit of this website, beautiful things happen when we imagine data in high dimensional spaces. Let's do that for the correlation between two vectors.

$$\rho(x, y) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{x_i - \bar{x}}{\sigma_x} \right) \left(\frac{y_i - \bar{y}}{\sigma_y} \right) \tag{1}$$

As we saw before, it is particularly useful to rewrite this formula in terms of z-scored variables:

$$\rho(x, y) = \frac{1}{N} \sum_{i=1}^{N} z_{x_i} z_{y_i} \tag{2}$$

where $z_{x_i} = \frac{x_i - \bar{x}}{\sigma_x}$ and $z_{y_i} = \frac{y_i - \bar{y}}{\sigma_y}$.

Now let's see the formula for the dot product of two vectors z_x and z_y:

$$z_x \cdot z_y = \sum_{i=1}^{N} z_{x_i} z_{y_i} = \frac{1}{N} \sum_{i=1}^{N} z_{x_i} z_{y_i} \cdot N = \rho(x, y) \cdot N \tag{3}$$

There is another formula for the dot product that involves the angle θ between the two vectors:

$$z_x \cdot z_y = \|z_x\| \, \|z_y\| \cos(\theta) \tag{4}$$

where $||z_x||$ and $||z_y||$ are the magnitudes (or lengths) of the vectors z_x and z_y.

The magnitude squared of a z-scored vector is:

$$
\begin{aligned}
\|z_x\|^2 &= \sum_{i=1}^{N} z_{x_i}^2 \\
&= \sum_{i=1}^{N} \left(\frac{x_i - \bar{x}}{\sigma_x} \right)^2 \\
&= \frac{1}{\sigma_x^2} \sum_{i=1}^{N} (x_i - \bar{x})^2 \\
&= \frac{1}{\sigma_x^2} \left(\frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2 \right) N \\
&= N \frac{\sigma_x^2}{\sigma_x^2} \\
&= N
\end{aligned} \tag{5}
$$

Of course, the same goes for z_y, so we have $\|z_x\| = \|z_y\| = \sqrt{N}$. Substituting this into Eq. (4) gives:

$$
z_x \cdot z_y = \sqrt{N} \cdot \sqrt{N} \cdot \cos(\theta) = N \cos(\theta) \tag{6}
$$

Finally, we equate Eqs. (3) and (6):

$$
\rho(x, y) = \cos(\theta) \tag{7}
$$

The correlation between two variables is equal to the cosine of the angle between their corresponding z-scored vectors, in a high-dimensional space.

- $\theta = 0$, the vectors point in the same direction, $\cos(\theta) = 1$, indicating a perfect positive correlation.
- $\theta = \pi/2$, the vectors are orthogonal, $\cos(\theta) = 0$, indicating no correlation.
- $\theta = \pi$, the vectors point in opposite directions, $\cos(\theta) = -1$, indicating a perfect negative correlation.

21.1 cosine similarity

The cosine similarity is a measure of similarity between two non-zero vectors. It is defined as:

$$\text{cosine\_similarity}(x, y) = \frac{x \cdot y}{\|x\| \, \|y\|} \tag{8}$$

This measure is common in text analysis, where a non-zero element of a vector represents the presence of a word in a document, and the value of the element represents the frequency of that word. The cosine similarity measures the cosine of the angle between two vectors, which indicates how similar the two vectors are in terms of their direction, regardless of their magnitude. If we were to z-score the vectors, the absence of a word would be represented by a negative value (below average), which is not useful in this context.

When the vectors are z-scored, the cosine similarity is identical to the Pearson correlation coefficient.

- **cosine similarity**: works on any non-zero vectors, does not require z-scoring. There is no need to alter the reference point.
- **Pearson correlation**: works on z-scored vectors, requires centering and scaling. The reference point is the mean of each variable. Useful when comparing variables with different units or scales.

22 significance (p-value)

Given a correlation coefficient r, we can assess its significance using a p-value. Let's formulate the hypotheses:

- **Null Hypothesis (H_0)**: There is no correlation between the two variables (i.e., $r = 0$).
- **Alternative Hypothesis (H_a)**: There is a correlation between the two variables (i.e., $r \neq 0$).

To calculate the p-value, we can use the following formula for the test statistic t:

$$t = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}} \tag{1}$$

where n is the number of data points.

This formula follows the fundamental structure of a t-statistic:

$$t = \frac{\text{Signal}}{\text{Noise}} = \frac{\text{Observed Statistic} - \text{Null Value}}{\text{Standard Error of the Statistic}} \tag{2}$$

Let's rearrange Eq. (1) to match the structure of Eq. (2):

$$t = \frac{r - 0}{\sqrt{\frac{1-r^2}{n-2}}} \tag{3}$$

The numerator is clear enough. Let's discuss the denominator, which represents the standard error of the correlation coefficient.

The term $1 - r^2$ is the proportion of **unexplained variance** in the data. As the correlation r gets stronger (closer to 1

or -1), the unexplained variance gets smaller. This makes intuitive sense: a very strong correlation is less likely to be a result of random chance, so the standard error (noise) should be smaller.

The term $n-2$ is the degrees of freedom. As your sample size n increases, the denominator gets larger, which makes the overall standard error smaller. This also makes sense: a correlation found in a large sample is more reliable and less likely to be a fluke than the same correlation found in a small sample.

Let's try a concrete example.

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

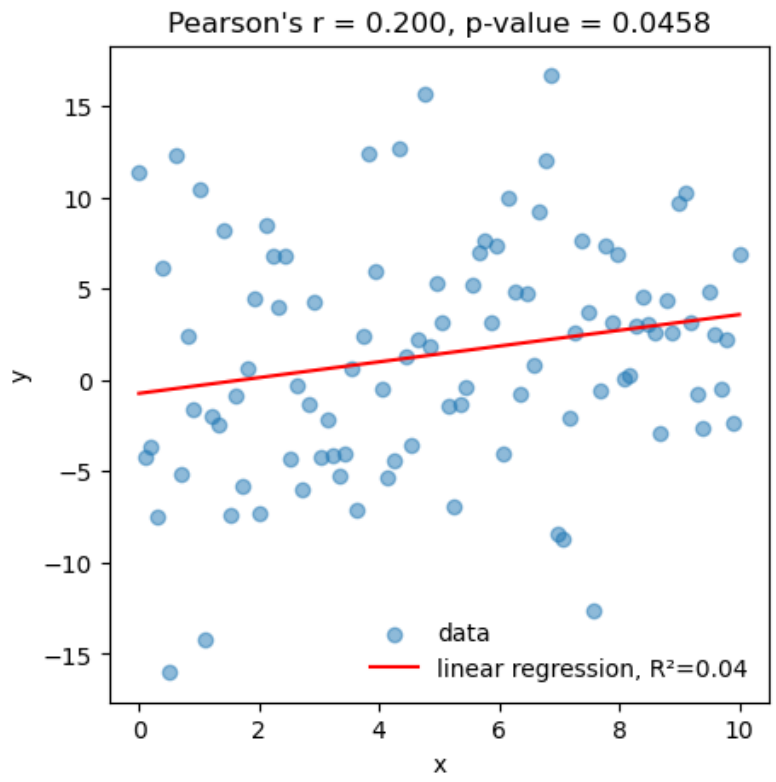
```
# set seed for reproducibility
np.random.seed(1)
N = 100
x = np.linspace(0, 10, N)
y = 0.2 * x + 7*np.random.normal(size=x.size)
```

```
# compute sample z-scores of x, y
zx = (x - np.mean(x)) / np.std(x, ddof=0)
zy = (y - np.mean(y)) / np.std(y, ddof=0)

# compute Pearson correlation coefficient
rho = np.sum(zx * zy) / N
# compute t-statistic
t = rho * np.sqrt((N-2) / (1-rho**2))
# compute two-sided p-value
p = 2 * (1 - stats.t.cdf(np.abs(t), df=N-2))
```

```
fig, ax = plt.subplots(1, 1, figsize=(5, 5))
ax.scatter(x, y, label='data', alpha=0.5)

# linear fit and R2 with scipy
slope, intercept, r_value, p_value, std_err = stats.linregress(x, y)
ax.plot(x, slope*x + intercept, color='red', label=f'linear regression, R²={r_value**2:.2f}')
ax.legend(frameon=False)
```

```
# print p-value
print(f'our p-value:   {p:.4f}')
print(f'scipy p-value: {p_value:.4f}')

# compute correlation coefficients and their p-values
r = np.corrcoef(x, y)[0,1]
ax.set(xlabel='x',
       ylabel='y',
       title=f"Pearson's r = {rho:.3f}, p-value = {p_value:.4f}");
```

```
our p-value:   0.0458
scipy p-value: 0.0458
```



The linear regression accounts for 4% of the variance in the data, which corresponds to a correlation coefficient of $r = 0.2$. This correlation is statistically significant at $p = 0.0458 < 0.05$.

Part VII

bayes

23 from the ground up

23.1 the scenario

Imagine we're in a room with a large group of people. We know the group consists of men and women, and we have height measurements for everyone. Someone walks in, we measure their height, but we don't know if they are a man or a woman. Our goal is to figure out the probability that this person is a man, given their height. For simplicity, let's say that heights are categorized into three groups: short, medium, and tall. The breakdown of the group is as follows:

	Short	Medium	Tall
Man	15	30	20
Woman	25	35	10

23.2 joint and conditional probabilities

What is the probability that a person is both a man and tall?

This is the same as asking: what fraction does the rectangle on the bottom left have with respect to the whole area?

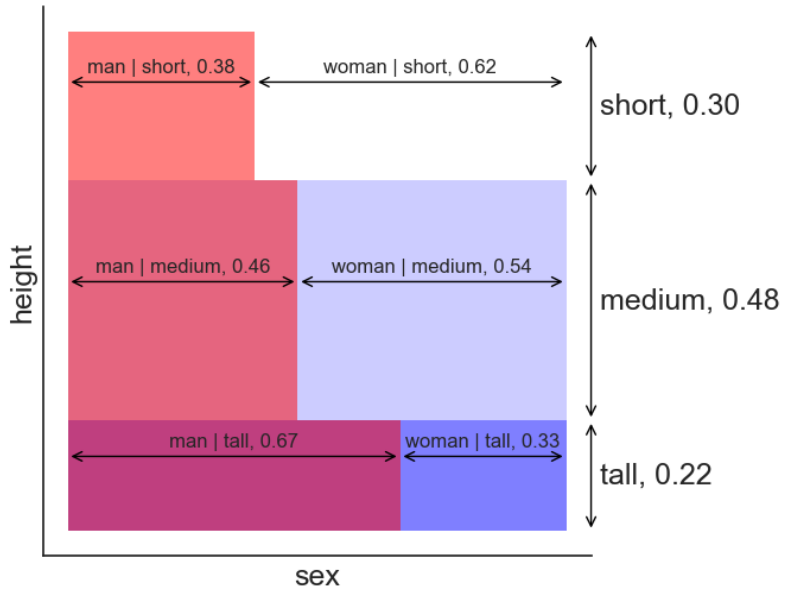
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
```

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.set_aspect('equal')

m1, m2, m3 = 15, 30, 20
f1, f2, f3 = 25, 35, 10

m = m1 + m2 + m3
f = f1 + f2 + f3
total = m + f
h1 = m1 + f1
h2 = m2 + f2
h3 = m3 + f3
h = h1 + h2 + h3

p_3 = h3 / total
p_2 = h2 / total
p_1 = h1 / total
p_m = m / total
p_f = f / total

p_m_given_1 = m1 / h1
p_f_given_1 = f1 / h1
p_m_given_2 = m2 / h2
p_f_given_2 = f2 / h2
p_m_given_3 = m3 / h3
p_f_given_3 = f3 / h3

p_1_given_m = m1 / m
p_2_given_m = m2 / m
p_3_given_m = m3 / m
p_1_given_f = f1 / f
p_2_given_f = f2 / f
p_3_given_f = f3 / f

# tall shaded area
ax.fill_between([0, 1], 0, p_3, color="blue", alpha=0.5, edgecolor="none")
# medium shaded area
ax.fill_between([0, 1], p_3, p_3 + p_2, color="blue", alpha=0.2, edgecolor="none")
# man given tall shaded area
ax.fill_between([0, p_m_given_3], 0, p_3, color="red", alpha=0.5, edgecolor="none")
```

```
# man given medium shaded area
ax.fill_between([0, p_m_given_2], p_3, p_3+p_2, color="red", alpha=0.5, edgecolor="none")
# man given short shaded area
ax.fill_between([0, p_m_given_1], p_3+p_2, 1.0, color="red", alpha=0.5, edgecolor="none")


sns.despine(ax=ax, top=True, right=True)

# tall arrow
ax.annotate("",
            (1.05, 0),
            (1.05, p_3),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.05, p_3 / 2, f" tall, {p_3:.2f}", va="center", rotation=0)
# medium arrow
ax.annotate("",
            (1.05, p_3),
            (1.05, p_2+p_3),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.05, p_3 + (p_2) / 2, f" medium, {p_2:.2f}", va="center", rotation=0)
# short arrow
ax.annotate("",
            (1.05, p_2+p_3),
            (1.05, 1.0),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
```

```
ax.text(1.05, 1.0 - (1.0 - p_2 - p_3) / 2, f" short, {1.0-p_2-p_3:.2f}", va="center", rotation=
# man given short arrow
ax.annotate("",
            (0, 0.90),
            (p_m_given_1, 0.90),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(p_m_given_1 / 2, 0.92, f"man | short, {p_m_given_1:.2f}", ha="center", fontsize=12)

# woman given short arrow
ax.annotate("",
            (1.0 - p_f_given_1, 0.90),
            (1.0, 0.90),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.0 - (p_f_given_1) / 2, 0.92, f"woman | short, {p_f_given_1:.2f}", ha="center", fontsi

# man given medium arrow
ax.annotate("",
            (0, 0.50),
            (p_m_given_2, 0.50),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(p_m_given_2 / 2, 0.52, f"man | medium, {p_m_given_2:.2f}", ha="center", fontsize=12)

# woman given medium arrow
ax.annotate("",
            (1.0 - p_f_given_2, 0.50),
```

```
            (1.0, 0.50),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.0 - (p_f_given_2) / 2, 0.52, f"woman | medium, {p_f_given_2:.2f}", ha="center", fonts

# man given tall arrow
ax.annotate("",
            (0, 0.15),
            (p_m_given_3, 0.15),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(p_m_given_3 / 2, 0.17, f"man | tall, {p_m_given_3:.2f}", ha="center", fontsize=12)

# woman given tall arrow
ax.annotate("",
            (1.0 - p_f_given_3, 0.15),
            (1.0, 0.15),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.0 - (p_f_given_3) / 2, 0.17, f"woman | tall, {p_f_given_3:.2f}", ha="center", fontsi


ax.set(xticks=[],
       yticks=[],
       xlabel="sex",
       ylabel="height",);
```



- The numbers next to each category denote the proportions. That makes sense: according to the table, most of tall people are men, and most of the short people are women.
- "men | tall" is a short way to write "men given tall". In simple words, it is the fraction of men, given that we know the person is tall.

The answer to the question is obvious now. The probability that a person is both **man** and **tall** the product of 0.22 with 0.67.

- 22% of people are tall.
- 67% of those are men.

The answer is 0.22 * 0.67 = 0.1474, or about 15%.

In mathematical notation, we write this as:

$$P(\text{man} \cap \text{tall}) = P(\text{tall}) \cdot P(\text{man}|\text{tall}), \tag{1}$$

where the symbol $\cap$ means "and".

- P(man tall) is called the **joint probability**, because it describes the probability of two events happening together.
- P(man|tall) is called the **conditional probability**, because it describes the probability of one event happening, given that another event is already known to have occurred.

When Eq. (1) is rewritten in terms of $P(\text{man}|\text{tall})$, it is called the equation for conditional probability:

$$P(\text{man}|\text{tall}) = \frac{P(\text{man} \cap \text{tall})}{P(\text{tall})}.$$

Of course, "man" and "tall" are only labels, the general formula we should remember is:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}.$$

23.3 different perspective

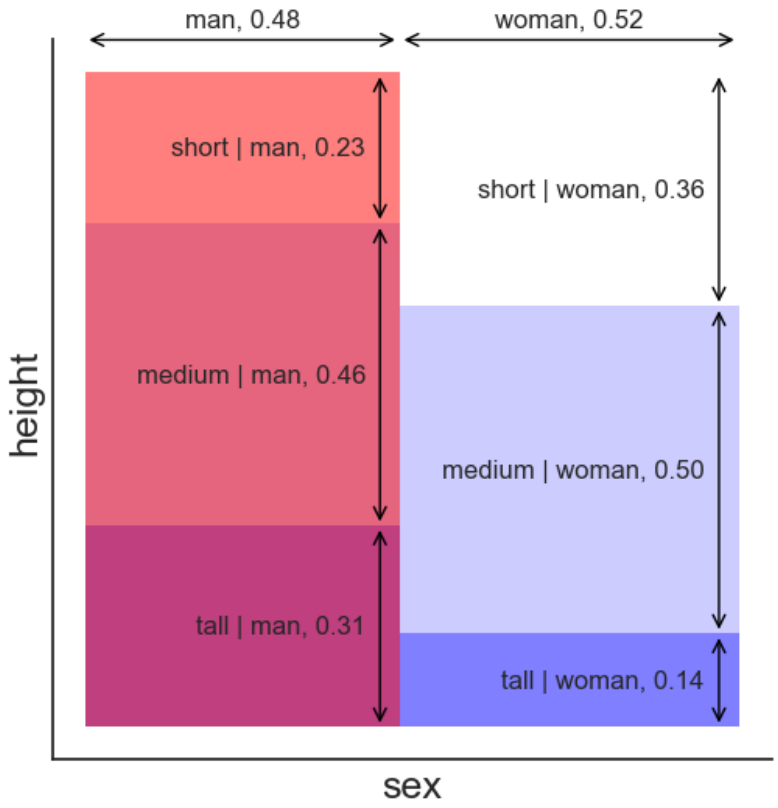
We could have made sense of the data in a different way. Above, we first categorized people by their height, and only then by sex. Let's try the opposite.

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.set_aspect('equal')

# tall given man shaded area
ax.fill_between([0, p_m], 0, p_3_given_m, color="blue", alpha=0.5, edgecolor="none")
# medium given man shaded area
ax.fill_between([0, p_m], p_3_given_m, p_3_given_m+p_2_given_m, color="blue", alpha=0.2, edgec

# man shaded area
ax.fill_between([0, p_m], 0, 1.0, color="red", alpha=0.5, edgecolor="none")

# tall given woman shaded area
ax.fill_between([p_m, 1.0], 0, p_3_given_f, color="blue", alpha=0.5, edgecolor="none")
```

```
# medium given man shaded area
ax.fill_between([p_m, 1.0], p_3_given_f, p_3_given_f+p_2_given_f, color="blue", alpha=0.2, edg

sns.despine(ax=ax, top=True, right=True)

# tall given man arrow
ax.annotate("",
            (p_m-0.03, 0),
            (p_m-0.03, p_3_given_m),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(p_m-0.03, p_3_given_m / 2, f"tall | man, {p_3_given_m:.2f}  ", ha="right", va="center"
# medium given man arrow
ax.annotate("",
            (p_m-0.03, p_3_given_m),
            (p_m-0.03, p_3_given_m+p_2_given_m),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(p_m-0.03, p_3_given_m + p_2_given_m / 2, f"medium | man, {p_2_given_m:.2f}  ", ha="rig
# short given man arrow
ax.annotate("",
            (p_m-0.03, 1.0),
            (p_m-0.03, 1.0-p_1_given_m),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(p_m-0.03, 1.0 - p_1_given_m / 2, f"short | man, {p_1_given_m:.2f}  ", ha="right", va="

# tall given woman arrow
```

```
ax.annotate("",
            (1.0-0.03, 0),
            (1.0-0.03, p_3_given_f),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.0-0.03, p_3_given_f / 2, f"tall | woman, {p_3_given_f:.2f}  ", ha="right", va="cente
# medium given woman arrow
ax.annotate("",
            (1.0-0.03, p_3_given_f),
            (1.0-0.03, p_3_given_f+p_2_given_f),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.0-0.03, p_3_given_f + p_2_given_f / 2, f"medium | woman, {p_2_given_f:.2f}  ", ha="r
# short given woman arrow
ax.annotate("",
            (1.0-0.03, 1.0),
            (1.0-0.03, 1.0-p_1_given_f),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.0-0.03, 1.0 - p_1_given_f / 2, f"short | woman, {p_1_given_f:.2f}  ", ha="right", va


# man arrow
ax.annotate("",
            (0, 1.05),
            (p_m, 1.05),
            ha="right", va="center",
            size=14,
```

```
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(p_m / 2, 1.07, f"man, {p_m:.2f}", ha="center", fontsize=12)
# woman arrow
ax.annotate("",
            (1.0 - p_f, 1.05),
            (1.0, 1.05),
            ha="right", va="center",
            size=14,
            arrowprops=dict(arrowstyle='<->',
                            color="black",
                            ),
)
ax.text(1.0 - (p_f) / 2, 1.07, f"woman, {p_f:.2f}", ha="center", fontsize=12)

ax.set(xticks=[],
       yticks=[],
       # xlim=(0, 1),
       # ylim=(0, 1),
       xlabel="sex",
       ylabel="height",);
```



The probability that a person is both **man** and **tall** is still the area of the purple rectangle on the bottom left. The rectangle has a different shape, but it has to have the same area. The answer to our question now can be understood thus:

- 48% of people are men.
- 31% of those are tall.

The answer is 0.48 * 0.31 = 15%, exactly the same result as before.

In mathematical notation, we write this as:

$$P(\text{tall} \cap \text{man}) = P(\text{man}) \cdot P(\text{tall}|\text{man}). \tag{2}$$

One thing should become clear from the images above. The probability that a person is a man, given that they are tall, **is**

not the same as the probability that a person is tall, given that they are a man. In mathematical notation:

$$P(\text{man}|\text{tall}) \neq P(\text{tall}|\text{man}).$$

- $P(\text{man}|\text{tall})$: of all tall people, 67% are men.
- $P(\text{tall}|\text{man})$: of all men, 31% are tall.

23.4 Bayes' theorem

When two things are true at the same time, it doesn't matter the order we choose to write them. In mathematical notation:

$$P(A \cap B) = P(B \cap A).$$

Because of this, we equate the right-hand sides of Eqs. (1) and (2), and we get Bayes' theorem:

$$P(\text{man}|\text{tall}) \cdot P(\text{tall}) = P(\text{tall}|\text{man}) \cdot P(\text{man})$$

Personally, I choose to remember this equation, because it is symmetric. But the more common way to write Bayes' theorem is to solve for $P(\text{man}|\text{tall})$:

$$P(\text{man}|\text{tall}) = \frac{P(\text{tall}|\text{man}) \cdot P(\text{man})}{P(\text{tall})},$$

or in general terms:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}.$$

Each term has a name:

- $P(A|B)$ is the **posterior**. This is what we are trying to find out: the probability of A given that we know B.
- $P(B|A)$ is the **likelihood**. This is the probability of B given that we know A.

- $P(A)$ is the **prior**. This is what we know about A before we know anything about B.
- $P(B)$ is the **evidence**. This is what we know about B before we know anything about A.

23.5 the law of total probability

In the example above, we had enough information to plug all the numbers into Bayes' theorem. But this is not always the case. Sometimes, we don't know $P(B)$, the evidence. In this case, we can compute it using the law of total probability. It is best to give a concrete example.

A famous use of Bayes' theorem is in desease testing.

- A given desease affects 1% of the population.
- A test for the desease is 95% accurate. This means that:
  - If a person has the desease, the test will be positive 95% of the time.
  - If a person does not have the desease, the test will be negative 95% of the time.
- A person is randomly selected from the population, and they test positive. Should they be worried? What is the probability that they actually have the desease?

Let's translate this into the language of Bayes' theorem:

- A is the event "the person has the desease".
- B is the event "the test is positive".
- We need to find $P(A|B)$, the probability that the person has the desease, given that they tested positive.
- $P(A) = 0.01$ is the prior, the probability that a random person has the desease.
- $P(B|A) = 0.95$ is the likelihood, the probability that the test is positive, given that the person has the desease.

We don't know $P(B)$, the evidence, the probability that a random person tests positive. But we can compute it using the law of total probability:

$$P(\text{test is positive}) = P(\text{test is positive } \textbf{and} \text{ person has the desease}) + P(\text{test is positive } \textbf{and} \text{ person doesn't have the desease}).$$

This has to be true, because a person can either have the desease or not. In a more compact form:

$$P(B) = P(B \cap A) + P(B \cap A^c),$$

where A^c is the complement of A, i.e., "the person does not have the desease".

Using the definition of conditional probability, we can rewrite this as:

$$P(B) = P(B|A) \cdot P(A) + P(B|A^c) \cdot P(A^c).$$

We know all the terms on the right-hand side:

- $P(B|A) = 0.95$, the probability that the test is positive, given that the person has the desease.
- $P(A) = 0.01$, the probability that a random person has the desease.
- $P(A^c) = 0.99$, the probability that a random person does not have the desease.
- $P(B|A^c) = 0.05$, the probability that the test is positive, given that the person does not have the desease. This is 1 minus the accuracy of the test for healthy people.

Plugging in the numbers, we get:

$$P(B) = 0.95 \cdot 0.01 + 0.05 \cdot 0.99 = 0.059.$$

Now we have everything we need to plug the numbers into Bayes' theorem:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} = \frac{0.95 \cdot 0.01}{0.059} = 0.161.$$

This means that, even if the person tested positive, there is only a 16.1% chance that they actually have the desease. This

is counter-intuitive, but it makes sense when we think about it. The desease is very rare, so even if the test is accurate, most of the positive results will be false positives.

In the case that there are several mutually exclusive ways for B to happen, we can generalize the law of total probability:

$$P(B) = \sum_i P(B \cap A_i) = \sum_i P(B|A_i) \cdot P(A_i),$$

where the A_i are all the possible ways for B to happen.

24 parametric generative classification

24.1 question

We are given a list of heights for men and women. Given one more data point (180 cm), could we assign a probability that it belongs to either class?

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm
```

```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
df_girls = pd.read_csv('../archive/data/height/girls_height_stats.csv', index_col=0)
age = 20.0
mu_boys = df_boys.loc[age, 'mu']
mu_girls = df_girls.loc[age, 'mu']
sigma_boys = df_boys.loc[age, 'sigma']
sigma_girls = df_girls.loc[age, 'sigma']

N_boys = 150
N_girls = 200
np.random.seed(314)  # set scipy seed for reproducibility
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
# pandas dataframe with the two samples in it
df = pd.DataFrame({
    'height (cm)': np.concatenate([sample_boys, sample_girls]),
    'sex': ['M'] * N_boys + ['F'] * N_girls
```

```
})
df = df.sample(frac=1, random_state=314).reset_index(drop=True)
df
```

	height (cm)	sex
0	178.558416	M
1	173.334306	M
2	183.084154	M
3	178.236047	F
4	175.868642	M
...	...	...
345	177.387837	M
346	157.122325	F
347	166.891746	F
348	181.090312	M
349	171.479631	M

24.2 explaining "parametric generative classification"

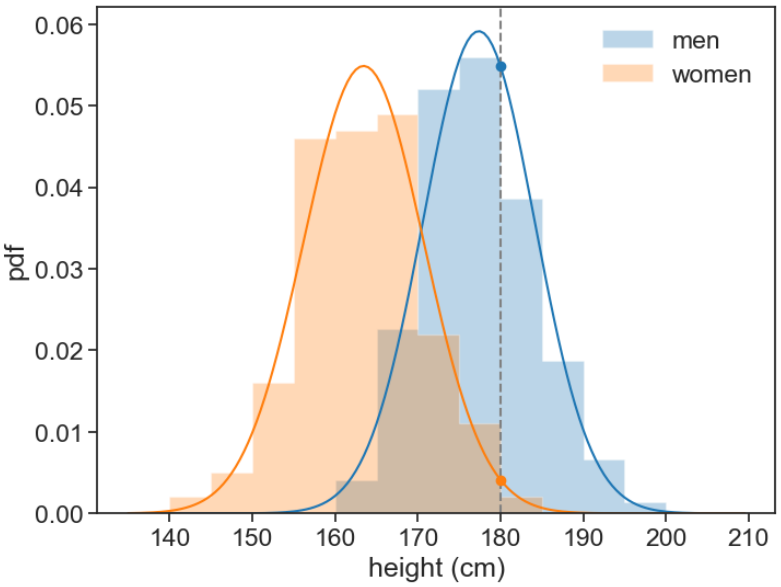
- **Parametric**: we assume a specific distribution for the data. In this case, we'll assume a Gaussian distribution. We call this parametric because the distribution can be fully described by a finite set of **parameters** (mean and variance for Gaussian).
- **Generative**: we model the distribution of each class separately. This would allow us to **generate** new data points from the learned distributions. "Learned" means estimating the parameters of the distributions from the sample data.
- **Classification**: we classify a new data point by comparing the likelihoods of it belonging to each class, given the learned distributions.

24.3 visualizing the problem

```
are_male = df['sex']=='M'
boys_sample = df[are_male]['height (cm)'].to_numpy()
girls_sample = df[~are_male]['height (cm)'].to_numpy()

xbar_boys = boys_sample.mean()
xbar_girls = girls_sample.mean()
s_boys = boys_sample.std(ddof=1)
s_girls = girls_sample.std(ddof=1)
```

```
fig, ax = plt.subplots(figsize=(8, 6))

# plot histogram
bins = np.arange(135, 210, 5)
ax.hist(boys_sample, bins=bins, alpha=0.3, density=True, label='men', color='tab:blue', histty
ax.hist(girls_sample, bins=bins, alpha=0.3, density=True, label='women', color='tab:orange', h
# plot gaussian pdf based on sample mean and std
x = np.arange(135, 210, 0.5)
pdf_boys = norm.pdf(x, loc=xbar_boys, scale=s_boys)
pdf_girls = norm.pdf(x, loc=xbar_girls, scale=s_girls)
ax.plot(x, pdf_boys, color='tab:blue')
ax.plot(x, pdf_girls, color='tab:orange')
h0 = 180
# plot vertical line at h0
ax.axvline(h0, color='gray', linestyle='--')
# plot circles where each pdf intersects h0
likelihood_boys = norm.pdf(h0, loc=xbar_boys, scale=s_boys)
likelihood_girls = norm.pdf(h0, loc=xbar_girls, scale=s_girls)
ax.plot(h0, likelihood_boys, marker='o', color='tab:blue')
ax.plot(h0, likelihood_girls, marker='o', color='tab:orange')

print(f"men:   mean={xbar_boys:.1f} cm, std={s_boys:.1f} cm")
print(f"women: mean={xbar_girls:.1f} cm, std={s_girls:.1f} cm")

ax.legend(frameon=False)
ax.set_xlabel('height (cm)')
ax.set_ylabel('pdf');
```

```
men:   mean=177.4 cm, std=6.7 cm
women: mean=163.5 cm, std=7.3 cm
```



From the sample data, we can compute the mean and standard deviation for each sex (the generative part). We printed these values above. We can then use these parameters to compute the likelihood of the new data point (180 cm) belonging to each class using the Gaussian probability density function (the parametric part). These are plotted as circles in the graph.

24.4 Bayes' theorem

We can then use Bayes' theorem to compute the posterior probabilities of the new data point belonging to each class (the classification part). Bayes' theorem states that:

$$P(\text{man}|x) = \frac{P(x|\text{man})}{P(x)} P(\text{man})$$

- **posterior**, $P(\text{man}|x)$. This is what we are looking for, the probability of a new data point corresponding to a man, given that its height is x.

- **likelihood**, $P(x|\text{man})$. This is the likelihood of observing a height x given that we know it is a man.
- **evidence**, $P(x)$. This is the total probability of observing a height x across all classes (regardless of sex). It acts as a normalization factor. We calculate the evidence using the law of total probability:

$$\begin{aligned} P(x) &= P(x \cap \text{man}) + P(x \cap \text{woman}) \\ &= P(x|\text{man}) \cdot P(\text{man}) + P(x|\text{woman}) \cdot P(\text{woman}) \end{aligned}$$

- **prior**, $P(\text{man})$. This is the overall probability of a person being a man in my dataset (regardless of their height).

Think of this as a "battle of likelihoods," adjusted for the group sizes. You have two groups, men and women, and you've modeled their typical heights. When you get a new height, x, you ask two main questions:

1. **Likelihood Question:** How "typical" is height x for a man compared to how typical it is for a woman? If men in your data are generally tall and x is a tall height, it's more likely to be a man. We measure this "typicalness" using a probability distribution.
2. **Prior Belief Question:** In your dataset, are men or women more common? If your dataset contains 90 women and 10 men, any new person is, initially, more likely to be a woman, regardless of their height.

24.5 Step-by-Step Calculation

Step 1: Model Your Data

We assume a Gaussian distribution, and calculate the sample mean and standard deviation for each sex.

```
men:   mean=177.4 cm, std=6.7 cm
women: mean=163.5 cm, std=7.3 cm
```

Step 2: Calculate the Priors

The priors are simply the proportion of each group in the total dataset.

$$P(\text{Man}) = \frac{N_{\text{men}}}{N_{\text{men}} + N_{\text{women}}}$$

$$P(\text{Woman}) = \frac{N_{\text{women}}}{N_{\text{men}} + N_{\text{women}}}$$

Step 3: Calculate the Likelihoods

Using the normal distribution's probability density function (PDF), find the likelihood of the new height h for each model. The PDF formula is:

$$f(x|\bar{x}, s) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\bar{x}}{s}\right)^2}$$

1. **Likelihood for Men:** Plug x into the PDF for the men's model.
   $P(x|\text{Man}) = f(x|\bar{x}_{\text{men}}, s_{\text{men}})$
2. **Likelihood for Women:** Plug x into the PDF for the women's model.
   $P(x|\text{Woman}) = f(x|\bar{x}_{\text{women}}, s_{\text{women}})$

Step 4: Put It All Together

Now, apply Bayes' Theorem. The "evidence" term $P(x)$ in the denominator is the sum of all ways you could observe height x:

$$P(x) = P(x|\text{Man}) \cdot P(\text{Man}) + P(x|\text{Woman}) \cdot P(\text{Woman})$$

So, the final calculation for the probability of being a man is:

$$P(\text{Man}|x) = \frac{P(x|\text{Man}) \cdot P(\text{Man})}{P(x|\text{Man}) \cdot P(\text{Man}) + P(x|\text{Woman}) \cdot P(\text{Woman})}$$

.

Crunching the number gives:

```
h0 = 180.0
likelihood_boys = norm.pdf(h0, loc=xbar_boys, scale=s_boys)
likelihood_girls = norm.pdf(h0, loc=xbar_girls, scale=s_girls)
prior_boys = N_boys / (N_boys + N_girls)
prior_girls = N_girls / (N_boys + N_girls)
evidence = likelihood_boys * prior_boys + likelihood_girls * prior_girls
p_man_given_180 = likelihood_boys * prior_boys / evidence
print(f"Answer: {p_man_given_180 * 100:.2f}%.")
```

```
Answer: 90.92%.
```

The probability that the person is a man, given that their height is 180 cm, is 90.92%

25 odds and log likelihood

25.1 the scenario

Imagine we are researchers studying a potential link between a specific mutated gene and a certain disease. We have collected data from a sample of 356 people.

Here's our data:

	Has Disease	**No Disease**
Has Mutated Gene	23	117
No Mutated Gene	6	210

Our goal is to figure out how finding this mutated gene in a person should change our belief about whether they have the disease.

25.2 prior

The **prior probability** of someone having the disease is the chance of having the disease before we know anything about their gene status. We can calculate this from our data.

$$P(\text{Disease}) = \frac{\text{Number of people with the disease}}{\text{Total number of people}}$$

From our table:

$$P(\text{Disease}) = \frac{23 + 6}{23 + 117 + 6 + 210} = \frac{29}{356} \approx 0.081$$

25.3 odds

Odds are a different way to express the same information. Odds compare the chance of an event happening to the chance of it *not* happening.

$$\text{Odds} = \frac{P(\text{event})}{P(\text{not event})}$$

In our case, "event" is having the disease. Because we have only two choices, the probability of not having the disease is simply $1 - P(\text{Disease})$, therefore:

$$\text{Odds(Disease)} = \frac{P(\text{Disease})}{1 - P(\text{Disease})}$$

Plugging in the numbers:

$$\text{Odds(Disease)} = \frac{29/356}{1 - 29/356} \approx 0.0887 \approx \frac{1}{11}$$

This means that for every person with the disease, about 11 do not have it.

25.3.1 log odds

The **log odds** is simply the natural logarithm of the odds.

$$\text{Log Odds} = \ln\left(\frac{P}{1-P}\right)$$

We will see soon enough why this is useful. For now, let's point out that:

- if the odds are 1 (meaning a 50/50 chance), the log odds is 0.
- if the odds are greater than 1 (more likely than not), the log odds is positive.
- if the odds are less than 1 (less likely than not), the log odds is negative.

- the log odds have a symmetric shape around $p = 1/2$, see figure below.

For our example, the log odds of having the disease is:

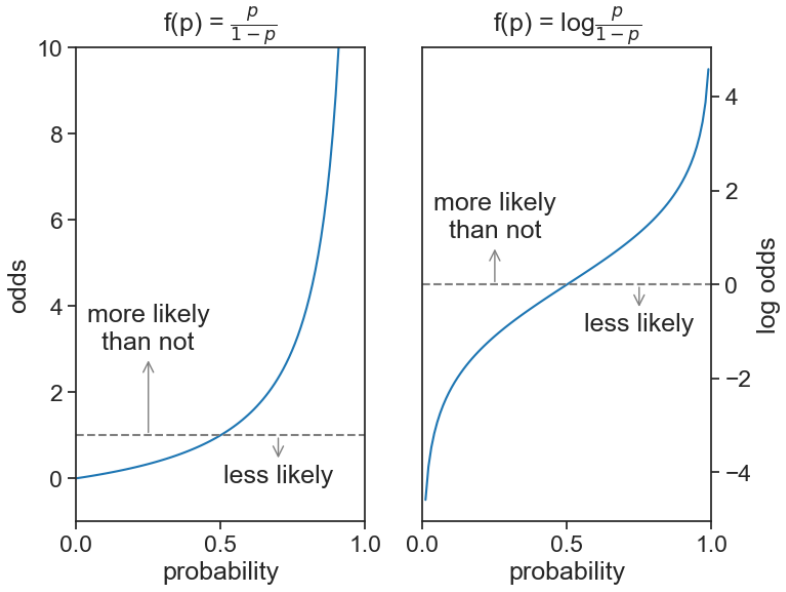
$$\text{Log Odds(Disease)} = \ln(\text{Odds(Disease)}) = \ln(0.0887) \approx -2.42$$

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
```

```
fig, ax = plt.subplots(1, 2, figsize=(8, 6))

p = np.linspace(0, 1, 100)
odds = p / (1 - p)
ax[0].plot(p, odds, label="odds", color="tab:blue");
ax[0].axhline(1, ls="--", color="gray")
ax[0].set(ylabel="odds",
          xlabel="probability",
          title=r"f(p) = $\frac{p}{1-p}$",
          ylim=(-1, 10),
          xlim=(0, 1),
          xticks=[0,0.5,1]);
ax[0].annotate("more likely\nthan not", xy=(0.25, 1), xytext=(0.25, 3),
               ha="center", arrowprops=dict(arrowstyle="<-", color="gray"))
ax[0].annotate("less likely", xy=(0.7, 1), xytext=(0.7, -0.1),
               ha="center", arrowprops=dict(arrowstyle="<-", color="gray"))

ax[1].plot(p, np.log(odds), label="odds", color="tab:blue")
ax[1].axhline(0, ls="--", color="gray")
ax[1].yaxis.set_ticks_position('right')
ax[1].yaxis.set_label_position('right')
ax[1].set(ylabel="log odds",
          xlabel="probability",
          title=r"f(p) = log$\frac{p}{1-p}$",
          xlim=(0, 1),
          xticks=[0,0.5,1]);
```

```
ax[1].annotate("more likely\nthan not", xy=(0.25, 0), xytext=(0.25, 1),
               ha="center", arrowprops=dict(arrowstyle="<-", color="gray"))
ax[1].annotate("less likely", xy=(0.75, 0), xytext=(0.75, -1),
               ha="center", arrowprops=dict(arrowstyle="<-", color="gray"));
```

```
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10104/1581004145.py:5: RuntimeWarni
  odds = p / (1 - p)
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10104/1581004145.py:19: RuntimeWarn
  ax[1].plot(p, np.log(odds), label="odds", color="tab:blue")
```



25.4 Bayes' theorem in odds form

The standard form of Bayes' theorem is:

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}. \qquad (1)$$

In our example, the hypothesis H is "having the disease", and the evidence E is detecting the mutated gene.

Let's write Bayes' theorem for the alternative hypothesis $\neg H$ ("not having the disease"):

$$P(\neg H|E) = \frac{P(E|\neg H) \cdot P(\neg H)}{P(E)}. \tag{2}$$

The **odds form** of Bayes' theorem is the ratio of these two equations:

$$\underbrace{\frac{P(H|E)}{P(\neg H|E)}}_{\text{posterior odds}} = \underbrace{\frac{P(E|H)}{P(E|\neg H)}}_{\text{likelihood ratio}} \cdot \underbrace{\frac{P(H)}{P(\neg H)}}_{\text{prior odds}} \tag{3}$$

We already discussed the prior odds. The posterior odds represent the odds of having the desease *after* we have seen the evidence (the mutated gene). The new piece we need is the **likelihood ratio**.

25.5 likelihood ratio

The likelihood ratio (LR) tells us how much more likely we are to see the evidence if the hypothesis is true compared to if it is false.

$$\text{LR} = \frac{P(E|H)}{P(E|\neg H)}$$

We can compute this from our data:

- $P(E|H)$: the probability of having the mutated gene given that the person has the disease. From the left column in our table, this is $\frac{23}{23+6} \approx 0.793$.
- $P(E|\neg H)$: the probability of having the mutated gene given that the person does not have the disease. From the right column in our table, this is $\frac{117}{117+210} \approx 0.358$.

Finally:

$$\text{LR} = \frac{23/29}{117/327} \approx 2.22$$

The interpretation is that seeing the mutated gene is about 2.22 times more likely if the person has the disease than if they do not.

25.6 log likelihood ratio

Here too, taking the logarithm transforms a quantity between 0 and infinity into a number between negative infinity and positive infinity. The log likelihood ratio is:

$$\text{Log-LR} = \ln(\text{LR}) = \ln(2.22) \approx 0.797$$

The fact that this is a positive number says that seeing the mutated gene **increases** our belief that the person has the disease.

25.7 Bayes' theorem in log odds form

Taking the logarith of Eq. (3) gives us the log odds form of Bayes' theorem:

$$\underbrace{\ln\left(\frac{P(H|E)}{P(\neg H|E)}\right)}_{\text{posterior log odds}} = \underbrace{\ln\left(\frac{P(E|H)}{P(E|\neg H)}\right)}_{\text{log-likelihood ratio}} + \underbrace{\ln\left(\frac{P(H)}{P(\neg H)}\right)}_{\text{prior log odds}} \quad (4)$$

Plugging in our numbers, we can see how our belief about the person having the disease changes after seeing the evidence (the mutated gene).

$$\begin{aligned}\text{Posterior Log Odds} &= \text{Log-LR} + \text{Prior Log Odds} \\ &= 0.797 + (-2.42) \\ &\approx -1.623\end{aligned}$$

From the posterior log odds, we can get back to the posterior odds by exponentiating:

$$\text{Posterior Odds} = e^{\text{Posterior Log Odds}} = e^{-1.623} \approx 0.197$$

Finally, we can convert the posterior odds back to a probability:

$$P(H|E) = \frac{\text{Posterior Odds}}{1 + \text{Posterior Odds}} = \frac{0.197}{1 + 0.197} \approx 0.164$$

This means that after seeing the mutated gene, our estimate of the probability that the person has the disease has increased from about 8.1% to about 16.4%.

26 logistic connection

26.1 from Bayes the logistic

The arguments below follow those in subsection 12.2 of "Introduction to Environmental Data Science" by William W. Hsieh.

We start with Bayes' theorem for two classes C_1 and C_2:

$$P(C_1|x) = \frac{P(x|C_1)P(C_1)}{P(x)} \tag{1}$$

Using the law of total probability in the denominator, we get:

$$P(C_1|x) = \frac{P(x|C_1)P(C_1)}{P(x|C_1)P(C_1) + P(x|C_2)P(C_2)} \tag{2}$$

We now divide the numerator and denominator by $P(x|C_1)P(C_1)$:

$$P(C_1|x) = \frac{1}{1 + \frac{P(x|C_2)P(C_2)}{P(x|C_1)P(C_1)}} \tag{3}$$

We now note that the ratio $P(C_2|x)/P(C_1|x)$ can be expressed as:

$$\frac{P(C_2|x)}{P(C_1|x)} = \frac{\frac{P(x|C_2)P(C_2)}{P(x)}}{\frac{P(x|C_1)P(C_1)}{P(x)}} = \frac{P(x|C_2)P(C_2)}{P(x|C_1)P(C_1)} \tag{4}$$

In the expression above, we used the Bayes' theorem in (1) to express $P(C_2|x)$ and $P(C_1|x)$ in terms of $P(x|C_2)$ and $P(x|C_1)$. We can now rewrite (3) as:

$$P(C_1|x) = \frac{1}{1 + \frac{P(C_2|x)}{P(C_1|x)}} = \frac{1}{1 + \left(\frac{P(C_1|x)}{P(C_2|x)}\right)^{-1}} \tag{5}$$

The posterior probability $P(C_1|x)$ is a function of the ratio $P(C_1|x)/P(C_2|x)$. This ratio is called the **posterior odds**, or simply **odds**. We can make this function look like a sigmoid function by taking the logarithm of the posterior odds. The logarithm of the posterior odds is called the **log-odds** or **logit**:

$$\text{logit} = u = \ln\left(\frac{P(C_1|x)}{P(C_2|x)}\right) \tag{6}$$

We can now rewrite (5) in terms of the logit:

$$P(C_1|x) = \frac{1}{1 + e^{-u}} \tag{7}$$

Finally, we assume that there is a linear relationship between u and the features x:

$$u = \sum_j w_j x_j + w_0 = \mathbf{w}^T\mathbf{x} + w_0 \tag{8}$$

We now have the logistic function that connects the features x to the posterior probability $P(C_1|x)$:

$$P(C_1|x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x} + w_0)}} \tag{9}$$

The one assumption that is needed to make the connection from Bayes' theorem to the logistic function is that there is a linear relationship between the log-odds and the features x:

$$\ln\left(\frac{P(C_1|x)}{P(C_2|x)}\right) = \mathbf{w}^T\mathbf{x} + w_0 \tag{10}$$

This seems a rather arbitrary assumption. Why does this make sense?

1. A linear relationship between the log odds and the features is simple and easy to interpret.
2. Linear models are easy to implement and computationally efficient.
3. In a few specific cases (see below) the linearity doesn't have to be assumed, it emerges naturally from the model.

26.2 emergent linearity

Let's start from the log odds definition in (6):

$$u = \ln\left(\frac{P(C_1|x)}{P(C_2|x)}\right) = \ln\left(\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)}\right) \tag{11}$$

We rewrite this as:

$$\begin{aligned} u &= \ln\frac{P(x|C_1)}{P(x|C_2)} + \ln\frac{P(C_1)}{P(C_2)} \\ &= \ln P(x|C_1) - \ln P(x|C_2) + \ln\frac{P(C_1)}{P(C_2)}. \end{aligned} \tag{12}$$

We now **make the assumption** that the likelihoods $P(x|C_k)$ are Gaussian distributions. For simplicity, let's assume that x is a single feature (univariate case).

$$P(x|C_k) = \frac{1}{\sqrt{2\pi}\sigma_k}\exp\left(-\frac{(x-\mu_k)^2}{2\sigma_k^2}\right), \tag{13}$$

where C_k are the two classes we have, C_1 and C_2.

We now calculate the log of the likelihoods:

$$\ln P(x|C_k) = -\ln\sqrt{2\pi\sigma_k^2} - \frac{(x-\mu_k)^2}{2\sigma_k^2}. \tag{14}$$

We now substitute this into Eq. (12) for the log odds:

$$u = -\ln\sqrt{2\pi\sigma_1^2} - \frac{(x-\mu_1)^2}{2\sigma_1^2} + \ln\sqrt{2\pi\sigma_2^2} + \frac{(x-\mu_2)^2}{2\sigma_2^2} + \ln\frac{P(C_1)}{P(C_2)}$$
$$= \ln\frac{\sigma_2}{\sigma_1} + \frac{(x-\mu_2)^2}{2\sigma_2^2} - \frac{(x-\mu_1)^2}{2\sigma_1^2} + \ln\frac{P(C_1)}{P(C_2)}. \quad (15)$$

KEY ASSUMPTION: if we assume that the two classes have the same variance, $\sigma_1 = \sigma_2 = \sigma$, the expression simplifies to:

$$u = \frac{1}{2\sigma^2}\left((x-\mu_2)^2 - (x-\mu_1)^2\right) + \ln\frac{P(C_1)}{P(C_2)}$$
$$= \frac{1}{2\sigma^2}\left(x^2 - 2x\mu_2 + \mu_2^2 - x^2 + 2x\mu_1 - \mu_1^2\right) + \ln\frac{P(C_1)}{P(C_2)}$$
$$= \frac{\mu_1 - \mu_2}{\sigma^2}x + \frac{\mu_2^2 - \mu_1^2}{2\sigma^2} + \ln\frac{P(C_1)}{P(C_2)}. \quad (16)$$

The first term depends on x linearly, and the other two terms are constants. We can thus rewrite the log odds u as:

$$u = wx + w_0, \quad (17)$$

where

$$w = \frac{\mu_1 - \mu_2}{\sigma^2}, \quad w_0 = \frac{\mu_2^2 - \mu_1^2}{2\sigma^2} + \ln\frac{P(C_1)}{P(C_2)}. \quad (18)$$

Under the assumption that the distributions have equal variance, the posterior probability can be expressed as a logistic function of a linear combination of the input feature.

This is probably the simplest example of a connection between a generative model (Gaussian distributions for each class) and a discriminative model (logistic regression). It would work for other distributions from the exponential family, e.g., Poisson, Bernoulli, Exponential, etc. The one condition they all need to satisfy is that the non-linear part of the log-likelihoods cancels out when we compute the log-odds, leaving a linear function of x.

When we have real data in our hands, we usually don't know the underlying distributions. The calculation above showed us that a linear relationship between the log odds and the features naturally emerges in a few cases, and this is the motivation for the wider assumption in (10).

Part VIII

svd and pca

27 SVD for image compression

This chapter is partially based on these sources:

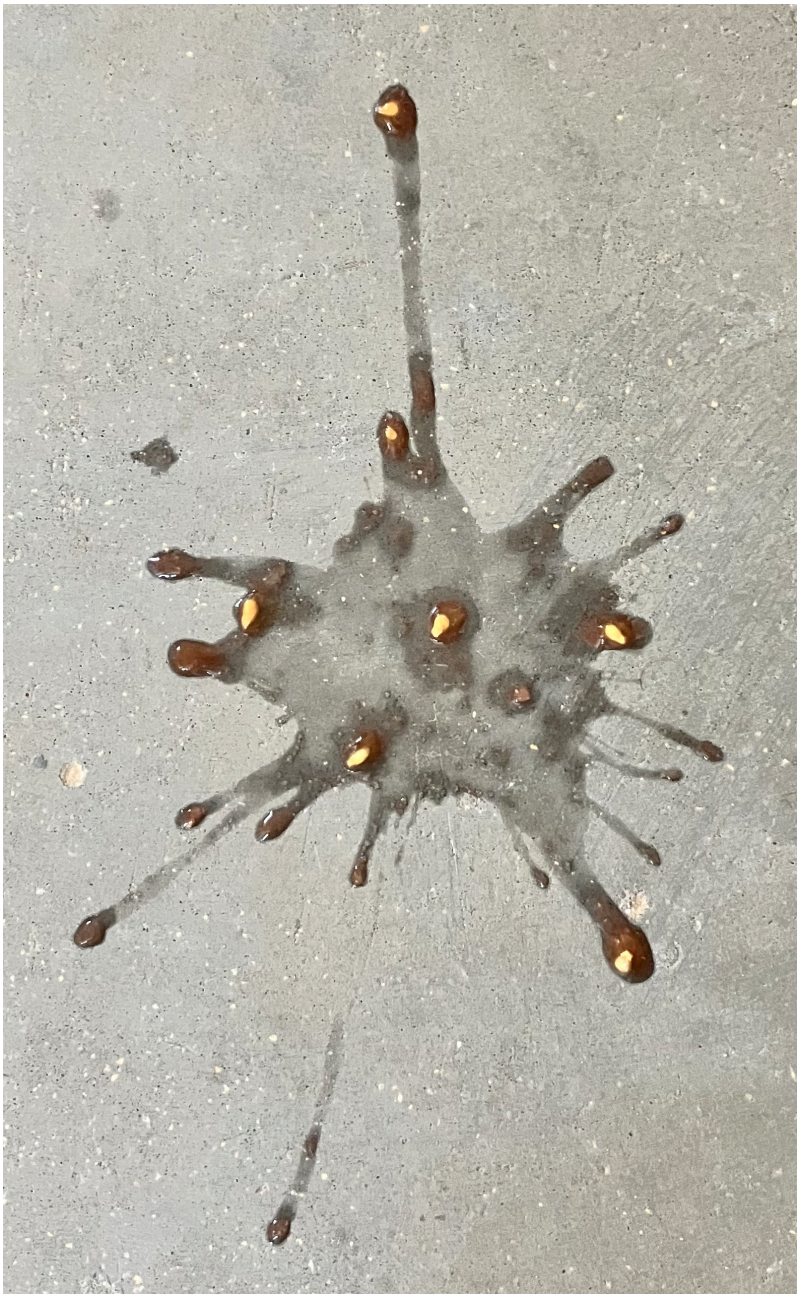
- "Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control" by Steven L. Brunton, J. Nathan Kutz. SVD is covered in Chapter 1.
- Dr. Roi Yehoshua's "Singular Value Decomposition (SVD), Demystified".

Check out also this cool demo by Tim Baumann.

27.1 the image

I will use a black-and-white version of the photo below as the matrix to decompose. There are two reasons to use this image:

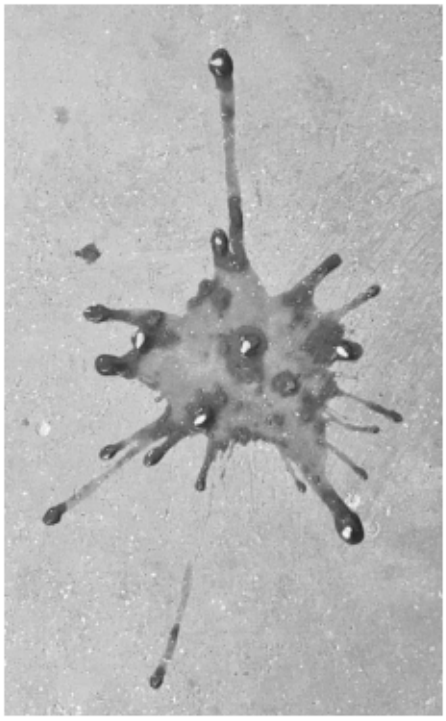
1. it is a tall and skinny matrix (width 1600 px, height 2600 px). Tall and skinny matrices are usually used in overdetermined systems, which are common in data science.
2. this is the image of the juice of a tomato I ate, as it fell on the concrete floor. I found this splat pattern so beautiful that I took a picture, and I wanted to immortalize it in this tutorial. You're welcome.



```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import time
```

```
from sklearn.decomposition import TruncatedSVD
```

```
image = Image.open('../archive/images/splat.jpg')
gray_image = image.convert('L')  # convert to grayscale
image_array = np.array(gray_image)  # make it a numpy array
# display the image
fig, ax = plt.subplots()
ax.imshow(image_array, cmap='gray')
ax.axis('off')  # Hide axis
```



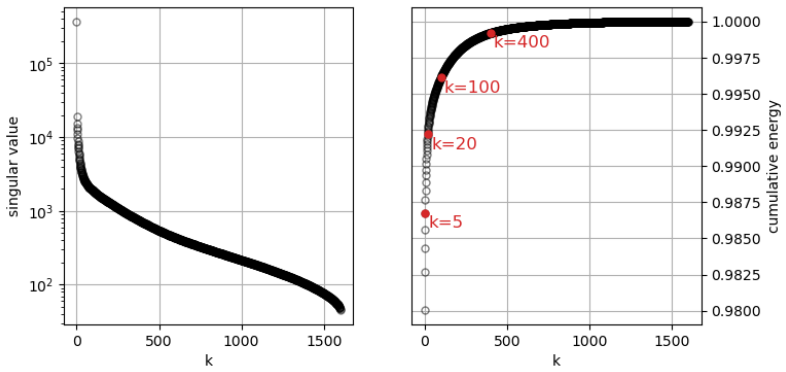
27.2 decomposition

```
U, S, Vt = np.linalg.svd(image_array)
```

Let's see how the magnitude of the singular values decreases as we go from the first to the last (k goes from zero to 900-1).

Also, let's see how much of the total energy accumulated up to the k-th singular value squared.

```
fig, ax = plt.subplots(1, 2, figsize=(8, 4))
lenS = len(S)
ax[0].plot(np.arange(lenS), S, marker='o', markeredgecolor="black", markerfacecolor='None', ma
ax[0].set_yscale('log')  # Set y-axis to log scale
ax[0].set(xlabel='k',
          ylabel='singular value'
         )
ax[0].grid(True)  # Enable grid on the first panel

cumS2 = np.cumsum(S**2) / np.sum(S**2)
ax[1].plot(np.arange(lenS), cumS2, marker='o', markeredgecolor="black", markerfacecolor='None'
klist = [5, 20, 100, 400]
for k in klist:
    ax[1].plot([k-1], [cumS2[k-1]], marker='o', markeredgecolor="tab:red", markerfacecolor='tal
    ax[1].text(k+20, cumS2[k-1]-0.001, f'k={k}', color='tab:red', fontsize=12, ha='left')

ax[1].set(xlabel='k',
          ylabel='cumulative energy'
         )
ax[1].grid(True)  # Enable grid on the second panel
ax[1].yaxis.set_label_position("right")  # Move ylabel to the right
ax[1].yaxis.tick_right()  # Move yticks to the right
```



The square of the Frobenius norm of the matrix X is equal to the sum of the squares of all its singular values.

$$\|X\|_F^2 = \sum_{i=1}^{r} \sigma_i^2$$

The Frobenius norm is a measure of the "magnitude" or "size" of the matrix, which can be interpreted as the total amount of "information" in the data. By taking the cumulative sum of the squared singular values, we are effectively measuring how much of this total information is retained with each successive truncation. The term "energy" is an analogy from physics and signal processing. In these fields, the total energy of a signal is often defined as the integral of its squared magnitude over time. This concept carries over to data analysis where the squared singular values are a direct measure of the variance in the data along each singular vector, and the sum of these squares represents the total variance.

27.3 truncation and reconstruction

Our original matrix X has dimensions (1600, 2600) and rank 1600. Therefore, it has 1600 non-zero singular values. We can truncate the SVD to a lower rank $k < 1600$ and reconstruct an approximation of the original matrix using only the first k singular values and their corresponding singular vectors:

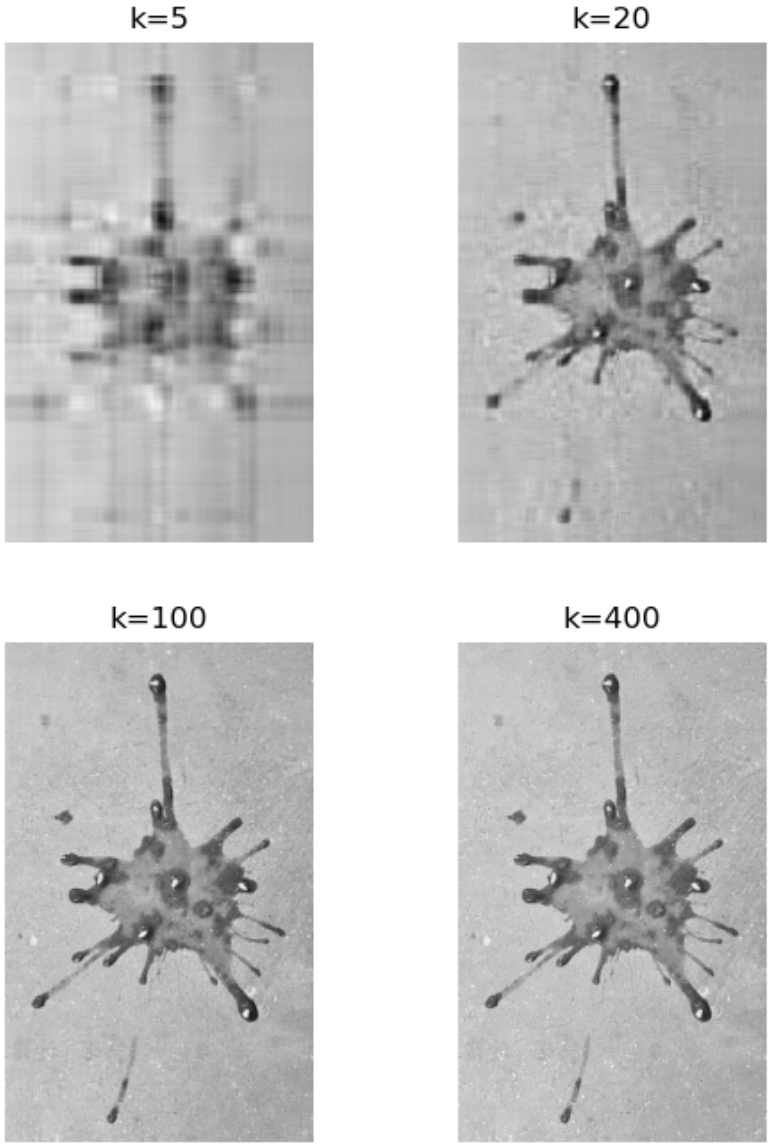
$$X_k = U_k \Sigma_k V_k^T = \sum_{i=1}^{k} \sigma_i \cdot \text{outer}(u_i, v_i^T)$$

where U_k is the matrix of the first k left singular vectors, Σ_k is the diagonal matrix of the first k singular values, and V_k^T is the transpose of the matrix of the first k right singular vectors. The outer product $\text{outer}(u_i, v_i^T)$ creates a rank-1 matrix from the i-th left and right singular vectors.

Using the same truncation values k shown in red in the plot above, we can reconstruct approximations of the original image. As k increases, the reconstructed image becomes more detailed and closer to the original image.

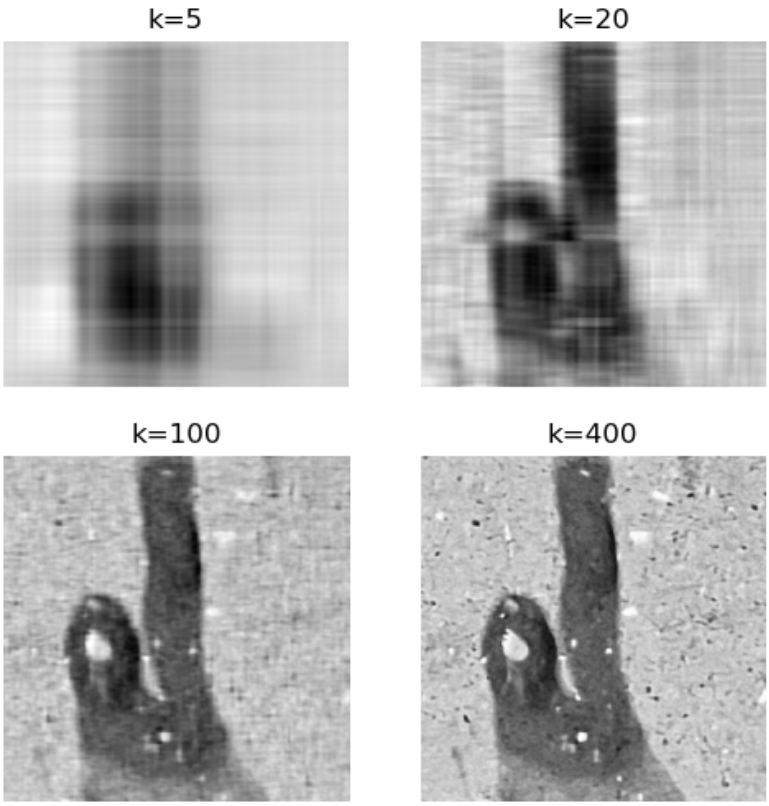
```
def reconstruct_image(U, S, Vt, k):
    X_reconstructed = np.zeros_like(image_array, dtype=np.float64)
    for i in range(k):
        X_reconstructed += S[i] * np.outer(U[:, i], Vt[i, :])
    X_reconstructed = np.clip(X_reconstructed, 0, 255)  # ensure values are in byte range
    X_reconstructed = X_reconstructed.astype(np.uint8)  # convert to uint8
    return X_reconstructed

reconstructed_images = []
for k in klist:
    X_reconstructed = reconstruct_image(U, S, Vt, k)
    reconstructed_images.append(X_reconstructed)
```

```
fig = plt.figure(figsize=(6, 8))
for i, k in enumerate(klist):
    ax = fig.add_subplot(2, 2, i+1)
    X_k = reconstructed_images[i]
    ax.imshow(X_k, cmap='gray')
    ax.set_title(f'k={k}')
    ax.axis('off')  # Hide axis
```



The truncation for $k = 5$ gives a blurry image, but for $k = 20$ it is recognizably a tomato splat. The reconstructions for $k = 100$ and $k = 400$ seem indistinguishable at this resolution. Let's zoom in on a small section of the image to see the differences more clearly.

```
fig = plt.figure(figsize=(6, 6))
for i, k in enumerate(klist):
    ax = fig.add_subplot(2, 2, i+1)
    X_k = reconstructed_images[i][700:1000, 700:1000]  # zoom in
    ax.imshow(X_k, cmap='gray')
    ax.set_title(f'k={k}')
    ax.axis('off')  # Hide axis
```



To capture all the details in the concrete floor we need more than 100 singular values. If we're interested in the overall shape of the splat, 100 singular values are more than enough. This justifies the name of this chapter: **SVD for image compression**. We can compress images by storing only the first k singular values and their corresponding singular vectors, instead of the entire image matrix.

27.4 computational efficiency

We calculated the reconstruction "the hard way", by explicitly forming the outer products and summing them. However, we can also use matrix multiplication to achieve the same result more efficiently.

$$X_k = U_k \Sigma_k V_k^T$$

where:

- U_k is the matrix formed by the first k columns of U.
- Σ_k is the $k \times k$ diagonal matrix formed by the first k singular values, $\Sigma_{11} = \sigma_1$, $\Sigma_{22} = \sigma_2$, etc.
- V_k^T is the matrix formed by the first k rows of V^T.

Let's leverage matrix multiplication to compute the reconstruction: $X_k = U_k(\Sigma_k V_k^T)$.

First, let's look at the product of the diagonal singular value matrix Σ_k and the truncated V^T matrix, V_k^T. Σ_k is a $k \times k$ diagonal matrix with singular values $\sigma_1, \sigma_2, ..., \sigma_k$ on the diagonal. V_k^T is a $k \times n$ matrix where each row is a singular vector v_i^T.

$$\Sigma_k V_k^T = \begin{bmatrix} \sigma_1 & 0 & ... & 0 \\ 0 & \sigma_2 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & \sigma_k \end{bmatrix} \begin{bmatrix} — & v_1^T & — \\ — & v_2^T & — \\ & \vdots & \\ — & v_k^T & — \end{bmatrix}$$

Multiplying a diagonal matrix by another matrix from the left scales each row of the second matrix by the corresponding diagonal element of the first matrix.

$$\Sigma_k V_k^T = \begin{bmatrix} — & \sigma_1 v_1^T & — \\ — & \sigma_2 v_2^T & — \\ & \vdots & \\ — & \sigma_k v_k^T & — \end{bmatrix}$$

This matrix contains the scaled singular vectors as its rows.

Now, we multiply the truncated U matrix, U_k, with the result from the first step. U_k is an $m \times k$ matrix whose columns are the singular vectors $u_1, u_2, ..., u_k$. Let's call the result from the first step, the matrix A. The product is $U_k A$.

$$X_k = U_k A = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & ... & u_k \\ | & | & & | \end{bmatrix} \begin{bmatrix} — & \sigma_1 v_1^T & — \\ — & \sigma_2 v_2^T & — \\ & \vdots & \\ — & \sigma_k v_k^T & — \end{bmatrix}$$

Matrix multiplication can be seen as a sum of outer products of the columns of the first matrix and the rows of the second matrix.

$$X_k = \sum_{i=1}^{k} (\text{column } i \text{ of } U_k) \cdot (\text{row } i \text{ of } A)$$

$$X_k = \sum_{i=1}^{k} u_i (\sigma_i v_i^T)$$

$$X_k = \sum_{i=1}^{k} \sigma_i (u_i v_i^T)$$

Let's time the two methods of reconstruction to see the efficiency gain from using matrix multiplication.

```
k = 100

start_time1 = time.time()
X1 = reconstruct_image(U, S, Vt, k)
end_time1 = time.time()

start_time2 = time.time()
X2 = np.dot(U[:, :k], np.dot(np.diag(S[:k]), Vt[:k, :]))
end_time2 = time.time()

T1 = end_time1 - start_time1
T2 = end_time2 - start_time2
```

```
print(f"explicitly computing the outer products: {T1:.6f} seconds")
print(f"leveraging matrix multiplication: {T2:.6f} seconds")
print(f"speedup: {T1/T2:.2f}x")
```

```
explicitly computing the outer products: 1.508656 seconds
leveraging matrix multiplication: 0.011750 seconds
speedup: 128.40x
```

There are two equivalent ways of leveraging matrix multiplication. Because of the associativity of matrix multiplication, we can compute the product in two different orders:

1. First compute $B = \Sigma_k V_k^T$, then compute $X_k = U_k B$.
2. First compute $C = U_k \Sigma_k$, then compute $X_k = C V_k^T$.

For a tall-and-skinny matrix like our image (m>n), the first method is more efficient because it involves multiplying a smaller intermediate matrix B (of size $k \times n$) with U_k (of size $m \times k$). The second method would involve multiplying a larger intermediate matrix C (of size $m \times k$) with V_k^T (of size $k \times n$), which is less efficient. For our 2600x1600 image, the difference is tiny, but for larger datasets it can be significant.

SVD is such a common operation that most numerical computing libraries have highly optimized implementations. See below `sklearn`'s `TruncatedSVD`, which uses `scipy.sparse.linalg.svds` under the hood. It is designed to compute only the first `k` singular values and vectors, making it more efficient for large datasets where only a few singular values are needed.

```
start_time2b = time.time()
X2b = np.dot(np.dot(U[:, :k], np.diag(S[:k])), Vt[:k, :])
end_time2b = time.time()

svd = TruncatedSVD(n_components=100)
truncated_image = svd.fit_transform(image_array)
start_time3 = time.time()
X3 = svd.inverse_transform(truncated_image)
end_time3 = time.time()
```

```
T2b = end_time2b - start_time2b
T3 = end_time3 - start_time3

print(f"matrix multiplication, option 2: {T2b:.6f} seconds")
print(f"using sklearn's TruncatedSVD: {T3:.6f} seconds")
```

```
matrix multiplication, option 2: 0.068557 seconds
using sklearn's TruncatedSVD: 0.011218 seconds
```

28 SVD for regression

This tutorial is partly based on the following sources:

- Understanding Linear Regression using the Singular Value Decomposition by Thalles Silva.
- Brunton and Kutz's book, Data-Driven Science and Engineering, subsection 1.4 called "Pseudo-inverse, least-squares, and regression".

28.1 the problem with Ordinary Least Squares

Let's say I want to predict the weight of a person based on their height. I have the following data:

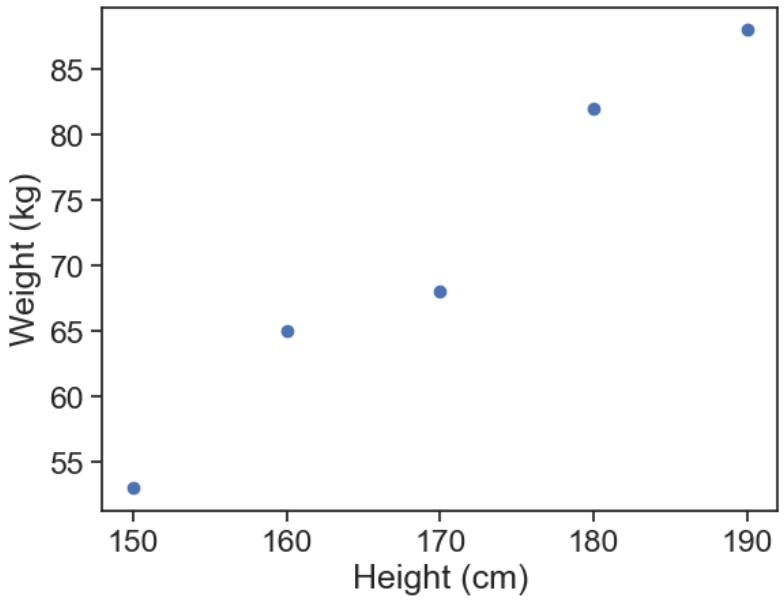
Height (cm)	Weight (kg)
150	53
160	65
170	68
180	82
190	88

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
```

```
h = np.array([150 ,160, 170, 180, 190])
w = np.array([53,  65,  68,  82,  88])
fig, ax = plt.subplots()
ax.scatter(h, w)
```

```
ax.set_xlabel("Height (cm)")
ax.set_ylabel("Weight (kg)")
```

```
Text(0, 0.5, 'Weight (kg)')
```



I can try to predict the weight using a linear model:

$$\text{weight} = \beta_0 + \beta_1 \cdot \text{height}. \tag{1}$$

In a general form, we can write this as:

$$X\beta = y, \tag{2}$$

where X is the design matrix, β is the vector of coefficients, and y is the vector of outputs (weights). This problem probably has no exact solution for β, because the design matrix X is not square (there are more data points than parameters). So we want to find the best approximation $\hat{\beta}$ that minimizes the error:

$$\hat{\beta} = \arg\min_{\beta} \|y - X\beta\|^2. \tag{3}$$

We know how to solve this, we use the equation we derived in the chapter "the geometry of regression":

$$\hat{\beta} = (X^T X)^{-1} X^T y, \tag{4}$$

For a linear model, the design matrix is:

$$X = \begin{bmatrix} | & | \\ \mathbf{1} & h \\ | & | \end{bmatrix} = \begin{bmatrix} 1 & 150 \\ 1 & 160 \\ 1 & 170 \\ 1 & 180 \\ 1 & 190 \end{bmatrix}. \tag{5}$$

What does the matrix $X^T X$ look like?

$$\begin{aligned} X^T X &= \begin{bmatrix} - & \mathbf{1}^T & - \\ - & h^T & - \end{bmatrix} \begin{bmatrix} | & | \\ \mathbf{1} & h \\ | & | \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{1}^T\mathbf{1} & \mathbf{1}^T h \\ h^T\mathbf{1} & h^T h \end{bmatrix} \\ &= \begin{bmatrix} 5 & 850 \\ 850 & 153000 \end{bmatrix} \end{aligned} \tag{6}$$

There's no problem inverting this matrix, so we can find the coefficient estimates $\hat{\beta}$ using the formula above.

Suppose now that we have a new predictor, the height of the person in inches. The design matrix now looks like this:

$$X = \begin{bmatrix} | & | & | \\ \mathbf{1} & h_{cm} & h_{inch} \\ | & | & | \end{bmatrix} \tag{7}$$

Obviously, the columns h_{cm} and h_{inch} are linearly dependent ($h_{cm} = a h_{inch}$). This means that the matrix $X^T X$ also has linearly dependent columns:

$$X^T X = \begin{bmatrix} - & \mathbf{1}^T & - \\ - & h_{cm}^T & - \\ - & h_{inch}^T & - \end{bmatrix} \begin{bmatrix} | & | & | \\ \mathbf{1} & h_{cm} & h_{inch} \\ | & | & | \end{bmatrix}$$
$$= \begin{bmatrix} \mathbf{1}^T\mathbf{1} & \mathbf{1}^T h_{cm} & \mathbf{1}^T h_{inch} \\ h_{cm}^T\mathbf{1} & h_{cm}^T h_{cm} & h_{cm}^T h_{inch} \\ h_{inch}^T\mathbf{1} & h_{inch}^T h_{cm} & h_{inch}^T h_{inch} \end{bmatrix} \tag{8}$$

Using the fact that $h_{cm} = a h_{inch}$, we can see that the second and third columns are linearly dependent. This means that the matrix $X^T X$ is not invertible, and we cannot use the formula above to find the coefficient estimates $\hat{\beta}$. What now?

This is an extreme case, but problems similar to this can happen in real life. For example, if we have two predictors that are highly correlated, the matrix $X^T X$ will be close to singular (not invertible). In this case, the coefficients β will be very sensitive to small changes in the data.

28.2 SVD to the rescue

Let's use Singular Value Decomposition (SVD) to find the coefficients β. SVD is a powerful technique that can handle multicollinearity and other issues in the data.

The SVD of a matrix X is given by:

$$X = U \Sigma V^T, \tag{9}$$

where U and V are orthogonal matrices and Σ is a diagonal matrix with singular values on the diagonal.

We can plug the SVD of X into the least squares problem, which is to find the $\hat{\beta}$ that best satisfies $X\hat{\beta} = \hat{y}$:

$$U \Sigma V^T \hat{\beta} = \hat{y}. \tag{10}$$

We define now the Moore-Penrose pseudo-inverse of X, which is given by:

$$X^+ = V\Sigma^+ U^T, \tag{11}$$

where Σ^+ is obtained by taking the reciprocal of the non-zero singular values in Σ and transposing the resulting matrix.

This pseudo-inverse has the following properties:

- $X^+X = I$. This means that X^+ is a left-inverse of X.
- XX^+ is a projection matrix onto the column space of X. In other words, left-multiplying XX^+ to any vector gives the projection of that vector onto the column space of X. In particular, $XX^+y = \hat{y}$.

  This is very similar to the property we used in the chapter "the geometry of regression", where we had $P_X y = \hat{y}$, with P_X being the projection matrix onto the column space of X. There, we found that

  $$\hat{\beta} = (X^T X)^{-1} X^T y,$$

  Left-multiplying both sides by X gives:

  $$X\hat{\beta} = X(X^T X)^{-1} X^T y = P_X y = \hat{y}.$$

  So we found that in the OLS case, $X(X^T X)^{-1} X^T$ is the projection matrix onto the column space of X. Here, we have a more general result that works even when $X^T X$ is not invertible: XX^+ is the projection matrix onto the column space of X.

We left-multiply Eq. (10) by X^+, and also substitute $\hat{y} = XX^+y$ as we just saw:

$$\begin{aligned} X^+(U\Sigma V^T\hat{\beta}) &= X^+XX^+y \\ (V\Sigma^+U^T)(U\Sigma V^T)\hat{\beta} &= X^+y \end{aligned} \tag{12}$$

The term $(V\Sigma^+U^T)(U\Sigma V^T)$ simplifies to the identity matrix, so we have:

$$\hat{\beta} = X^{+}y = V\Sigma^{+}U^{T}y. \tag{13}$$

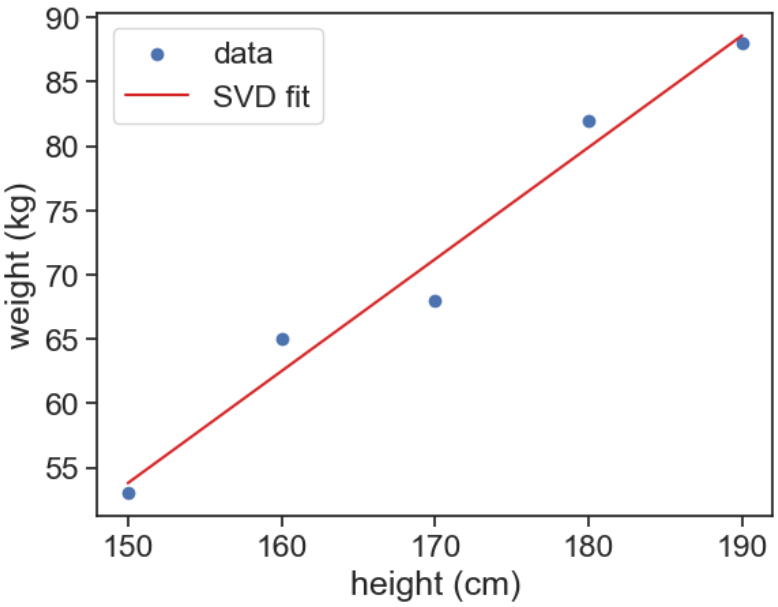
This is the formula we will use to find the coefficients $\hat{\beta}$ using SVD. This formula works even when $X^{T}X$ is not invertible, and it is more stable than the OLS formula.

28.3 in practice

Let's go back to the problematic example with height in cm and inches. We can use the SVD to find the coefficients $\hat{\beta}$.

```
# design matrix with intercept, height in cm and height in inches
conversion_factor = 2.54
X = np.vstack([np.ones(len(h)), h, h/conversion_factor]).T
# compute SVD of X
U, S, VT = np.linalg.svd(X, full_matrices=False)
Sigma = np.diag(S)
V = VT.T
# compute coefficients using SVD
y = w
Sigma_plus = np.zeros(Sigma.T.shape)
for i in range(len(S)):
    if S[i] > 1e-10:  # avoid division by zero
        Sigma_plus[i, i] = 1 / S[i]
X_plus = V @ Sigma_plus @ U.T
beta_hat = X_plus @ y
# make predictions
y_hat = X @ beta_hat
```

```
fig, ax = plt.subplots()
ax.scatter(h, w, label="data")
# plot predictions
ax.plot(h, y_hat, color="tab:red", label="SVD fit")
ax.legend()
ax.set_xlabel("height (cm)")
ax.set_ylabel("weight (kg)");
```


data
SVD fit
weight (kg)
height (cm)
90
85
80
75
70
65
60
55
150
160
170
180
190

Part IX

decision trees

29 CART: classification

29.1 a jungle party

Imagine you're throwing a party in the jungle. You know you have three types of guests—koalas, foxes, and bonobos—but you don't know who is who. To make sure you serve the right food, you need to automatically figure out which animal is which. Koalas only eat eucalyptus leaves, foxes prefer meat, and bonobos love fruit, so it's important to get it right!

To solve this problem, you'll use a decision tree. You've gathered some data on your past animal guests, and for each one, you have their height and weight, as well as their species (their label). Your goal is to build a system that can learn from this historical data to correctly classify a new, unlabeled animal based on its height and weight alone.

The data is structured as:

- **Features:** height and weight, continuous, numerical features.
- **Categories (Classes):** Koala, Fox, Bonobo

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
```

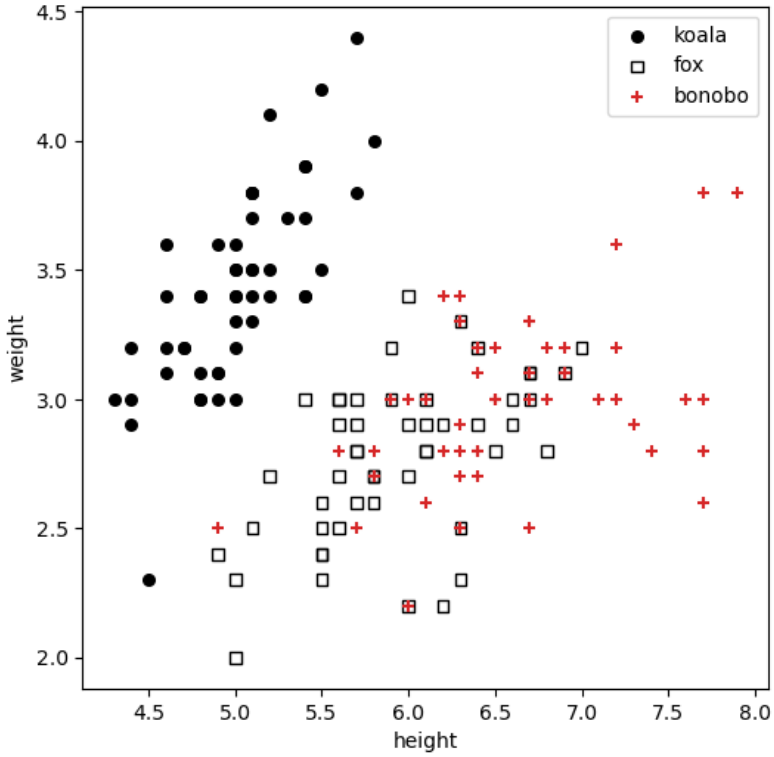
We are using the famous Iris dataset structure, but pretending they are animals for this example.

```
iris = load_iris()
X = iris.data[:, [0, 1]]
y = iris.target
```

```
fig, ax = plt.subplots(figsize=(6, 6))
markers = ['o', 's', '+']
colors = ['black', 'None', 'tab:red']
iris.target_names = ['koala', 'fox', 'bonobo']
iris.feature_names = ['height', 'weight', 'age']
for i, marker in enumerate(markers):
    ax.scatter(X[y == i, 0], X[y == i, 1],
               c=colors[i],
               edgecolor='k', s=30, marker=marker, label=iris.target_names[i])
ax.legend()
ax.set_xlabel(iris.feature_names[0])
ax.set_ylabel(iris.feature_names[1])
```

```
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/3638754974.py:8: UserWarning:
  ax.scatter(X[y == i, 0], X[y == i, 1],
```

```
Text(0, 0.5, 'weight')
```



We will use the CART (Classification and Regression Trees) algorithm to build a decision tree. There are many other methods, but CART is one of the most popular, it's important to know it well.

29.2 the split

We will cut the data into two parts along one of the features. We will try **all possible cut thresholds** for each of the features and see which one gives us the best split. Ideally, we want to end up with two groups, "left" and "right", that are as "pure" as possible, meaning that each group contains animals of only one species. It might not be possible to get perfect homogeneity, so we will need to quantify how good a split is. We will use two different metrics to evaluate the quality of a split:

- **Information Gain (based on Entropy):** This measures how much uncertainty in the data is reduced after the split. A higher information gain means a better split.

$$IG(T, X) = \underbrace{H(T)}_{\text{original entropy}} - \underbrace{\frac{N_{\text{left}}}{N} H(T_{\text{left}})}_{\text{left group}} - \underbrace{\frac{N_{\text{right}}}{N} H(T_{\text{right}})}_{\text{right group}}$$

  where T_{left} and T_{right} are the subsets after the split, and N denotes the total number of samples in a (sub) dataset.

- **Gini Impurity:** This measure is often explained as the impurity of a split. A better point of view is to interpret is as the probability of misclassification. If we picked an element at random from the dataset, what is the probability that we would misclassify it if we labeled it according to the distribution of classes in the dataset? The probability of picking an element of class i is P_i, and we would misclassify it with probability $1 - P_i$. So the total probability of misclassification is $P_i(1 - P_i)$. If we sum this over all classes, we get the Gini Impurity:

$$G(T) = \sum_{i=1}^{C} P_i(1 - P_i) = 1 - \sum_{i=1}^{C} P_i^2$$

Now let's take our dataset and try to find the best split using both metrics.

29.2.1 entropy

We will start with the **entropy** metric.

```
def calculate_gini(y_subset):
    """Calculates the Gini Impurity for a given subset of class labels."""
    # If the subset is empty, there's no impurity.
    if len(y_subset) == 0:
        return 0.0

    # Get the counts of each unique class in the subset.
    unique_classes, counts = np.unique(y_subset, return_counts=True)
```

```
    # Calculate the probability of each class.
    probabilities = counts / len(y_subset)

    # Gini Impurity formula: 1 - sum(p^2) for each class
    return 1.0 - np.sum(probabilities**2)

def calculate_entropy(y_subset):
    """Calculates the Entropy for a given subset of class labels."""
    if len(y_subset) == 0:
        return 0.0
    unique_classes, counts = np.unique(y_subset, return_counts=True)
    probabilities = counts / len(y_subset)
    epsilon = 1e-9
    return -np.sum(probabilities * np.log2(probabilities + epsilon))

def find_best_split(X_feature, y, criterion='entropy'):
    """
    Finds the best split point for a single feature based on a specified criterion.
    Returns the best threshold and the score (Information Gain or Gini Impurity) after the spli
    """
    # Get the unique values of the feature to consider as split points.
    unique_values = np.sort(np.unique(X_feature))
    differences = np.diff(unique_values)
    threshold_candidates = unique_values[:-1] + differences / 2
    if criterion == 'entropy':
        initial_score = calculate_entropy(y)
        best_score = 0.0  # We want to maximize Information Gain
        criterion_function = calculate_entropy
    elif criterion == 'gini':
        best_score = 1.0  # We want to minimize Gini Impurity
        criterion_function = calculate_gini
    else:
        raise ValueError("Criterion must be 'entropy' or 'gini'")

    best_threshold = None

    # Iterate through each unique value as a potential split point
    for threshold in threshold_candidates:
        # Split the data into two groups based on the threshold
        condition = X_feature <= threshold
```

```
        y_left = y[condition]    # condition is True
        y_right = y[~condition]  # condition is False

        score_left = criterion_function(y_left)
        score_right = criterion_function(y_right)
        fraction_left = len(y_left) / len(y)
        fraction_right = len(y_right) / len(y)
        weighted_score = fraction_left * score_left + fraction_right * score_right

        if criterion == 'entropy':
            information_gain = initial_score - weighted_score
            # If this split is the best so far, save it!
            if information_gain > best_score:  # Max Information Gain
                best_score = information_gain
                best_threshold = threshold

        elif criterion == 'gini':
            # If this split is the best so far, save it!
            if weighted_score < best_score:  # Min Gini Impurity
                best_score = weighted_score
                best_threshold = threshold

    return best_threshold, best_score

def quantify_all_splits(X_feature, y, criterion='entropy'):
    """
    Finds the best split point for a single feature based on a specified criterion.
    Returns the best threshold and the score (Information Gain or Gini Impurity) after the spli
    """
    # Get the unique values of the feature to consider as split points.
    unique_values = np.sort(np.unique(X_feature))
    differences = np.diff(unique_values)
    threshold_candidates = unique_values[:-1] + differences / 2
    score_list = []
    if criterion == 'entropy':
        initial_score = calculate_entropy(y)
        best_score = 0.0  # We want to maximize Information Gain
        criterion_function = calculate_entropy
    elif criterion == 'gini':
        best_score = 1.0  # We want to minimize Gini Impurity
```

```
        criterion_function = calculate_gini
    else:
        raise ValueError("Criterion must be 'entropy' or 'gini'")

    best_threshold = None

    # Iterate through each unique value as a potential split point
    for threshold in threshold_candidates:
        # Split the data into two groups based on the threshold
        condition = X_feature <= threshold
        y_left = y[condition]    # condition is True
        y_right = y[~condition]  # condition is False

        score_left = criterion_function(y_left)
        score_right = criterion_function(y_right)
        fraction_left = len(y_left) / len(y)
        fraction_right = len(y_right) / len(y)
        weighted_score = fraction_left * score_left + fraction_right * score_right

        if criterion == 'entropy':
            information_gain = initial_score - weighted_score
            score_list.append((threshold, information_gain))
            # If this split is the best so far, save it!

        elif criterion == 'gini':
            score_list.append((threshold, weighted_score))

    return np.array(score_list)
```

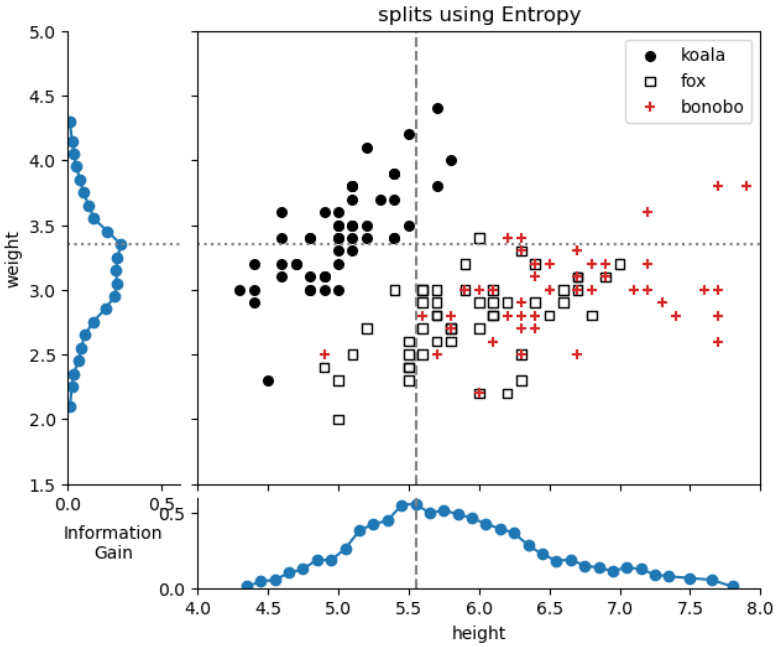
```
fig = plt.figure(1, figsize=(8, 6))
gs = gridspec.GridSpec(2, 2, width_ratios=[0.2,1], height_ratios=[1,0.2])
gs.update(left=0.16, right=0.86,top=0.88, bottom=0.13, hspace=0.05, wspace=0.05)

ax_main = plt.subplot(gs[0, 1])
ax_height = plt.subplot(gs[1, 1])
ax_weight = plt.subplot(gs[0, 0])

for i, marker in enumerate(markers):
    ax_main.scatter(X[y == i, 0], X[y == i, 1],
                    c=colors[i],
```

```
                    edgecolor='k', s=30, marker=marker, label=iris.target_names[i])

split_feature_height = quantify_all_splits(X[:, 0], y, criterion='entropy')
split_feature_weight = quantify_all_splits(X[:, 1], y, criterion='entropy')

ax_height.plot(split_feature_height[:, 0], split_feature_height[:, 1], marker='o')
ax_weight.plot(split_feature_weight[:, 1], split_feature_weight[:, 0], marker='o')

best_height_split = np.argmax(split_feature_height[:, 1])
best_weight_split = np.argmax(split_feature_weight[:, 1])

ax_main.axvline(split_feature_height[best_height_split, 0], color='gray', linestyle='--')
ax_height.axvline(split_feature_height[best_height_split, 0], color='gray', linestyle='--')
ax_main.axhline(split_feature_weight[best_weight_split, 0], color='gray', linestyle=':')
ax_weight.axhline(split_feature_weight[best_weight_split, 0], color='gray', linestyle=':')

ax_main.set(xticklabels=[],
            yticklabels=[],
            title="splits using Entropy",
            xlim=(4, 8),
            ylim=(1.5, 5.0)
            )
ax_main.legend()

ax_height.set(ylim=(0, 0.6),
              xlim=(4, 8),
              xlabel="height"
              )
ax_weight.set(xlim=(0, 0.6),
              ylim=(1.5, 5.0),
                ylabel="weight",
              )

ax_weight.spines['top'].set_visible(False)
ax_weight.spines['right'].set_visible(False)
ax_height.spines['top'].set_visible(False)
ax_height.spines['right'].set_visible(False)

ax_height.text(-0.15, 0.5, "Information\nGain", rotation=0, va='center', ha='center', transfor
plt.tight_layout()
```

```
print(f"Height: highest Information Gain: {split_feature_height[best_height_split, 1]:.2f} at h
print(f"Weight: highest Information Gain: {split_feature_weight[best_weight_split, 1]:.2f} at w
```

```
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/2101311549.py:11: UserWarning
  ax_main.scatter(X[y == i, 0], X[y == i, 1],
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/2101311549.py:52: UserWarning
  plt.tight_layout()
```

```
Height: highest Information Gain: 0.56 at height=5.55
Weight: highest Information Gain: 0.28 at weight=3.35
```



- The best split for the 'height' feature is at a height 5.55, with IG=0.56.
- The best split for the 'weight' feature is at a weight 3.35, with IG=0.28.

Using the Entropy metric, the first split will be on the 'height' feature at a height of 5.55, since it has the highest information gain.

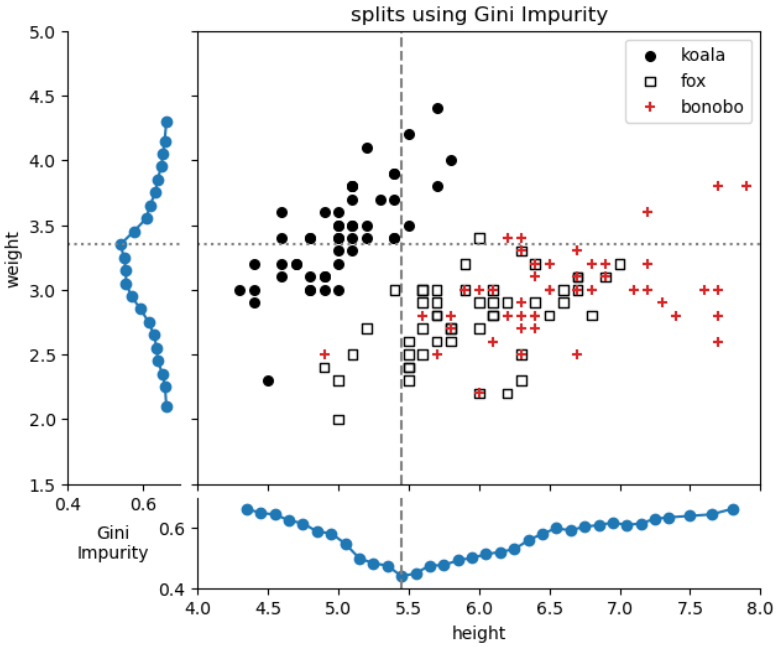
29.2.2 Gini impurity

Now let's try the **Gini impurity** metric.

```
fig = plt.figure(1, figsize=(8, 6))
gs = gridspec.GridSpec(2, 2, width_ratios=[0.2,1], height_ratios=[1,0.2])
gs.update(left=0.16, right=0.86,top=0.88, bottom=0.13, hspace=0.05, wspace=0.05)

ax_main = plt.subplot(gs[0, 1])
ax_height = plt.subplot(gs[1, 1])
ax_weight = plt.subplot(gs[0, 0])

for i, marker in enumerate(markers):
    ax_main.scatter(X[y == i, 0], X[y == i, 1],
                    c=colors[i],
                    edgecolor='k', s=30, marker=marker, label=iris.target_names[i])

split_feature_height = quantify_all_splits(X[:, 0], y, criterion='gini')
split_feature_weight = quantify_all_splits(X[:, 1], y, criterion='gini')

ax_height.plot(split_feature_height[:, 0], split_feature_height[:, 1], marker='o')
ax_weight.plot(split_feature_weight[:, 1], split_feature_weight[:, 0], marker='o')

best_height_split = np.argmin(split_feature_height[:, 1])
best_weight_split = np.argmin(split_feature_weight[:, 1])

ax_main.axvline(split_feature_height[best_height_split, 0], color='gray', linestyle='--')
ax_height.axvline(split_feature_height[best_height_split, 0], color='gray', linestyle='--')
ax_main.axhline(split_feature_weight[best_weight_split, 0], color='gray', linestyle=':')
ax_weight.axhline(split_feature_weight[best_weight_split, 0], color='gray', linestyle=':')

ax_main.set(xticklabels=[],
            yticklabels=[],
            title="splits using Gini Impurity",
            xlim=(4, 8),
            ylim=(1.5, 5.0)
            )
ax_main.legend()

ax_height.set(ylim=(0.4, 0.70),
              xlim=(4, 8),
```

```
              xlabel="height"
              )
ax_weight.set(xlim=(0.4, 0.70),
              ylim=(1.5, 5.0),
                ylabel="weight",
              )

ax_weight.spines['top'].set_visible(False)
ax_weight.spines['right'].set_visible(False)
ax_height.spines['top'].set_visible(False)
ax_height.spines['right'].set_visible(False)

ax_height.text(-0.15, 0.5, "Gini\nImpurity", rotation=0, va='center', ha='center', transform=ax
plt.tight_layout()

print(f"Height: lowest Gini Impurity: {split_feature_height[best_height_split, 1]:.2f} at heig
print(f"Weight: lowest Gini Impurity: {split_feature_weight[best_weight_split, 1]:.2f} at weig
```

```
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/3659548085.py:11: UserWarning
  ax_main.scatter(X[y == i, 0], X[y == i, 1],
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/3659548085.py:52: UserWarning
  plt.tight_layout()

Height: lowest Gini Impurity: 0.44 at height=5.45
Weight: lowest Gini Impurity: 0.54 at weight=3.35
```



- The best split for the 'height' feature is at a height 5.45, with Gini=0.44.
- The best split for the 'weight' feature is at a weight 3.35, with Gini=0.54.

Using the Gini Impurity metric, the first split will be on the 'height' feature at a height of 5.45, since it has the lowest Gini impurity.

29.3 successive splits

The same idea used for the first split is then applied to each of the subsets, recursively. The process continues until one of the stopping criteria is met, such as:

- All samples in a node belong to the same class.
- The maximum depth of the tree is reached.
- The number of samples in a node is less than a predefined minimum.

I find it useful to visualize the decision tree as a series of splits in the feature space:

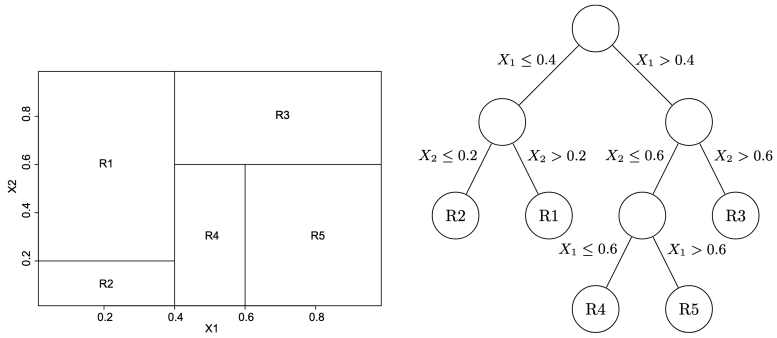


Fig. 1 Illustration of a decision tree partition of the predictor space in five regions and the associated decision tree

Source: "Predicting University Students' Academic Success and Major Using Random Forests", by Cédric Beaulac and Jeffrey S. Rosenthal

Splitting the feature space with vertical and horizontal lines reminds me of a classic 1990's computer game, JazzBall. Check out a video of the game here, and see it reminds you of the basic algorithm discussed so far.

29.4 sklearn tree DecisionTreeClassifier

We will now use the `DecisionTreeClassifier` from `sklearn.tree` to build a decision tree classifier. Let's restrict the maximum depth of the tree to 3, so we can visualize it easily. All the hard work was already coded for use, it'll take us only two lines of code to create and fit the model.

```
# using gini impurity
classifier_gini = DecisionTreeClassifier(criterion='gini', max_depth=3)#, random_state=42)
classifier_gini.fit(X, y)

# using entropy
classifier_entropy = DecisionTreeClassifier(criterion='entropy', max_depth=3)#, random_state=4
classifier_entropy.fit(X, y)
```

criterion	'entropy'

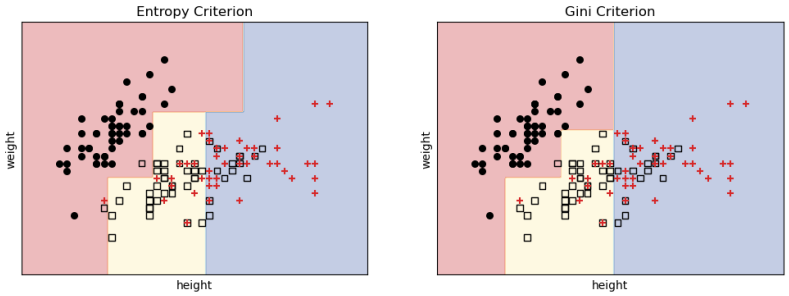
splitter	'best'
max_depth	3
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	None
random_state	None
max_leaf_nodes	None
min_impurity_decrease	0.0
class_weight	None
ccp_alpha	0.0
monotonic_cst	None

Now let's visualize the results.

```
# Helper function to plot decision boundaries
def plot_decision_boundaries(ax, model, X, y, title):
    """
    Plots the decision boundaries for a given classifier.
    """
    # Define a mesh grid to color the background based on predictions
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Predict the class for each point in the mesh grid
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the colored regions
    ax.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.RdYlBu)

    # Set labels and title
    ax.set_title(title)
    ax.set_xlabel(iris.feature_names[0])
    ax.set_ylabel(iris.feature_names[1])
    ax.set_xticks(())
    ax.set_yticks(())
```

```
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
plot_decision_boundaries(ax[0], classifier_entropy, X, y, "Entropy Criterion")
for i, marker in enumerate(markers):
    ax[0].scatter(X[y == i, 0], X[y == i, 1],
                  c=colors[i],
                  edgecolor='k', s=30, marker=marker, label=iris.target_names[i])

plot_decision_boundaries(ax[1], classifier_gini, X, y, "Gini Criterion")
for i, marker in enumerate(markers):
    ax[1].scatter(X[y == i, 0], X[y == i, 1],
                  c=colors[i],
                  edgecolor='k', s=30, marker=marker, label=iris.target_names[i])
```

```
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/759299291.py:5: UserWarning: 
  ax[0].scatter(X[y == i, 0], X[y == i, 1],
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/759299291.py:11: UserWarning:
  ax[1].scatter(X[y == i, 0], X[y == i, 1],
```



Just by using different criteria, we get different boundaries! We can now predict the species of a new animal based on its height and weight.

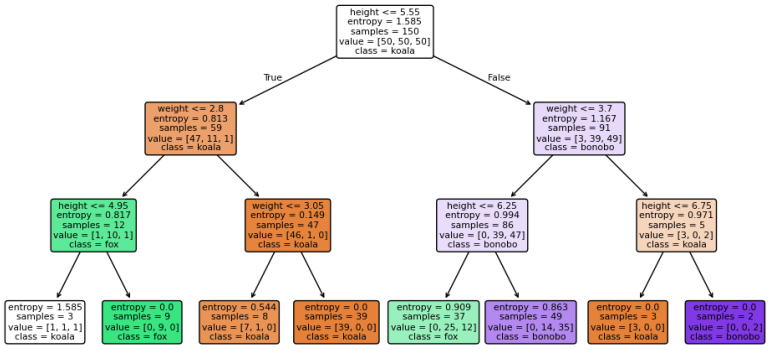
For example, an animal with height 6.3 and weight 4.0 would be classified as:

```
# Predict the species of a new animal with height 5.7 and weight 4.0 using the entropy-based cl
sample = np.array([[6.3, 4.0]])
predicted_class_entropy = classifier_entropy.predict(sample)
predicted_class_gini = classifier_gini.predict(sample)
print(f"Predicted species (Entropy): {iris.target_names[predicted_class_entropy[0]]}")
print(f"Predicted species (Gini): {iris.target_names[predicted_class_gini    [0]]}")
```

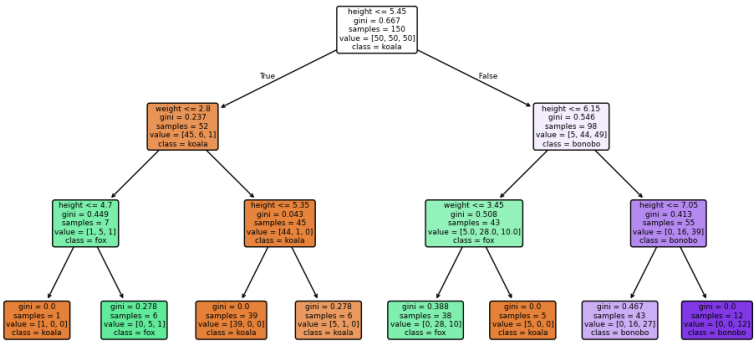
```
Predicted species (Entropy): koala
Predicted species (Gini): bonobo
```

See also the decision trees for each criterion.

```
fig, ax = plt.subplots(1, 1, figsize=(12, 6))
plot_tree(classifier_entropy,
          filled=True,
          rounded=True,
          class_names=iris.target_names,
          feature_names=[iris.feature_names[0], iris.feature_names[1]],
          ax=ax);
```




```
fig, ax = plt.subplots(1, 1, figsize=(12, 6))
plot_tree(classifier_gini,
          filled=True,
          rounded=True,
          class_names=iris.target_names,
          feature_names=[iris.feature_names[0], iris.feature_names[1]],
          ax=ax);
```



29.5 overfitting

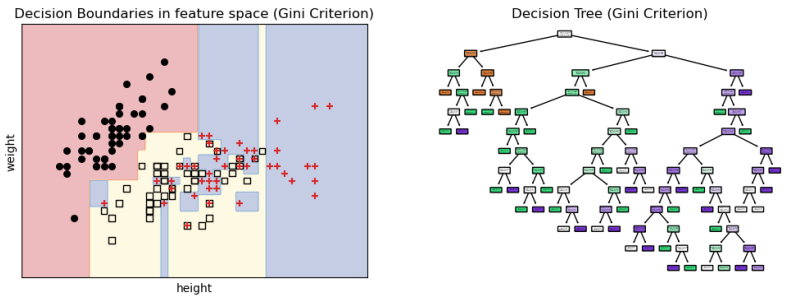
What would happen if we chose to grow our decision tree until all leaves are pure? This would lead to a very complex tree that perfectly classifies the training data, but might not generalize well to new, unseen data. This is known as overfitting.

```
classifier_gini = DecisionTreeClassifier(criterion='gini')
classifier_gini.fit(X, y)
```

criterion	'gini'
splitter	'best'
max_depth	None
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	None
random_state	None
max_leaf_nodes	None
min_impurity_decrease	0.0
class_weight	None
ccp_alpha	0.0
monotonic_cst	None

```
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
plot_decision_boundaries(ax[0], classifier_gini, X, y, "Decision Boundaries in feature space (
for i, marker in enumerate(markers):
    ax[0].scatter(X[y == i, 0], X[y == i, 1],
                  c=colors[i],
                  edgecolor='k', s=30, marker=marker, label=iris.target_names[i])

plot_tree(classifier_gini,
          filled=True,
          rounded=True,
          class_names=iris.target_names,
          feature_names=[iris.feature_names[0], iris.feature_names[1]],
          ax=ax[1]);
ax[1].set_title("Decision Tree (Gini Criterion)");
fig.savefig("decision_boundaries_no_max_depth.png", dpi=300)
```

```
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_10048/359429702.py:5: UserWarning: 
  ax[0].scatter(X[y == i, 0], X[y == i, 1],
```



We can see that some of the regions are very small and specific to the training data points. This means that the model has learned not only the underlying patterns in the data but also the noise and outliers, which can lead to poor performance on new data.

To avoid overfitting, we can use techniques such as:

- Setting a maximum depth. We limit how deep the tree can grow. Read about `max_depth`.
- Setting a minimum number of samples required to split a node. Read about `min_samples_split`.

- Post-pruning: Cutting back the tree after it has been grown to remove branches that do not provide significant predictive power. Read about `cost_complexity_pruning_path`.

Other machine learning algorithms, such as Random Forests and Gradient Boosted Trees, use ensemble methods that combine multiple decision trees to improve performance and reduce overfitting.

30 CART: regression

In the previous classification example, we saw how to put each animal into a category (koala, fox, bonobo) based on its features (height and weight). If you haven't read it yet, go back to the classification example.

In regression, instead of putting things into categories, we predict a continuous value. Building on the previous example, let's say we want to predict the age of an animal based on its height and weight.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeRegressor, plot_tree
```

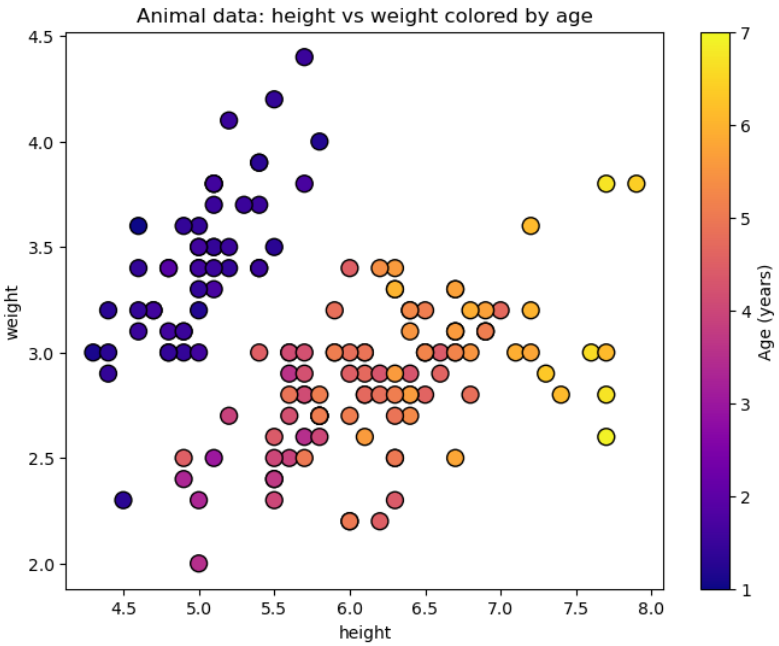
Again, we are using the famous Iris dataset structure:

- sepal length (cm) → height
- sepal width (cm) → weight
- petal length (cm) → age

```
iris = load_iris()
X = iris.data[:, [0, 1]]
y = iris.data[:,[2]].flatten()
iris.feature_names = ['height', 'weight', 'age']
```

```
fig, ax = plt.subplots(figsize=(8, 6))
scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap='plasma', edgecolor='k', s=100, vmin=1, vmax=
ax.set_xlabel(iris.feature_names[0])
ax.set_ylabel(iris.feature_names[1])
ax.set(xlabel='height',
       ylabel='weight',
```

```
       title="Animal data: height vs weight colored by age")
fig.colorbar(scatter, label='Age (years)')
```



30.1 the split

We will follow a similar procedure to split the data along the features, but this time our target variable is continuous (age) instead of categorical (animal type). In classification we wanted to have leaves as pure as possible, and we quantified that either with Gini impurity or entropy. In regression, we want to minimize the variance of the target variable within each leaf. In our example, this means that we want to split the data in a way that the ages of the animals in each leaf are as similar as possible.

The cost function we will use to evaluate the quality of a split is the Weighted Mean Squared Error (MSE):

$$J(j, s) = \frac{N_{\text{left}}}{N_{\text{total}}} \cdot \text{MSE}_{\text{left}} + \frac{N_{\text{right}}}{N_{\text{total}}} \cdot \text{MSE}_{\text{right}},$$

where N_{left} and N_{right} are the number of samples in the left and right child nodes, respectively, and N_{total} is the total number of samples in the parent node. MSE_{left} and $\text{MSE}_{\text{right}}$ are the mean squared errors of the target variable in the left and right child nodes:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \bar{y})^2,$$

where y_i are the target values in the node, $\bar{y}$ is the mean target value in the node, and N is the number of samples in the node.

The cost function is weighted by the number of samples in each child node to account for the fact that larger nodes have a greater impact on the overall variance.

So how do we know which split is the best? We will evaluate all possible split thresholds s along all features j and choose the one that minimizes the Weighted MSE. In a mathematical language:

$$(j^*, s^*) = \arg\min_{j,s} J(j, s).$$

30.2 sklearn tree DecisionTreeRegressor

We will use the `DecisionTreeRegressor` class from `sklearn.tree` to build our regression tree.

```
regressor = DecisionTreeRegressor(max_depth=3)
regressor.fit(X, y)
```

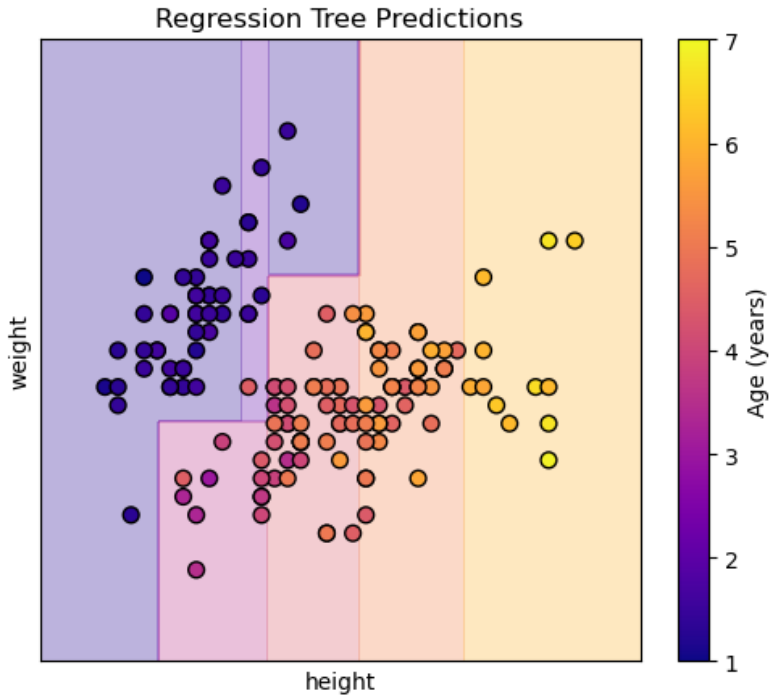
criterion	'squared_error'
splitter	'best'
max_depth	3
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	None

random_state	None
max_leaf_nodes	None
min_impurity_decrease	0.0
ccp_alpha	0.0
monotonic_cst	None

```
# Helper function to plot decision boundaries
def plot_decision_boundaries(ax, model, X, y, title):
    """
    Plots the decision boundaries for a given classifier.
    """
    # Define a mesh grid to color the background based on predictions
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    # Predict the class for each point in the mesh grid
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the colored regions
    ax.contourf(xx, yy, Z, alpha=0.3, cmap='plasma', vmin=1, vmax=7)

    # Set labels and title
    ax.set_title(title)
    ax.set_xlabel(iris.feature_names[0])
    ax.set_ylabel(iris.feature_names[1])
    ax.set_xticks(())
    ax.set_yticks(())

from mpl_toolkits.mplot3d import Axes3D

# Helper function to plot decision boundaries
def plot_decision_boundaries_3d(ax, model, X, y, title):
    # Define a mesh grid
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
```

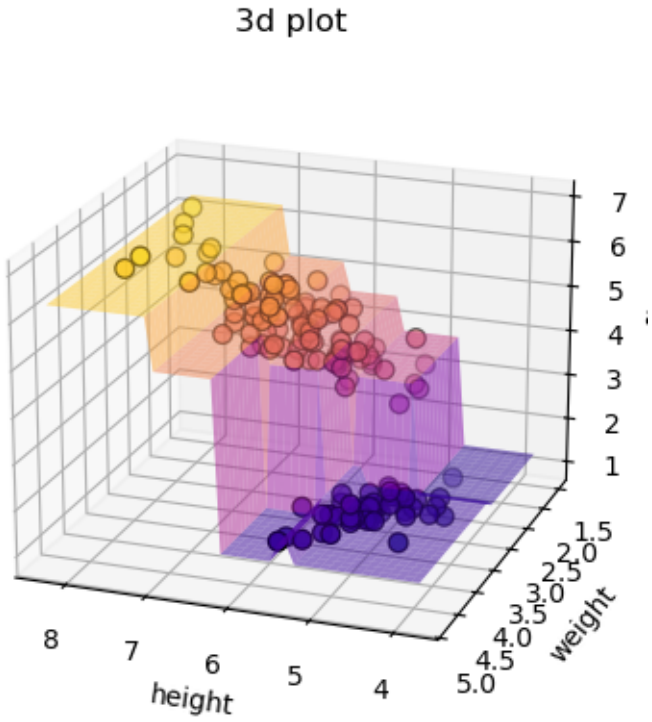
```
                         np.arange(y_min, y_max, 0.1))

    # Predict the values for each point in the mesh grid
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot the surface
    surf = ax.plot_surface(xx, yy, Z, alpha=0.5, cmap='plasma', vmin=1, vmax=7)

    # Plot the actual data points
    scatter = ax.scatter(X[:, 0], X[:, 1], y, c=y, cmap='plasma', edgecolor='k', s=50, vmin=1,

    # Set labels and title
    ax.set_xlabel(iris.feature_names[0])
    ax.set_ylabel(iris.feature_names[1])
    ax.set_zlabel(iris.feature_names[2])
    ax.set_title(title)

    # fig.colorbar(scatter, ax=ax, label='Age (years)', shrink=0.5)

    return fig, ax
```

```
fig, ax = plt.subplots(figsize=(6, 5))
plot_decision_boundaries(ax, regressor, X, y, "Regression Tree Predictions")
scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap='plasma', edgecolor='k', s=50, vmin=1, vmax=7)
fig.colorbar(scatter, label='Age (years)')
```



Each region (leaf) of the tree will predict the mean age of the training samples that fall into that region. Visualizing this in 3d shows us steps, because the regression tree creates a **piecewise constant** approximation of the target variable (age) over the feature space (height and weight).

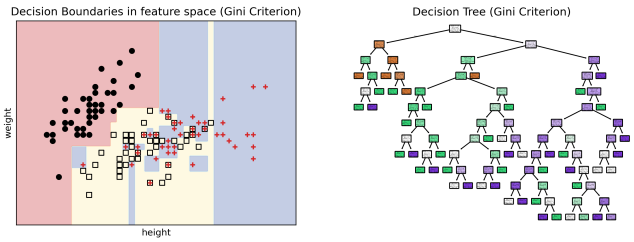
```
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(121, projection='3d')
plot_decision_boundaries_3d(ax, regressor, X, y, "3d plot")
ax.view_init(elev=20, azim=110)
```



The same consideration regarding overfitting applies here as in classification.

31 random forest

The motivation behind random forests is to avoid the weird regions in the feature space that a single decision tree might create. In the tutorial for classification with CART, we saw this:



The small blue regions inside the yellow region are highly undesirable. The decision tree learned very well the data it was given, but it will probably not generalize well to new data points. Random forests solve this problem in three steps.

31.1 step 1: bootstrap sampling

Instead of running the decision tree algorithm once on the entire training set, we run it multiple times on different bootstrap samples of the training set. We already learned about bootstrap sampling in the empirical confidence interval tutorial. In a nutshell, if we have a data set with N data points, we create a bootstrap sample by sampling N data points **with replacement** from the original data set. This means that some data points will appear multiple times in the bootstrap sample, while others will not appear at all. We can choose how many bootstrap samples we want to create. The default used by `sklearn`'s `RandomForestRegressor` is 100 (this argument is called `n_estimators`).

What fraction of the dataset will a given bootstrap sample contain, on average? The probability of choosing a specific data point in one draw is $1/N$. Therefore, the probability of **not** choosing that data point in one draw is $1 - 1/N$. If we draw N times with replacement, the probability of never choosing that data point is

$$\left(1 - \frac{1}{N}\right)^N$$

As N becomes large…

$$\lim_{N \to \infty} \left(1 - \frac{1}{N}\right)^N = e^{-1} \approx 0.37.$$

This means that, on average, a bootstrap sample will contain about 63% of the original data points (because about 37% of them will not be chosen at all).

This follows from the definition of the exponential function:

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n,$$

just set $x = -1$.

31.2 step 2: random feature selection

When training each decision tree on a bootstrap sample, we also randomly select a subset of the features to consider **for each split in the tree**. For example, if we have 10 features in total, we might randomly select 3 of them to consider for each split. This further increases the diversity among the trees in the forest, which helps to reduce overfitting.

Why is this important? Imagine a dataset where feature number 1 is very strongly correlated with the target variable. In that case, most decision trees will likely use that feature for the top split, leading to similar trees and less diversity in the forest.

As a rule of thumb, we typically use the square root or the logarithm (base 2) of the total number of features as the number of features to consider for each split. The argument in `sklearn`'s `RandomForestRegressor` that controls this is called `max_features`, and its default value is 1.0. This means that if we don't specify anything, it will use 100% of the features in

each split, and we will not have "feature decorrelation". In the documentation, search for max_features to find more details.

31.3 step 3: bagging

Finally, to make a prediction for a new data point, we pass it through each of the decision trees in the forest and average their predictions (for regression) or take a majority vote (for classification). This process is called "bagging" (short for **b**ootstrap **agg**regat**ing**). By averaging the predictions of multiple trees, we can reduce the variance of the model and improve its generalization performance.

31.4 example: iris dataset

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.ensemble import RandomForestRegressor
```

```
iris = load_iris()

# 1. Prepare Data (3 Inputs, 1 Output)
# columns: 0=SepalLen, 1=SepalWid, 2=PetalLen, 3=PetalWid
X_full = iris.data[:, [0, 1, 2]]  # 3 features
y_full = iris.data[:, 3]          # Target: Petal Width

# 2. Train Models
tree_model = DecisionTreeRegressor(random_state=42)
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)

tree_model.fit(X_full, y_full)
rf_model.fit(X_full, y_full)

# 3. Compare Errors (The numerical proof)
```

```
print(f"Single Tree Score: {tree_model.score(X_full, y_full):.3f}")
print(f"Random Forest Score: {rf_model.score(X_full, y_full):.3f}")

# X = iris.data[:, [0, 1]]
# y = iris.data[:,[2]].flatten()
# iris.feature_names = ['height', 'weight', 'age']
```

```
Single Tree Score: 0.999
Random Forest Score: 0.991
```

```
from sklearn.model_selection import train_test_split

# 1. Split the data (80% for training, 20% for testing)
X_train, X_test, y_train, y_test = train_test_split(X_full, y_full, test_size=0.2, random_state

# 2. Train on ONLY the training set
tree_model.fit(X_train, y_train)
rf_model.fit(X_train, y_train)

# 3. Test on the unseen data
print(f"Single Tree Test Score:   {tree_model.score(X_test, y_test):.3f}")
print(f"Random Forest Test Score: {rf_model.score(X_test, y_test):.3f}")
```

```
Single Tree Test Score:   0.845
Random Forest Test Score: 0.931
```

Part X

miscellaneous

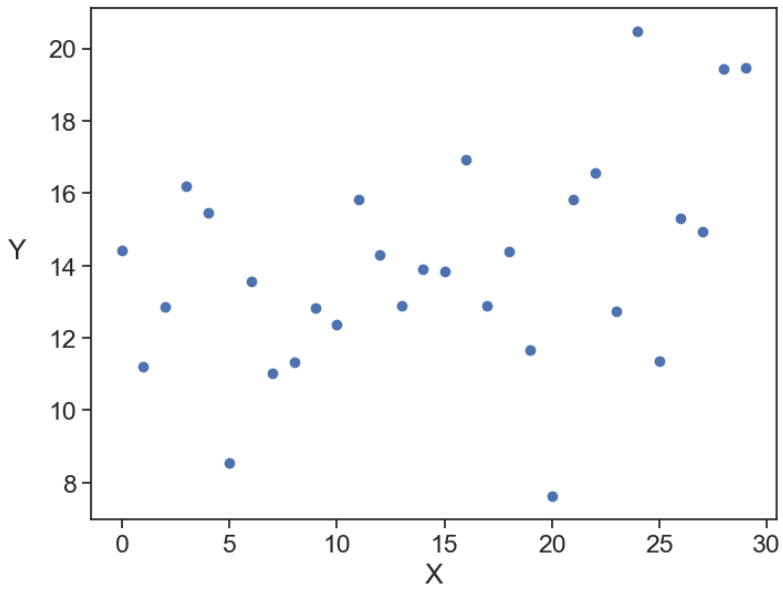
32 trend test

How to determine if there is a trend (positive or negative) between two variables?

```
import numpy as np
from scipy.stats import linregress
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
```

```
np.random.seed(0) # for reproducibility
x = np.arange(30)
y = (0.2 * x) + 10 + np.random.normal(loc=0, scale=2.5, size=30)

fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(x, y)
ax.set(xlabel='X')
ax.set_ylabel('Y', rotation=0, labelpad=20)
```

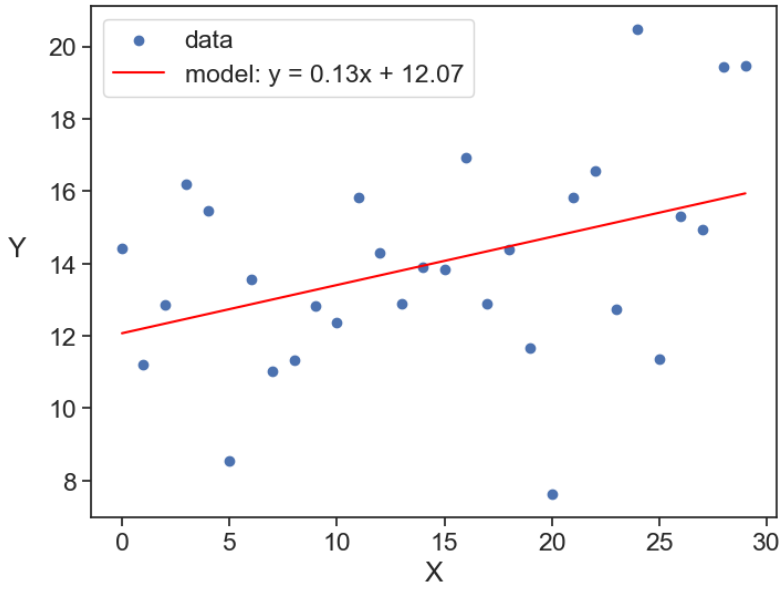
```
Text(0, 0.5, 'Y')
```



32.1 linear regression

One of the simplest ways to determine if there is a trend between two variables is to use linear regression.

```
slope, intercept, _, _, _ = linregress(x, y)
y_hat = slope * x + intercept

fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(x, y, label='data')
ax.plot(x, y_hat, color='red', label=f'model: y = {slope:.2f}x + {intercept:.2f}')
ax.set(xlabel='X')
ax.set_ylabel('Y', rotation=0, labelpad=20)
ax.legend()
```



Great, we found that there is a positive slope, its value is 0.13. It this enough to say that there is a trend?

We need to determine if the slope is significantly different from zero. For that we can use a t-test.

Null Hypothesis: The slope is equal to zero (no trend).
Alternative Hypothesis: The slope is not equal to zero (there is a trend).

The t-statistic is calculated as:

$$t = \frac{\text{slope} - 0}{SE_{\text{slope}}},$$

where SE_{slope} is the standard error of the slope, and it is given by:

$$SE_{\text{slope}} = \frac{SD_{\text{residuals}}}{\sqrt{\sum (x_i - \bar{x})^2}}.$$

where $SD_{\text{residuals}}$ is the standard deviation of the residuals, x_i are the individual x values, and $\bar{x}$ is the mean of the x values.

The standard deviation of the residuals is calculated as:

$$SD_{\text{residuals}} = \sqrt{\frac{\sum(y_i - \hat{y}_i)^2}{n-2}},$$

where y_i are the observed y values, $\hat{y}_i$ are the predicted y values from the regression, and n is the number of data points. The number of the degrees of freedom is $n-2$ because we are estimating two parameters (the slope and the intercept) from the data.

Let's compute SE_{slope}, the t-statistic, and the p-value for our example.

```
# calculate the residuals
residuals = y - y_hat
# calculate the sum of squared residuals (SSR)
sum_squared_residuals = np.sum(residuals**2)
# calculate the Residual Standard Error (s_e)
n = len(x)
degrees_of_freedom = n - 2
residual_std_error = np.sqrt(sum_squared_residuals / degrees_of_freedom)
# calculate the sum of squared deviations of x from its mean
x_mean = np.mean(x)
sum_squared_x_deviations = np.sum((x - x_mean)**2)
# put it all together to get SE_slope
# SE_slope = (typical error) / (spread of x)
SE_slope = residual_std_error / np.sqrt(sum_squared_x_deviations)
# verify the result against the value directly from scipy
scipy_slope, scipy_intercept, scipy_r, scipy_p, scipy_se = linregress(x, y)
print(f"manually calculated SE_slope:          {SE_slope:.6f}")
print(f"SE_slope from scipy.stats.linregress:  {scipy_se:.6f}")
```

```
manually calculated SE_slope:          0.057695
SE_slope from scipy.stats.linregress:  0.057695
```

```
t_statistic = (slope-0) / SE_slope
p_value = 2 * (1 - scipy.stats.t.cdf(np.abs(t_statistic), df=degrees_of_freedom))
print(f"manually calculated p-value: {p_value:.6f}")
print(f'scipy p-value:               {scipy_p:.6f}')
```

```
manually calculated p-value: 0.028344
scipy p-value:               0.028344
```

If we choose a significance level $\alpha = 0.05$, the p-value we found indicates that we can reject the null hypothesis and conclude that there is a significant trend between x and y.

If instead of testing if the slope is different from zero, but rather if it is greater than zero (i.e., a one-sided test), we would divide the p-value by 2.

```
p_value = (1 - scipy.stats.t.cdf(np.abs(t_statistic), df=degrees_of_freedom))
scipy_slope, scipy_intercept, scipy_r, scipy_p, scipy_se = linregress(x, y, alternative='greate
print(f"manually calculated p-value: {p_value:.6f}")
print(f'scipy p-value:               {scipy_p:.6f}')
```

```
manually calculated p-value: 0.014172
scipy p-value:               0.014172
```

One last remark. What does the formula for the standard error of the slope mean?

- inside the square root, we have a quantity dependent on y squared divided by a quantity dependent on x squared. Dimensionally this makes sense, because the standard error of the slope should have the same dimension as the slope $\Delta y/\Delta x$.
- the larger the variability of the residuals (i.e., the more scattered the data points are around the regression line), the larger the standard error of the slope, and thus the less precise our estimate of the slope is.
- We can manipulate the formula a little bit to get more intuition:

$$SE_{\text{slope}} = \sqrt{\frac{1}{n-2}}\frac{SD_y}{SD_x}\sqrt{1-r^2},$$

where SD_y and SD_x are the standard deviations of y and x, respectively, and r is the correlation coefficient between x and y. From this formula we can see that:

a) the standard error of the slope decreases with increasing sample size n (more data points lead to a more precise estimate of the slope);
b) imagine all the data points in a rectangular box, and all the possible slopes that can be drawn within that box. If you change the dimensions of the box, you need to account for that, and that is the second term.
c) The last term acounts for the spread of the points about the line.

32.2 Mann-Kendall Trend Test

The method above assumed that the relationship between x and y is linear, and that the residuals are normally distributed. If these assumptions are not met, we can use a non-parametric test like the Mann-Kendall trend test. The intuition behind this test works like a voting system. You go through your data and compare every data point to all points that come after it.

- if a later points is higher, you give a +1 vote
- if a later point is lower, you give a -1 vote
- if they are equal, you give a 0 vote

All these votes are summed up. A large positive sum indicates an increasing trend, a large negative sum indicates a decreasing trend, and a sum close to zero indicates no trend. We can then calculate a test statistic Z based on the sum of votes, and use it to determine the p-value. If the p-value is less than our chosen significance level (e.g., 0.05), we can reject the null hypothesis of no trend. We can use the package `pymannkendall` to perform the Mann-Kendall trend test.

```
from pymannkendall import original_test
mk_result = original_test(y)
print(mk_result)
```

```
Mann_Kendall_Test(trend='increasing', h=True, p=0.04970274086760851, z=1.9625134103851736, Tau=
```

The test concluded that there is an increasing trend, with a p-value of 0.0497.

32.3 Spearman's Rank Correlation

This is another non-parametric test. It assesses how well the relationship between two variables can be described using a monotonic function. It does this by converting the data to ranks and then calculating the Pearson correlation coefficient on the ranks. The Spearman's rank correlation coefficient, denoted by ρ (rho), ranges from -1 to 1, where:

- 1 indicates a perfect positive **monotonic** relationship,
- -1 indicates a perfect negative **monotonic** relationship,
- 0 indicates no monotonic relationship.

This test is robust to outliers and does not assume a linear relationship between the variables.

We can use the `scipy.stats.spearmanr` function to calculate Spearman's rank correlation coefficient and the associated p-value.

```
spearman_corr, spearman_p = scipy.stats.spearmanr(x, y)
print(f"Spearman's correlation: {spearman_corr:.6f}, p-value: {spearman_p:.6f}")
```

```
Spearman's correlation: 0.361958, p-value: 0.049356
```

We found that there is a positive monotonic relationship between x and y, with a p-value of 0.0494, indicating that the relationship is statistically significant at the 0.05 significance level.

32.4 Theil-Sen Estimator

The Theil-Sen estimator is a robust method for estimating the slope of a linear trend. It is particularly useful when the data contains outliers or is not normally distributed. The Theil-Sen estimator calculates the slope as the median of all possible pairwise slopes between data points. We can use the `scipy.stats.theilslopes` function to calculate the Theil-Sen estimator and the associated confidence intervals.

```
from scipy.stats import theilslopes
theil_slope, theil_intercept, theil_lower, theil_upper = theilslopes(y, x, 0.95)
print(f"Theil-Sen slope: {theil_slope:.6f}, intercept: {theil_intercept:.6f}")
```

```
Theil-Sen slope: 0.140364, intercept: 11.836644
```

33 entropy

Let's derive the formula for entropy from first principles.

33.1 image-generating machines

We're given machines that generate images of cats ( ) and dogs ( ). See the following outputs from the machines:

machine 1:
, 5 cats and 1 dog

machine 2:
, 14 cats and 0 dogs

machine 3:
, 4 cats and 6 dogs

machine 4:
, 6 cats and 6 dogs

33.2 surprise

How surprised are we by each output?

For machine 2, we wouldn't be surprised at all if the next image it generates is a cat. It's been generating only cats so far. However, if it generates a dog, that would be quite surprising.

Machine 4 produced equal numbers of cats and dogs. So we would be equally surprised if the next image is a cat or a dog.

Machines 1 and 3 are somewhere in between.

33.3 information

We could say similar things about information. If machine 2 generates another cat, we don't learn anything new, it's the same machine as ever. But if it generates a dog, we learn a lot about this machine's behavior. In contrast, machine 4 generates equal amounts of information whether it produces a cat or a dog, since both have had the same number so far.

To say that we are surprised by an event is the same as saying that we learn some information from it. If an event is not surprising at all, then it doesn't provide us with any new information.

> The sun will rise tomorrow.

This is neither surprising nor informative.

> I have a dragon living in my garage.

This is both surprising and informative.

33.4 quantifying surprise and information

An event that is very likely to happen is not surprising, and doesn't provide us with much information. An event that is unlikely to happen is surprising, and provides us with a lot of information. It seems reasonable to say that the amount of information I we gain from an event x, whose probability is $P(x)$, is inversely proportional to the probability:

$$I(P(x)) = \frac{1}{P(x)} \tag{1}$$

33.5 problems with the inverse formula

There are at least two problems with this inverse formula.

Problem one

An event with zero probability would provide us with infinite information. Maybe that's okay, since I could be infinitely surprised if the sun didn't rise tomorrow. However, a certain event (probability 1) would provide us with only 1 unit of information. That doesn't seem right. A certain event should provide us with no information at all, zero.

Problem two

Let's say that we learn about two completely independent events that happened yesterday, x and y:

- x: My sister flipped a coin and got "heads".
- y: My cousin has won the lottery.

Since the events are independent, the probability that both have happened is the product of their probabilities:

$$P(x \text{ and } y) = P(x)P(y) \tag{2}$$

Now let's calculate the information we gained from learning that both events happened:

$$\begin{aligned} I(P(x \text{ and } y)) &= I(P(x)P(y)) \\ &= \frac{1}{P(x)P(y)} \\ &= \frac{1}{P(x)} \cdot \frac{1}{P(y)} \\ &= I(P(x)) \cdot I(P(y)) \end{aligned} \tag{3}$$

The total information I gained is the product of the information I gained from each event. There's something wrong with that. Assume that my sister flipped a fair coin, so the probability of heads is 1/2. From the inverse formula, this means that the "heads" outcome **doubled** the information I got from yesterday's events. My surprise from learning that my cousin won the lottery was multiplied by 2 by a mere coin toss. That's obviously not ok.

What would make sense is if the total information I gained from both events was the **sum** of the information I gained from

each independent event. Then, if my sister flipped a fair coin, the information I gained from learning that my cousin won the lottery would be increased by a fixed amount, regardless of how surprising the lottery win was.

33.6 a new and better formula

We are looking for a function $I(P(x))$ such that:

- $I(1) = 0$
  A sure event provides no information.
- $I(P(x)P(y)) = I(P(x)) + I(P(y))$
  The information from two independent events is the sum of the information from each event.

We can guess another formula that satisfies these two properties:

$$I(P(x)) = \log\left(\frac{1}{P(x)}\right) \tag{4}$$

- This formula satisfies the first property, since $\log(1) = 0$.
- It also satisfies the second property, since $\log(ab) = \log(a) + \log(b)$.

Claude Shannon, the father of information theory, proposed this formula in his seminal 1948 paper "A Mathematical Theory of Communication". There, he proved (see Appendix 2) that this is the only function that satisfies three properties, related to the two stated above, but more restrictive. See Shannon's three properties, stated in Section 6 of his paper, titled "Choice, Uncertainty, and Entropy".

33.7 entropy

Shannon writes:

> Suppose we have a set of possible events whose probabilities of occurrence are $p_1, p_2, \ldots, p_n$. These probabilities are known but that is all we know concerning which event will occur. Can we find a measure of how much "choice" is involved in the selection of the event or of how uncertain we are of the outcome?

After introducing the three properties that a measure H must satisfy to answer the question above, Shannon presents the formula for the entropy:

> We shall call
>
> $$H = -\sum p_i \log(p_i) \tag{5}$$
>
> the entropy of the set of probabilities $p_1, \ldots, p_n$.

This is the standard formula for entropy, but a better rendition is:

$$H = \sum P_i \log\left(\frac{1}{P_i}\right) \tag{6}$$

(Shannon uses p_i, but I'll go back to using capital P for probabilities.)

This formula is better because it clearly answers the following question:

> What would be the expected amount of information (or surprise) we would gain from a probabilistic event?

We don't know what image our machines will produce next. But we do know the probabilities of each outcome. So we can calculate the **expected** information from the next image:

$$H = \sum_i P(x_i) I(P(x_i)) = \sum_i P(x_i) \log\left(\frac{1}{P(x_i)}\right) \tag{7}$$

The information from each possible outcome x_i is weighted by the probability of that outcome, and we sum over all possible outcomes.

For instance, machine 1 produces dogs with a probability of $1/6$, and cats with a probability of $5/6$. The expected information from the next image is the information from a dog times the probability of getting a dog, plus the information from a cat times the probability of getting a cat. If our machine produced N possible images, we would do the same, summing over all N possible images with their respective probabilities as weights.

33.8 binary machines

In the example of the machines that produce only two outcomes, we have that one probability is P (say, for getting cats) and the other is $Q = 1 - P$. So we can write the entropy as a function of a single probability:

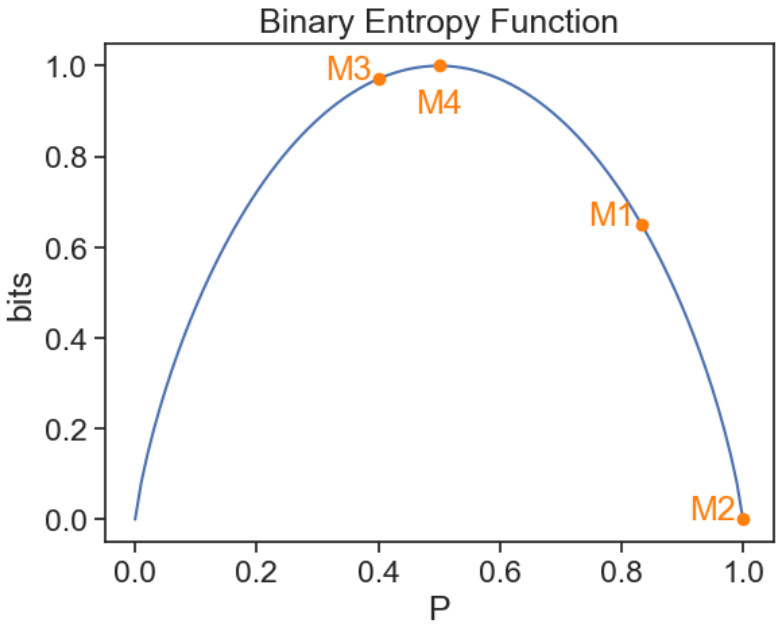
$$H(P) = -P\log(P) - (1-P)\log(1-P)$$

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
import pandas as pd
import matplotlib.gridspec as gridspec
```

```
fig, ax = plt.subplots()
p = np.linspace(0, 1, 100)
eps = 1e-10
def entropy_H(p):
    return -p*np.log2(p+eps)-(1-p)*np.log2(1-p+eps)
ax.plot(p, entropy_H(p), label=r"$  H(p)$")

p_machine1 = 5/6
ax.plot([p_machine1], [entropy_H(p_machine1)], marker='o', ls='none', color='tab:orange')
ax.text(p_machine1-0.05, entropy_H(p_machine1), "M1", ha='center', color='tab:orange')
```

```
p_machine2 = 1.0
ax.plot([p_machine2], [entropy_H(p_machine2)], marker='o', ls='none', color='tab:orange')
ax.text(p_machine2-0.05, entropy_H(p_machine2), "M2", ha='center', color='tab:orange')

p_machine3 = 4/10
ax.plot([p_machine3], [entropy_H(p_machine3)], marker='o', ls='none', color='tab:orange')
ax.text(p_machine3-0.05, entropy_H(p_machine3), "M3", ha='center', color='tab:orange')

p_machine4 = 6/12
ax.plot([p_machine4], [entropy_H(p_machine4)], marker='o', ls='none', color='tab:orange')
ax.text(p_machine4, entropy_H(p_machine4)-0.05, "M4", ha='center', va='top', color='tab:orange

ax.set_xlabel("P")
ax.set_ylabel("bits")
ax.set_title("Binary Entropy Function")
```

```
Text(0.5, 1.0, 'Binary Entropy Function')
```



The expected information is zero for machine 2, which certainly produces only cats ($P = 1$). The expected information is max-

imal for machine 4, which produces cats and dogs with equal probabilities ($P = 1/2$).

In Equations (5) and (6), we didn't explicitly say what the base of the logarithm is. It doesn't matter, since changing the base only changes the units of information. In the example above, we use base 2 because we have a binary choice, and the units for entropy are called bits, suggested by JW Tukey, meaning *binary digits*.

33.9 one last example

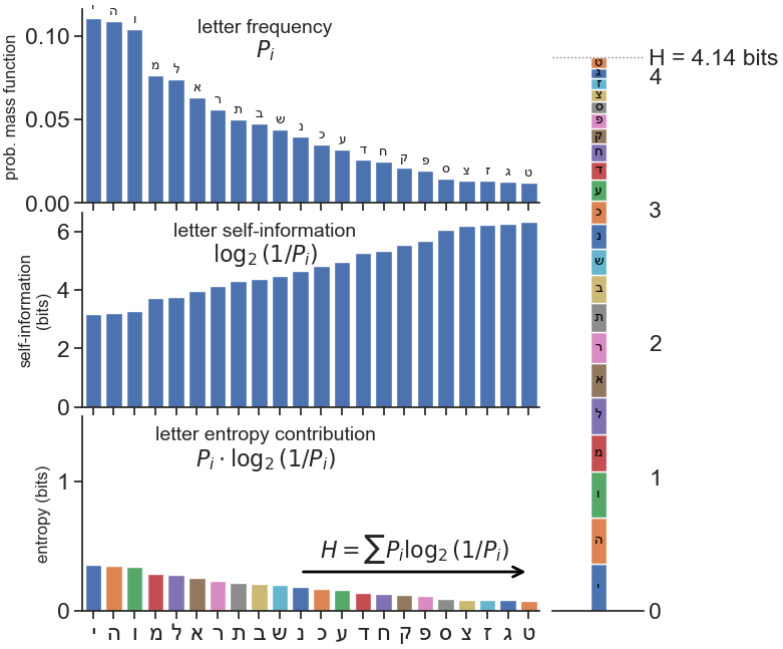
Consider the frequency of the letters in the Hebrew alphabet (there are 22 letters). The most common letter is ," " which appears with a frequency of about 11.06%. The least common letter is ," " which appears with a frequency of about 1.24%. If we opened a book written in Hebrew to a random page, and picked a random letter on that page, what would be the expected information from that letter?

```
df = pd.read_csv("../archive/data/letter_frequency_hebrew.csv", sep="\t")

fig= plt.figure(1, figsize=(8, 8))

gs = gridspec.GridSpec(3, 2, width_ratios=[1,0.2], height_ratios=[1,1,1])
gs.update(left=0.16, right=0.86,top=0.88, bottom=0.13, hspace=0.05, wspace=0.05)

ax0 = plt.subplot(gs[0, 0])
ax1 = plt.subplot(gs[1, 0], sharex=ax0)
ax2 = plt.subplot(gs[2, 0], sharex=ax0)
ax3 = plt.subplot(gs[:, 1])

bar_width = 0.8

ax0.bar(df['Letter'], df['Frequency(%)']/100, width=bar_width)
for i, letter in enumerate(df['Letter']):
    ax0.text(i, df['Frequency(%)'][i]/100 + 0.002, letter, ha='center', va='bottom', fontsize=
ax0.set_xticklabels(df['Letter'], rotation=0);
ax0.set_ylabel('prob. mass function', fontsize=12)
ax0.set(xlim=(-0.5, len(df['Letter'])-0.5))
```

```
ax0.text(0.4, 0.95, 'letter frequency', transform=ax0.transAxes, ha='center', va='top', fontsi
ax0.text(0.4, 0.85, r'$P_i$', transform=ax0.transAxes, ha='center', va='top', fontsize=16)

ax1.bar(df['Letter'], np.log2(1/(df['Frequency(%)']/100)), width=bar_width)
ax1.set_ylabel('self-information\n(bits)', fontsize=12)
ax1.text(0.4, 0.95, 'letter self-information', transform=ax1.transAxes, ha='center', va='top',
ax1.text(0.4, 0.85, r'$\log_2(1/P_i)$', transform=ax1.transAxes, ha='center', va='top', fontsi

Hi = (df['Frequency(%)']/100) * np.log2(1/(df['Frequency(%)']/100))
colors = sns.color_palette("deep", n_colors=len(df))
for i, (letter, h) in enumerate(zip(df['Letter'], Hi)):
    ax2.bar(letter, h, color=colors[i % len(colors)], width=bar_width)
ax2.set_ylabel('entropy (bits)', fontsize=12)
ax2.set(yticks=[0,1,2])
ax2.set_ylim(0,4.5/3)
ax2.text(0.4, 0.95, 'letter entropy contribution', transform=ax2.transAxes, ha='center', va='t
ax2.text(0.4, 0.85, r'$P_i\cdot\log_2(1/P_i)$', transform=ax2.transAxes, ha='center', va='top'

# Annotate arrow from x=10 to x=20 on ax2
ax2.annotate(
    '',
    xy=(len(df['Letter'])-1, 0.3),
    xytext=(10, 0.3),
    arrowprops=dict(arrowstyle='->', lw=2, color='black')
)
ax2.text(15.5, 0.35, r'$H=\sum P_i \log_2(1/P_i)$', ha='center', va='bottom', fontsize=16)
for spine in ['top', 'right']:
    ax0.spines[spine].set_visible(False)
    ax1.spines[spine].set_visible(False)
    ax2.spines[spine].set_visible(False)

# Plot stacked bars of the entropy contributions in ax3
bottom = 0
for i, H in enumerate(Hi):
    p = ax3.bar(0, H, width=bar_width, label=df['Letter'][i], bottom=bottom)
    ax3.text(0, bottom + H/2, f"{df['Letter'][i]}", ha='center', va='center', color='black', fo
    bottom += H
ax3.set_xlim(-len(df['Letter'])/2*0.2, len(df['Letter'])/2*0.2)
ax3.set_ylim(0,4.5)
for spine in ['left', 'top', 'right']:
```

```
    ax3.spines[spine].set_visible(False)
ax3.yaxis.tick_right()

total_H = np.sum(Hi)
ax3.axhline(total_H, color='gray', linestyle=':', linewidth=1)
ax3.set(
    xticks=[],
    yticks=[0,1,2,3,4,total_H],
    yticklabels=[0,1,2,3,4, f'H = {total_H:.2f} bits']
)
ax3.tick_params(axis='y', which='major', length=0)

fig.tight_layout();
```

```
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_99634/300921586.py:19: UserWarning:
  ax0.set_xticklabels(df['Letter'], rotation=0);
/var/folders/cn/m58l7p_j6j10_4c43j4pd8gw0000gq/T/ipykernel_99634/300921586.py:74: UserWarning:
  fig.tight_layout();
```



The expected information would be 4.14 bits. Because Hebrew has structure (like any other language), this is less than the

maximum possible information. If all 22 letters were equally likely, the expected information would be:

$$\begin{aligned} H_{\max} &= \sum_{i=1}^{22} \frac{1}{22} \log_2 \left(\frac{1}{\frac{1}{22}} \right) \\ &= \log_2(22) \\ &\approx 4.46 \text{ bits} \end{aligned}$$

From this exercise, we learned that in the case that all outcomes are equally likely, the entropy is simply the logarithm of the number of possible outcomes. This is a useful fact to remember.

34 cross-entropy and KL divergence

Assume I live in city A, where it rains 50% of the days. A friend of mine lives in city B, where it rains 10% of the days.

34.1 wrong model

What happens when my friend visits me in city A and, not knowing any better, assumes that it rains 10% of the days?

We have two probability distributions, the true weather distribution P, and the assumed weather distribution Q:

- **True distribution:** P(rain) = 0.5, P(no rain) = 0.5
- **Assumed distribution:** Q(rain) = 0.1, Q(no rain) = 0.9

What will be the expected surprise of my friend, when he visits me in city A? Now that we have discussed surprise (information) and entropy, we can calculate the following quantity, called *cross-entropy*:

$$H(P, Q) = -\sum_x P(x) \log Q(x)$$

My friend will evaluate his surprise using the mental model that he has, i.e., the assumed distribution Q. For example, because he comes from a dry city, every time it rains he is surprised a lot more than when it does not rain.

However, since my friend is visiting me in city A, he will actually experience the weather according to the true distribution P. He will not weigh the big surprise of rain with the probability of

rain in city B (10%), but with the probability of rain in city A (50%).

This reasoning explains the asymetry of the cross-entropy: the first argument is the true distribution, which determines how often each event happens, while the second argument is the assumed distribution, which determines how surprised my friend will be when each event happens.

Let's compute my friend's expected surprise when he visits me in city A:

$$\begin{aligned} H(P,Q) &= -\sum_x P(x)\log Q(x) \\ &= -(P(\text{rain})\log Q(\text{rain}) + P(\text{no rain})\log Q(\text{no rain})) \\ &= -(0.5\log 0.1 + 0.5\log 0.9) \\ &= -(0.5\cdot -1 + 0.5\cdot -0.045757) \\ &= 1.74 \text{ bits}, \end{aligned}$$

where we used the base-2 logarithm, so the result is in bits.

To see the asymetry of the cross-entropy, let's compute my expected surprise when I visit my friend in city B:

$$\begin{aligned} H(Q,P) &= -\sum_x Q(x)\log P(x) \\ &= -(Q(\text{rain})\log P(\text{rain}) + Q(\text{no rain})\log P(\text{no rain})) \\ &= -(0.1\log 0.5 + 0.9\log 0.5) \\ &= -(0.1\cdot -1 + 0.9\cdot -1) \\ &= 1 \text{ bits}, \end{aligned}$$

My friend's expected surprise will be higher when he visits me in city A (1.74 bits) than my expected surprise when I visit him in city B (1 bit).

34.2 Kullback-Leibler divergence

Not all of my friend's surprise is due to the fact that he has an inaccurate mental model of the weather. Some of his surprise is simply due to the inherent randomness of the weather. This would be the same surprise that I myself experience when I live in city A, and I have the correct mental model of the weather.

It would make sense to separate the surprise that is due to the inherent randomness of the weather from the surprise that is due to my friend's wrong mental model. We can do this by subtracting the entropy of the true distribution P from the cross-entropy $H(P, Q)$:

$$D_{KL}(P\|Q) = H(P, Q) - H(P)$$

This quantity is called the Kullback-Leibler divergence, and it measures the amount of surprise that is only due to my friend's wrong mental model.

Let's use the properties of logarithms to rewrite the KL divergence in a more convenient form:

$$\begin{aligned} D_{KL}(P\|Q) &= H(P, Q) - H(P) \\ &= -\sum_x P(x) \log Q(x) + \sum_x P(x) \log P(x) \\ &= \sum_x P(x)(\log P(x) - \log Q(x)) \\ &= \sum_x P(x) \log \frac{P(x)}{Q(x)}. \end{aligned}$$

This is the most common form of the KL divergence.

When our model is perfect, i.e., when $P = Q$, the KL divergence is zero ($\log(P/P) = 0$), because there is no extra surprise due to a wrong mental model. When our model is not perfect, the KL divergence is always positive, because we are always more surprised when our mental model is wrong.

34.3 model training and objective functions

We might want to train a model that classifies photos. We have a dataset of photos, and for each photo we know the correct label (cat, dog, elephant, etc.). The goal of the model is to predict the correct label for each photo.

At every step of the training process, we need to evaluate how well the model is doing. The true data distribution P is given by the labels in the training dataset, while the model's predicted distribution Q is given by the model's output. Ideally, our model's predicted distribution Q should be as close as possible to the true data distribution P. That's sounds like a job for the KL divergence!

We will adjust the model's parameters to minimize the KL divergence between the true data distribution P and the model's predicted distribution Q. In practice, we will minimize the cross-entropy $H(P, Q)$ instead of the KL divergence $D_{KL}(P\|Q)$, because the entropy $H(P)$ does not depend on the model's parameters, and therefore does not affect the optimization process. Think about it: no matter what the model's parameters are, the entropy of the true data distribution P will always be the same. So minimizing the KL divergence is equivalent to minimizing the cross-entropy. We don't care if they differ by a constant.