Statistics and Machine Learning

Yair Mau

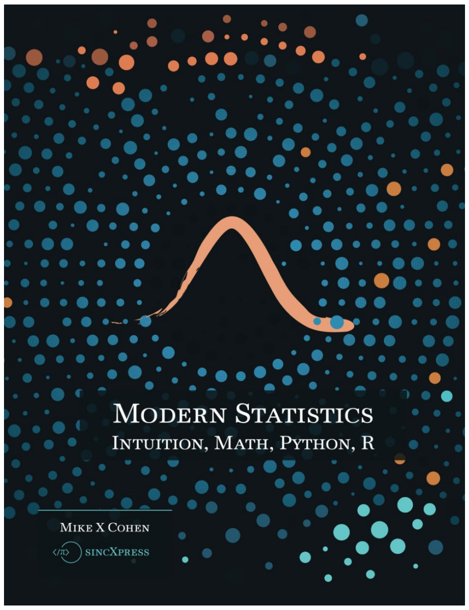
Table of contents

home

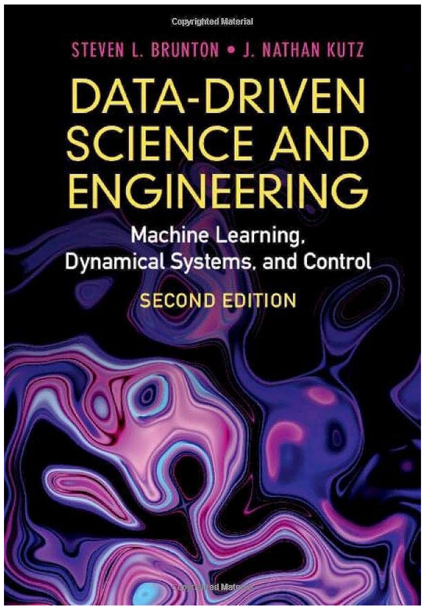
I'm teaching myself statistics and machine learning, and the best way to truly understand is to use the new tools I've acquired. This is what this website is for. It is mainly a reference guide for my future self.

books

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



Modern Statistics: Intuition, Math, Python, R
by Mike X Cohen
Github


Copyrighted Material
STEVEN L. BRUNTON • J. NATHAN KUTZ
DATA-DRIVEN SCIENCE AND ENGINEERING
Machine Learning, Dynamical Systems, and Control
SECOND EDITION

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.

Part I

data

1 height data

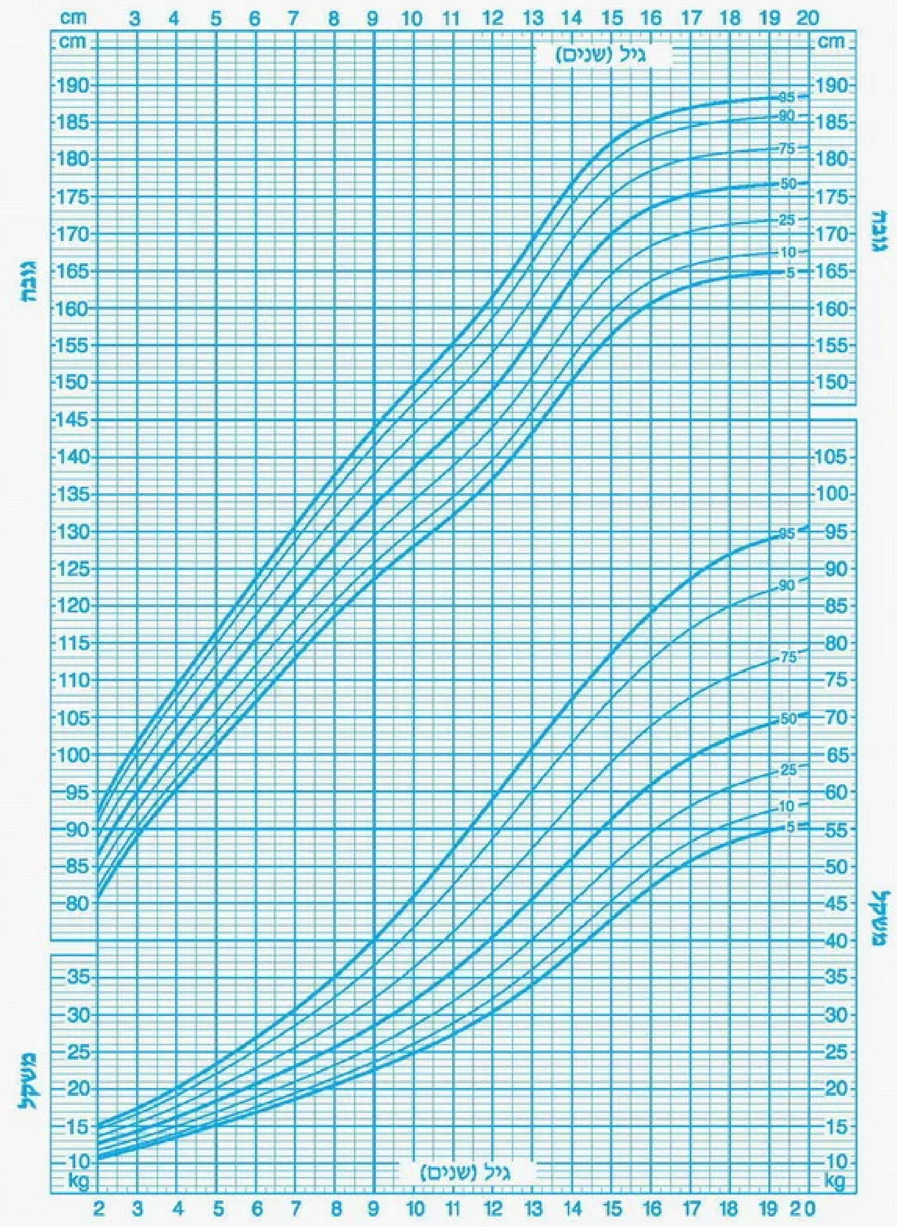
I found growth curves for girls and boys in Israel:

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

For example, see this:

בנים 20-2 שנים - עקומות גובה לפי גיל/ משקל לפי גיל

בנים



SOURCE: Developed by the National Center for Health Statistics in collaboration with the National Center for Chronic Disease Preventionand Health Promotion (2000)

מדינת ישראל - משרד הבריאות

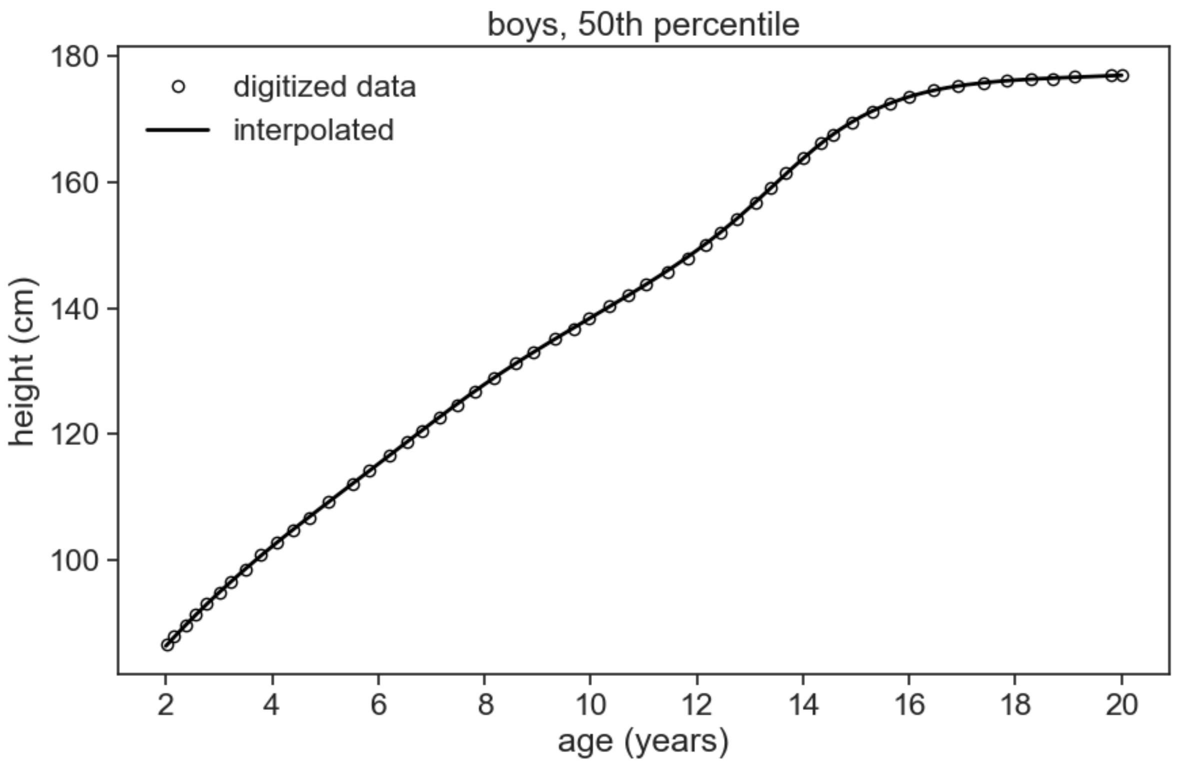
I used the great online resource Web Plot Digitizer v4 to extract the data from the images files. I captured all the growth curves as best as I could. The first step now is to get interpolated versions of the digitized data. For instance, see below the 50th percentile for boys:

```python
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
```

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

```python
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)
```

```python
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.

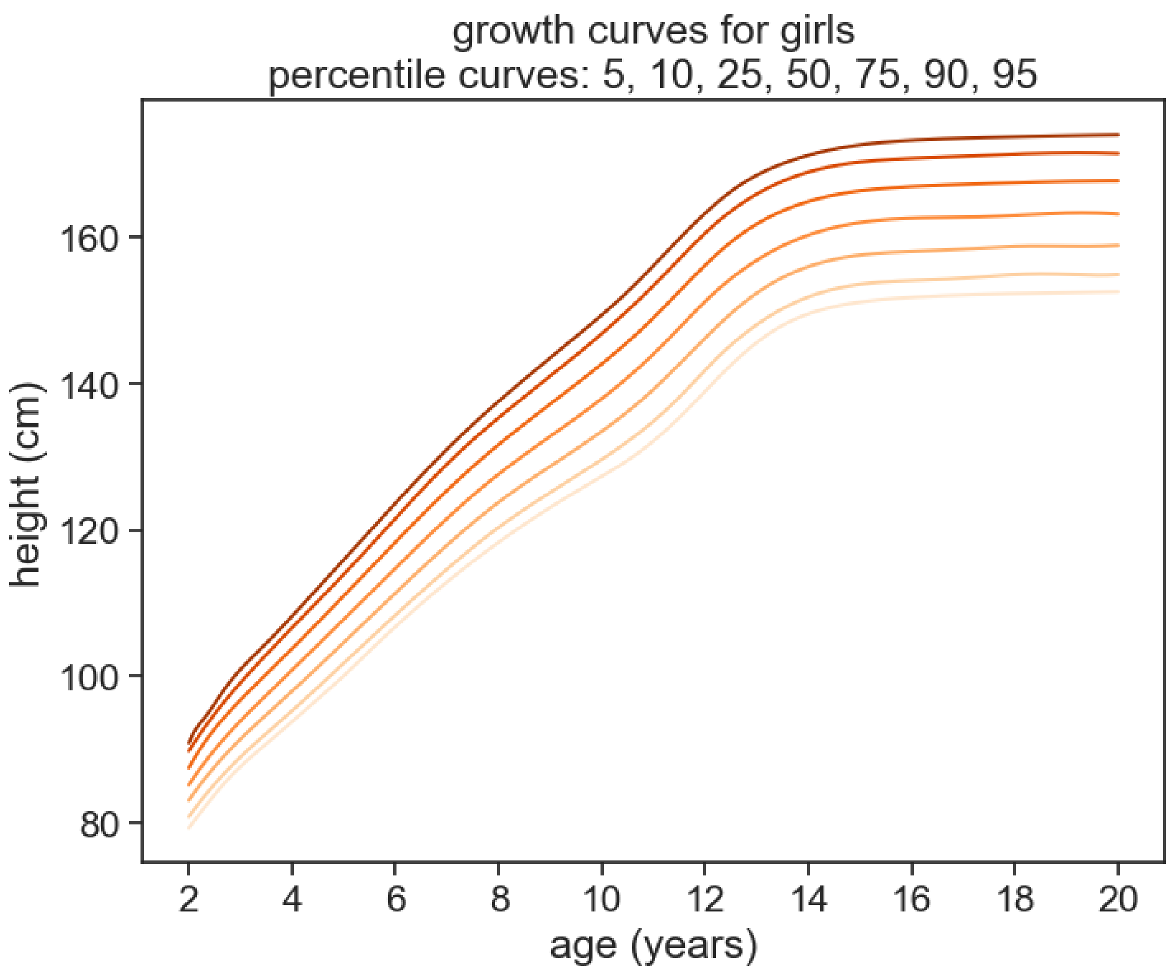
```python
col_names = ['p05', 'p10', 'p25', 'p50', 'p75', 'p90', 'p95']
file_names_boys = ['boys-p05.csv', 'boys-p10.csv', 'boys-p25.csv', 'boys-p50.csv',
                   'boys-p75.csv', 'boys-p90.csv', 'boys-p95.csv',]
file_names_girls = ['girls-p05.csv', 'girls-p10.csv', 'girls-p25.csv', 'girls-p50.csv',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

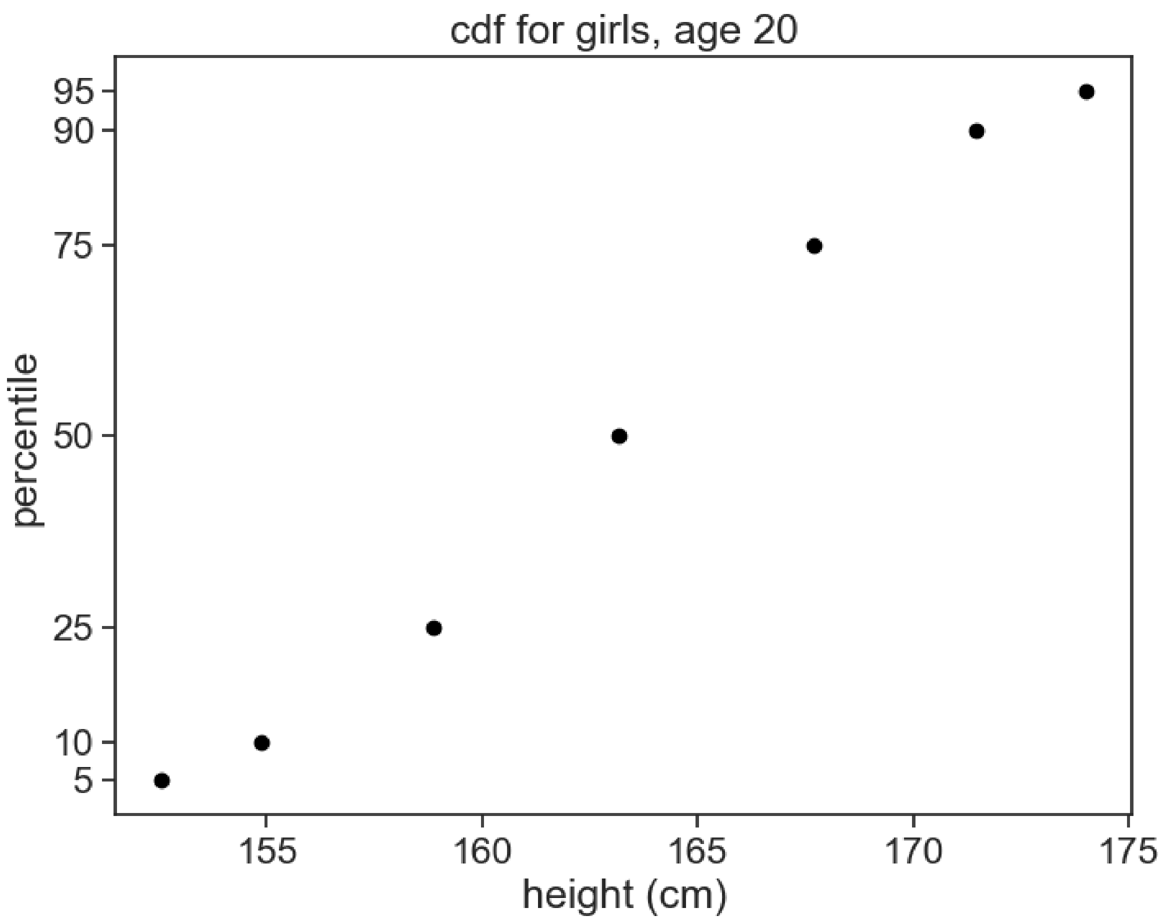
	p05	p10	p25	p50	p75	p90	p95
age							
2.0	79.269087	80.794167	83.049251	85.155597	87.475854	89.779822	90.882059
2.1	80.202106	81.772053	84.052858	86.207778	88.713405	90.883740	92.409913
2.2	81.130687	82.706754	85.011591	87.211543	89.856186	91.940642	93.416959
2.3	82.048325	83.601023	85.928399	88.170313	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

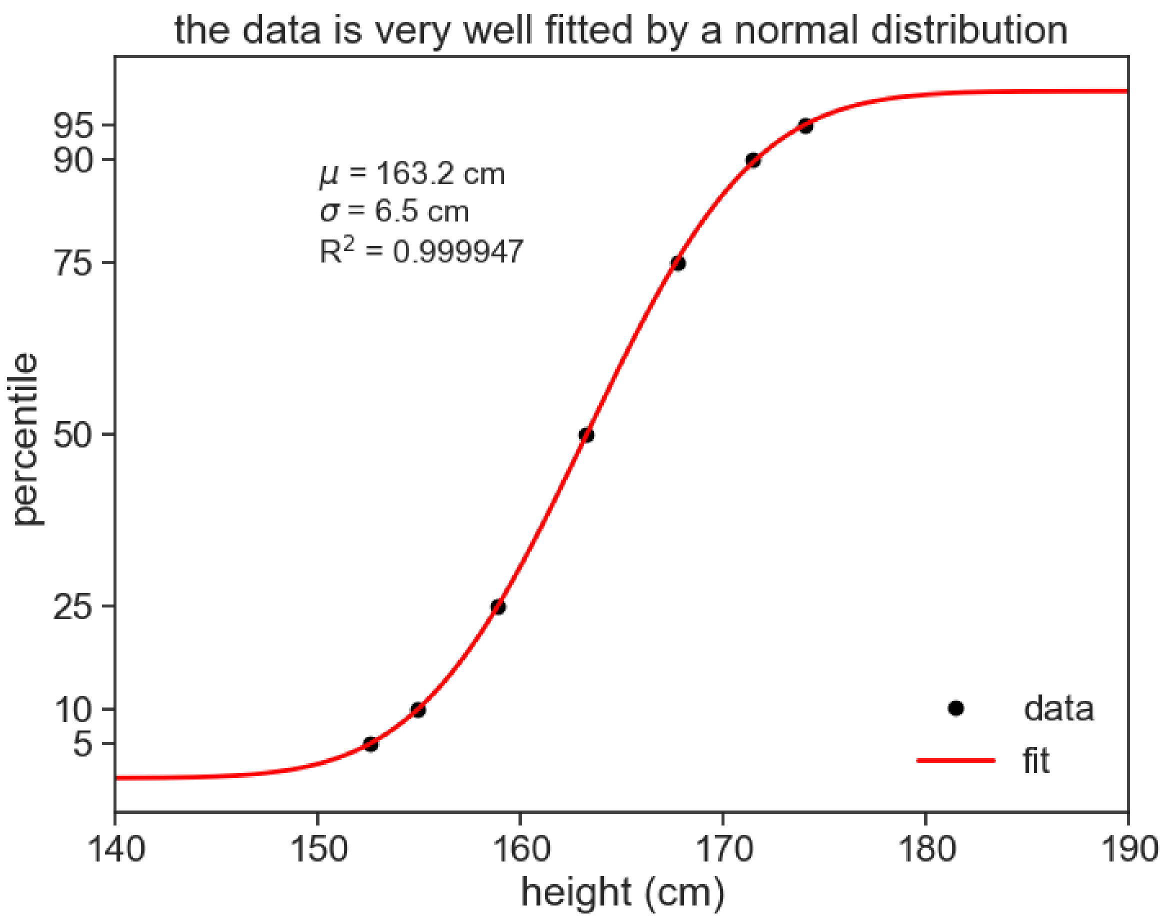
$$\Phi(x) = \frac{1}{2}\left(1 + \mathrm{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right)$$

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)
```

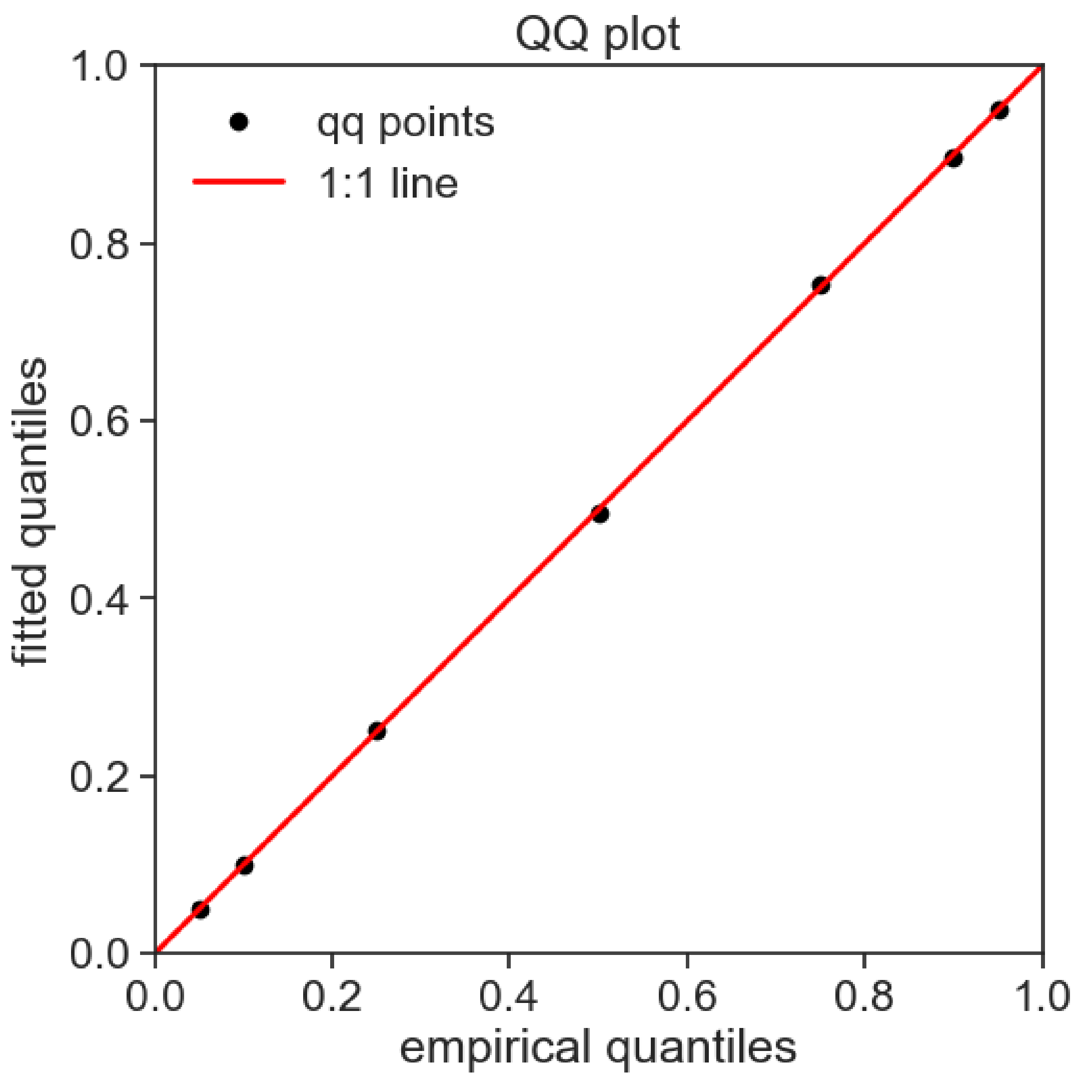
```
data = df_girls.loc[20.0]
params, _ = curve_fit(erf_model, data, percentile_list, p0=p0,
                        bounds=([100, 3],   # lower bounds for mu and sigma
                                [200, 10])  # upper bounds for mu and sigma
                        )
# store the parameters in the 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:.6
        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
```

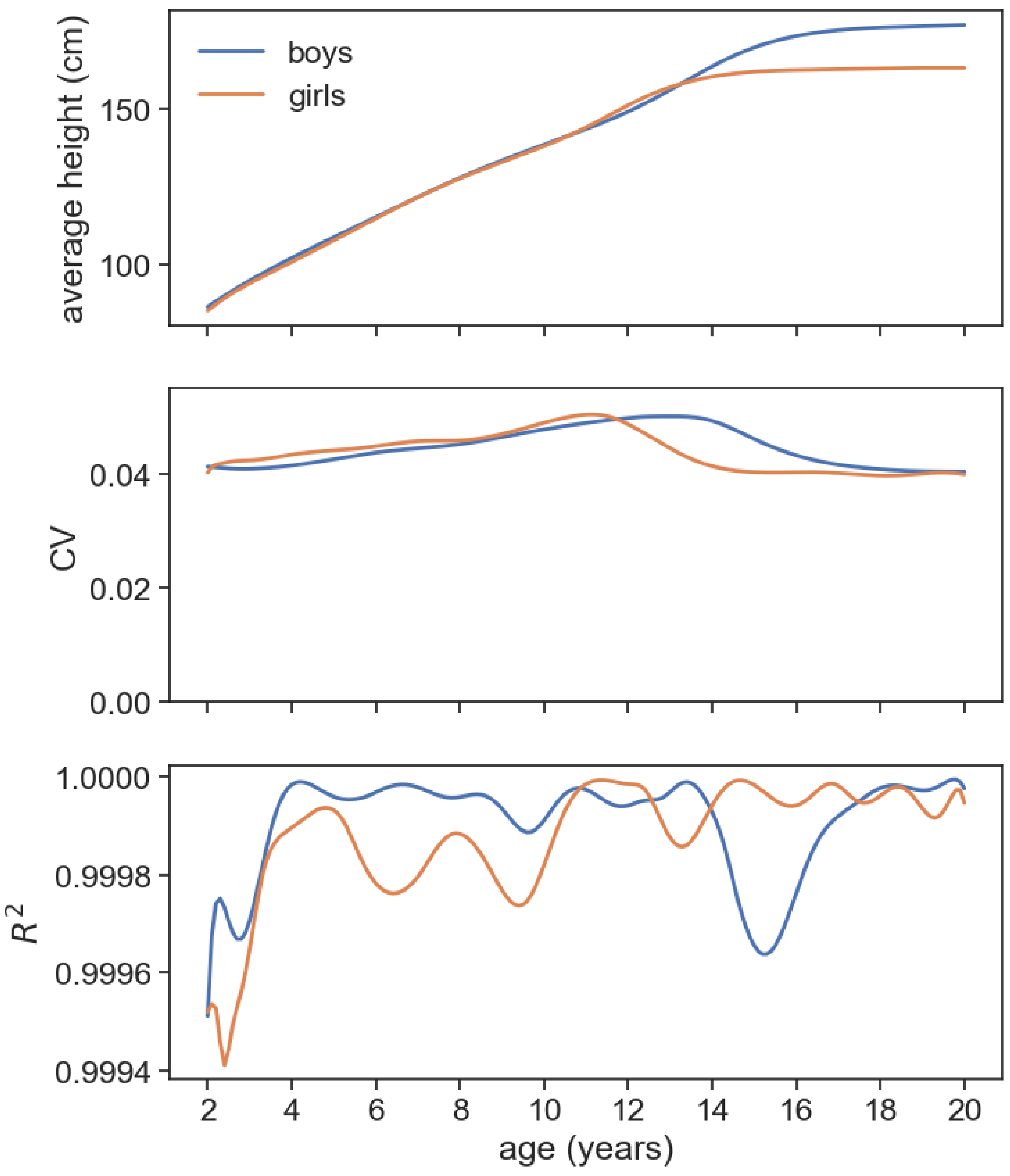
```
p0 = [80, 3]
# loop over ages in the index, calculate mu and sigma
for i in df_boys.index:
    # fit the model to the data
    data = df_boys.loc[i]
    params, _ = curve_fit(erf_model, data, percentile_list, p0=p0,
                          bounds=([70, 2],   # lower bounds for mu and sigma
                                  [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-
70.1	0.000007	3.369929e-06	1.401926e-06	5.423176e-07	2.172465e-07	9.118694e-08	3.914667e-
70.2	0.000008	3.830459e-06	1.597215e-06	6.199308e-07	2.490751e-07	1.048086e-07	4.509972e-
70.3	0.000009	4.350475e-06	1.818328e-06	7.081296e-07	2.853621e-07	1.203810e-07	5.192270e-
70.4	0.000010	4.937172e-06	2.068480e-06	8.082806e-07	3.267014e-07	1.381707e-07	5.973725e-
...	...	...	...	...	...	...	...
219.5	0.000000	5.214425e-307	1.377605e-289	3.568527e-277	6.457994e-266	2.232144e-255	6.340272e-
219.6	0.000000	1.813597e-307	5.050074e-290	1.356408e-277	2.537010e-266	9.046507e-256	2.642444e-
219.7	0.000000	6.302763e-308	1.849870e-290	5.151948e-278	9.959447e-267	3.663840e-256	1.100546e-
219.8	0.000000	2.188653e-308	6.771033e-291	1.955386e-278	3.906942e-267	1.482823e-256	4.580523e-
219.9	0.000000	7.594139e-309	2.476504e-291	7.416066e-279	1.531537e-267	5.997065e-257	1.905138e-

```
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
```

```python
# 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
```

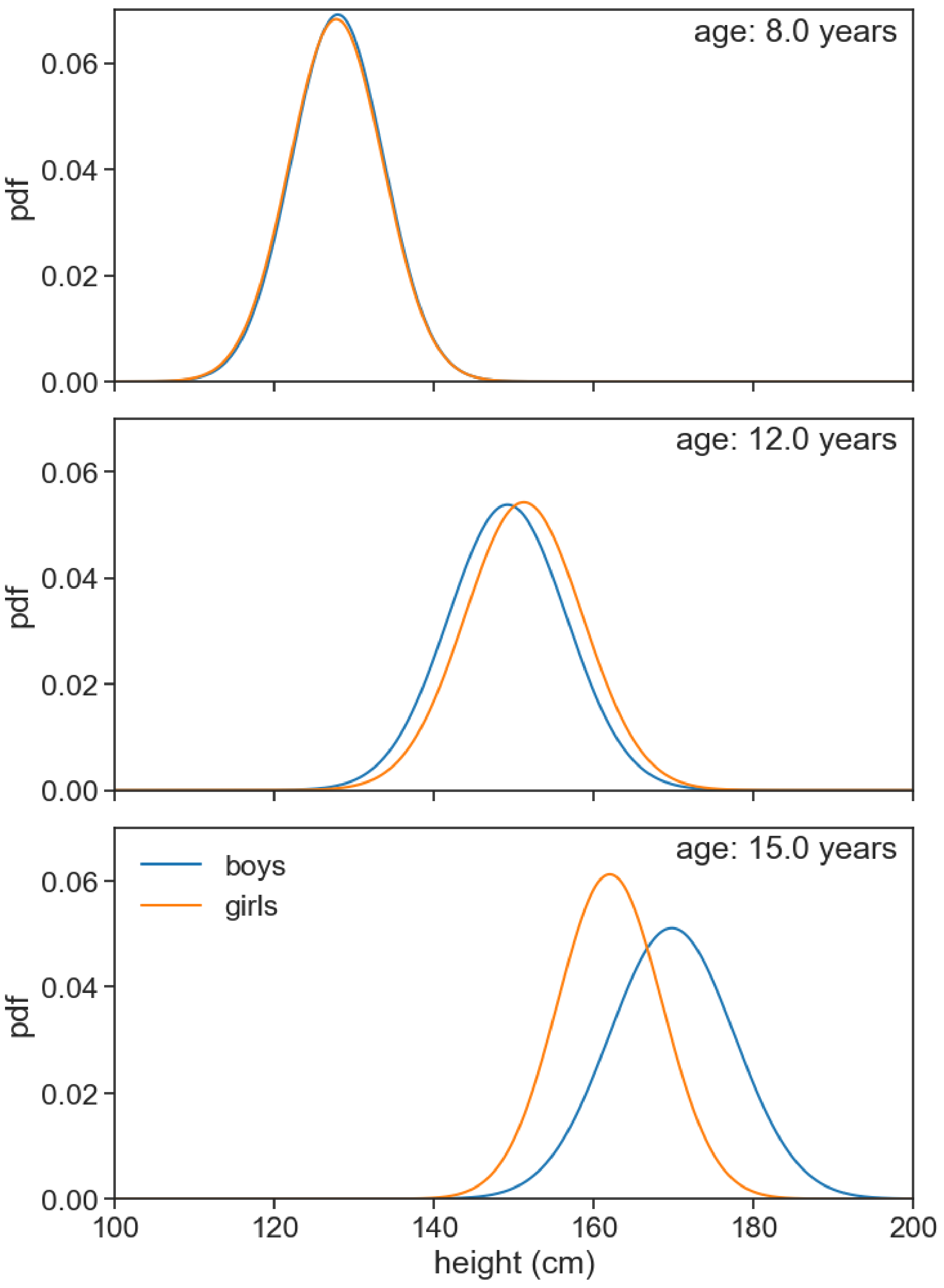
```
Unable to display output for mime type(s): text/html
```

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.lo
    pdf_girls = norm.pdf(height_list, loc=df_stats_girls.loc[age]['mu'], scale=df_stats_girls
    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=
    ax[i].set(ylabel='pdf',
              ylim=(0, 0.07),
             )
ax[2].legend(frameon=False)
ax[2].set(xlabel='height (cm)',
          xlim=(100, 200),);
```


age: 8.0 years
age: 12.0 years
age: 15.0 years
boys
girls
pdf
0.00
0.02
0.04
0.06
100
120
140
160
180
200
height (cm)

- 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.

Part II

hypothesis testing

2 one-sample t-test

2.1 Question

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

2.2 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

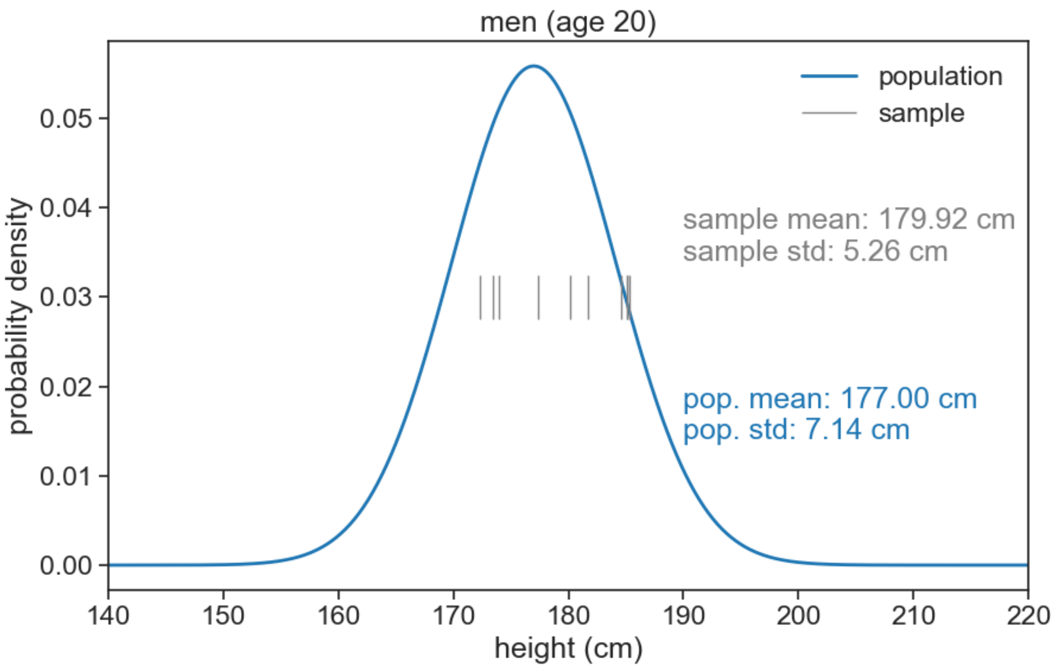
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_1samp, t
%matplotlib widget
```

```python
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.

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

```python
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

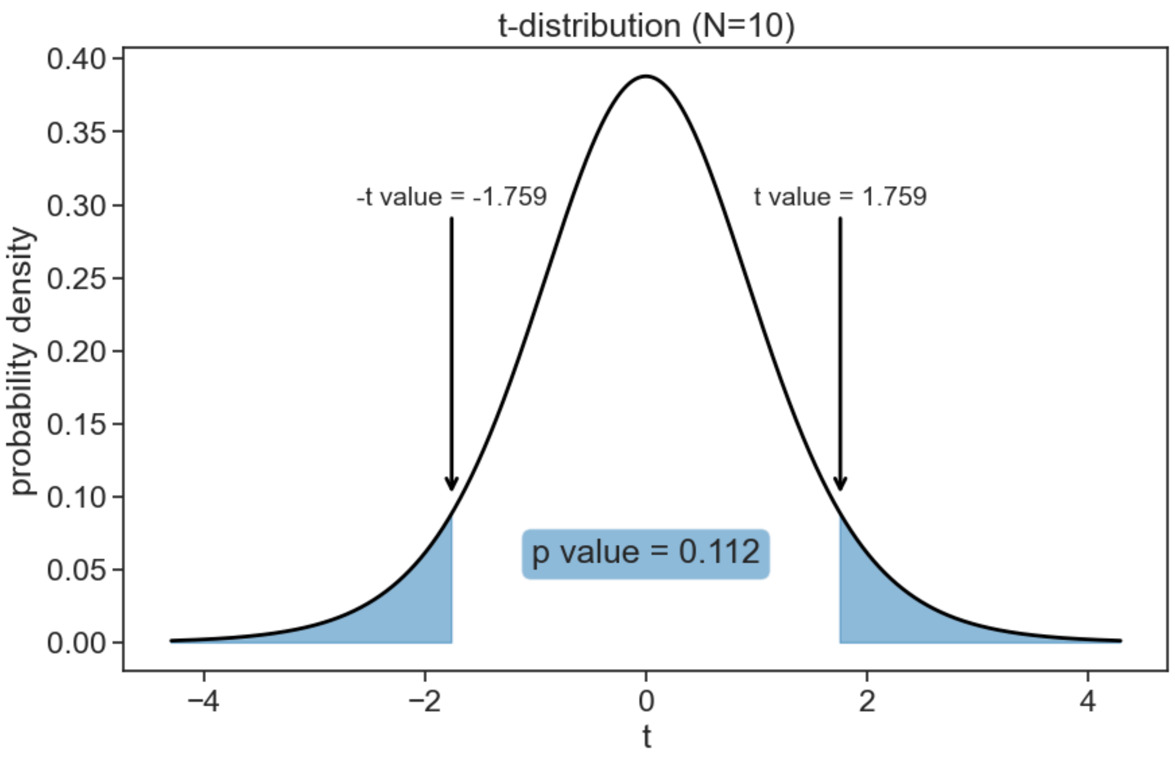
Let's try the formula above and compare it with scipy's ttest_1samp function.

```python
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)",
        );
```



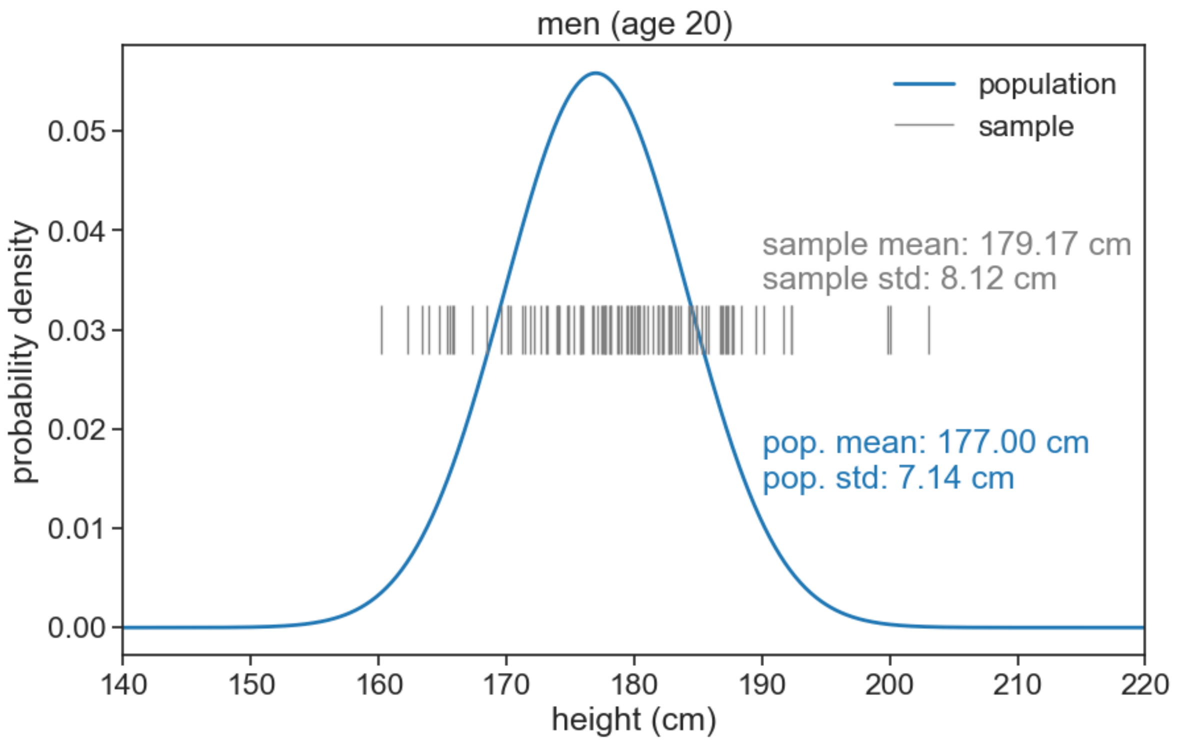
The p value is the fraction of the t distribution that is more extreme than the observed t value. If the p value is less than the significance level, we reject the null hypothesis. In this case, the p value is larger than the significance level, so we fail to reject the null hypothesis. This means that 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.

2.3 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)
```

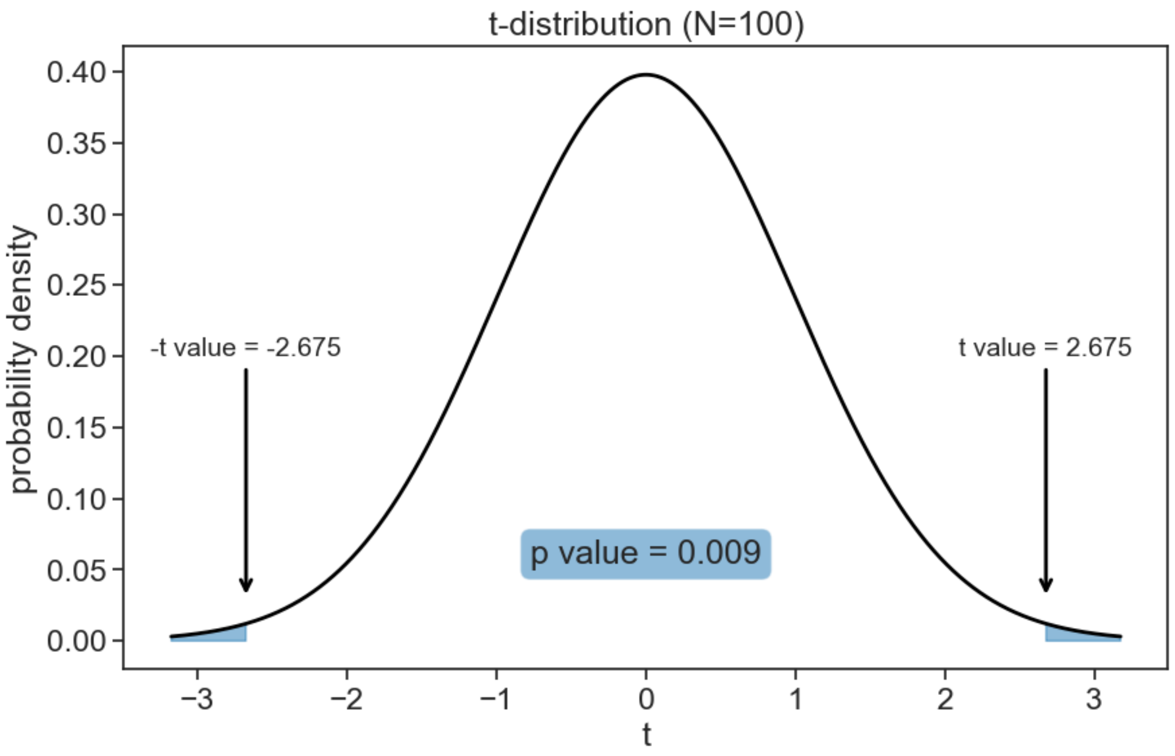
```python
height_list = np.arange(140, 220, 0.1)
pdf_boys = norm.pdf(height_list, loc=mu_boys, scale=sigma_boys)

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)",
       );
```



2.4 Question 2

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

2.5 Hypotheses

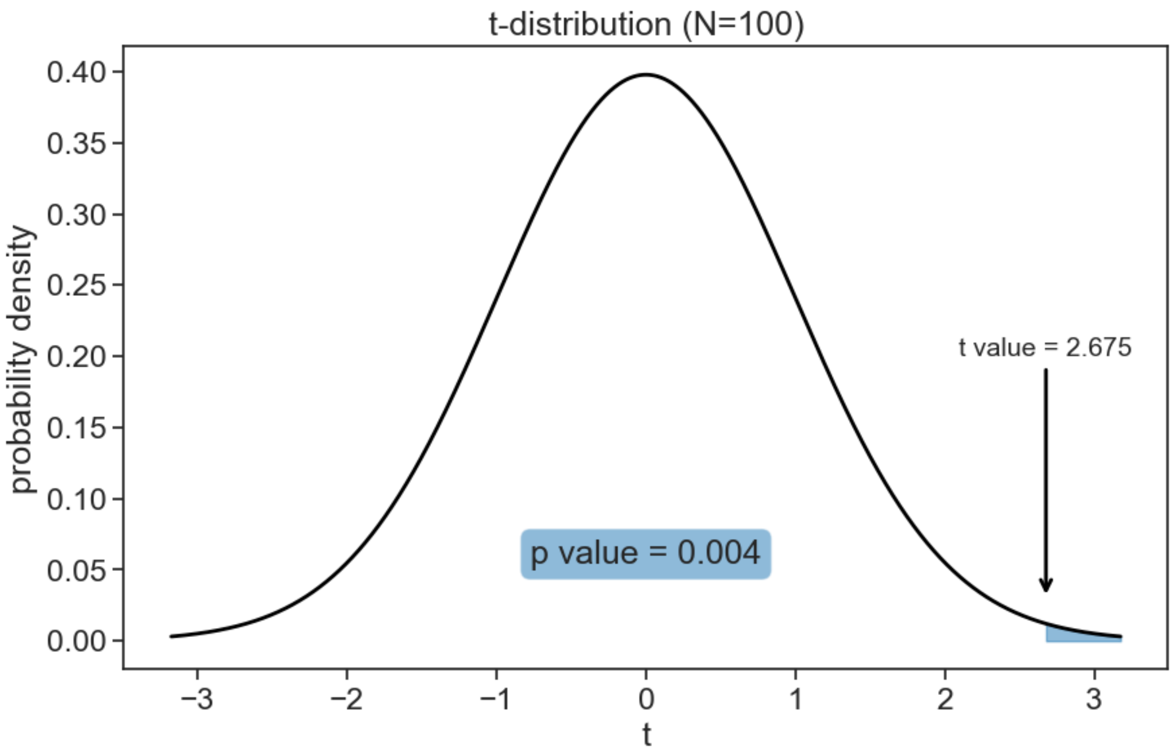
- Null hypothesis: The sample mean is equal to the population mean.
- Alternative hypothesis: The sample mean is higher the population mean.
- Significance level: 0.05

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.

```python
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
```

```python
# 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.

3 independent samples t-test

3.1 Question

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

3.2 Hypotheses

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

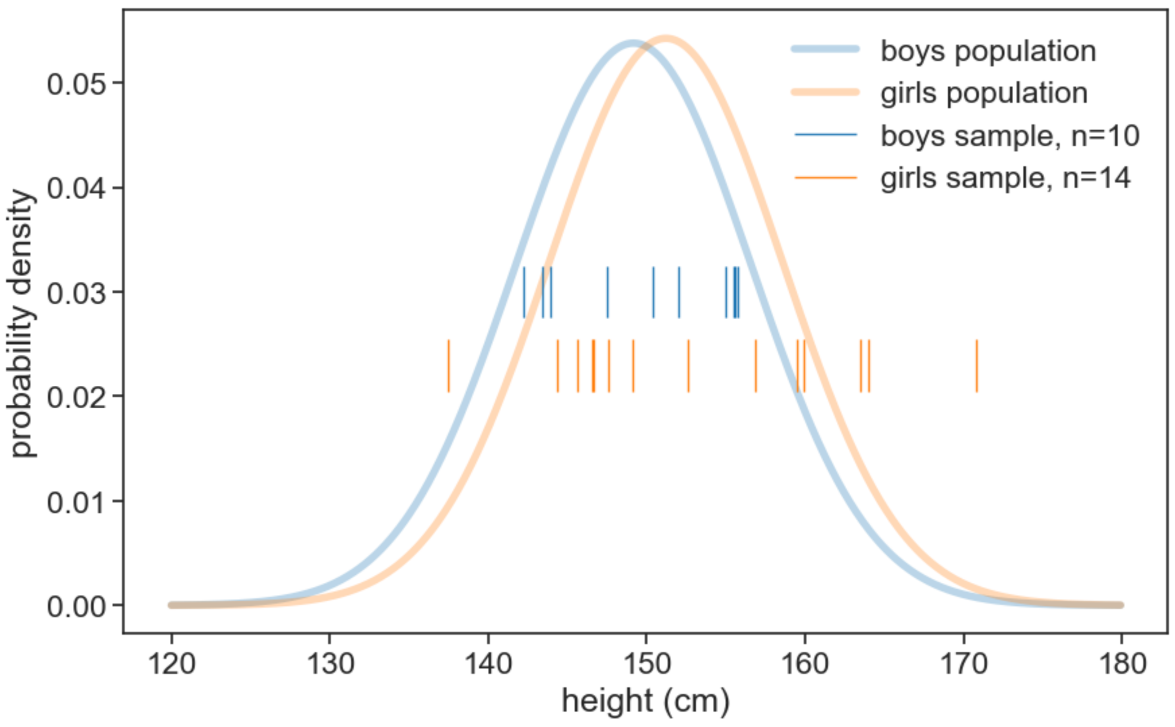
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
# %matplotlib widget
```

```python
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.

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\Theta} \tag{3.1}$$

$$\Theta = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \tag{3.2}$$

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.p
```

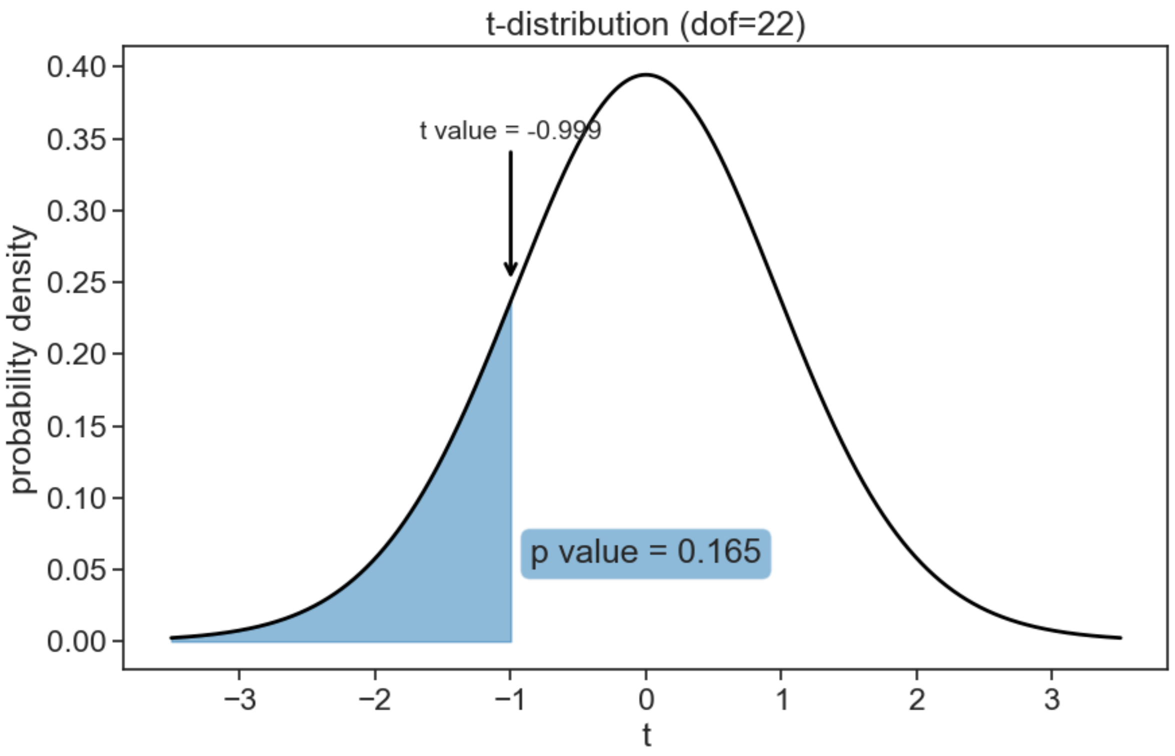
```
t-statistic: -0.999, p-value: 0.164
t-statistic (scipy): -0.999, p-value (scipy): 0.165
```

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.

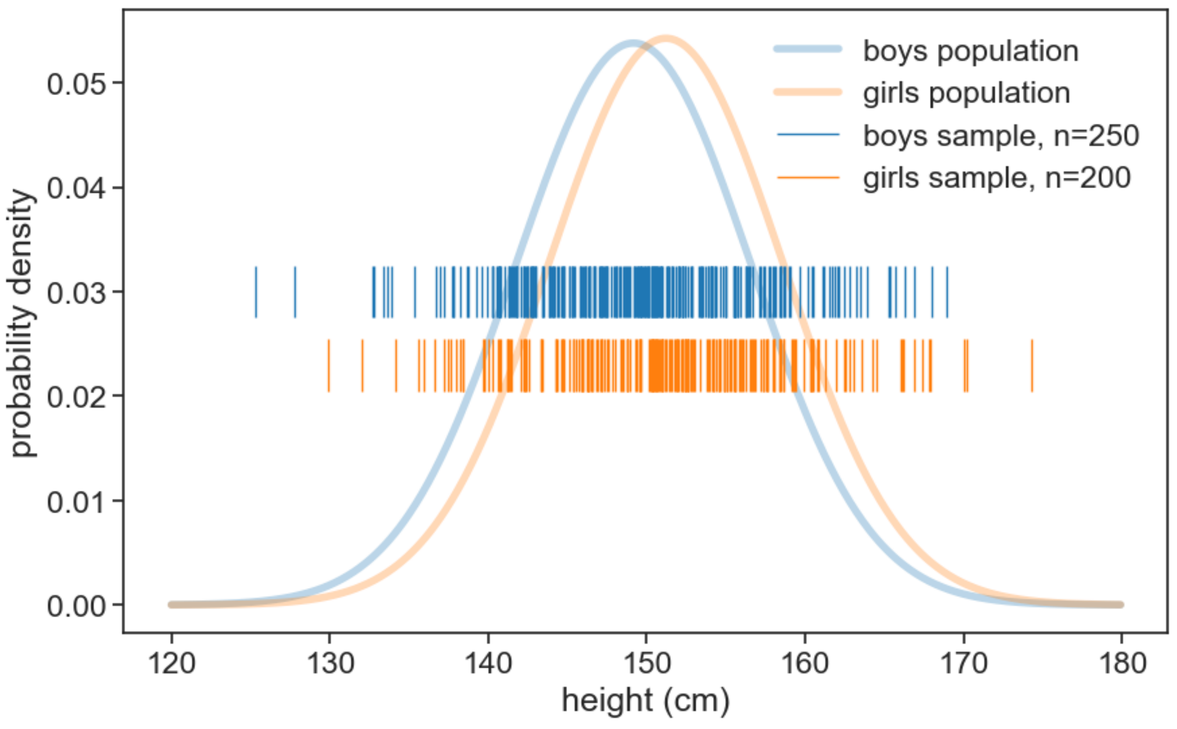
3.3 increasing sample size

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 (boy
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.
```

```
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

4 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
```

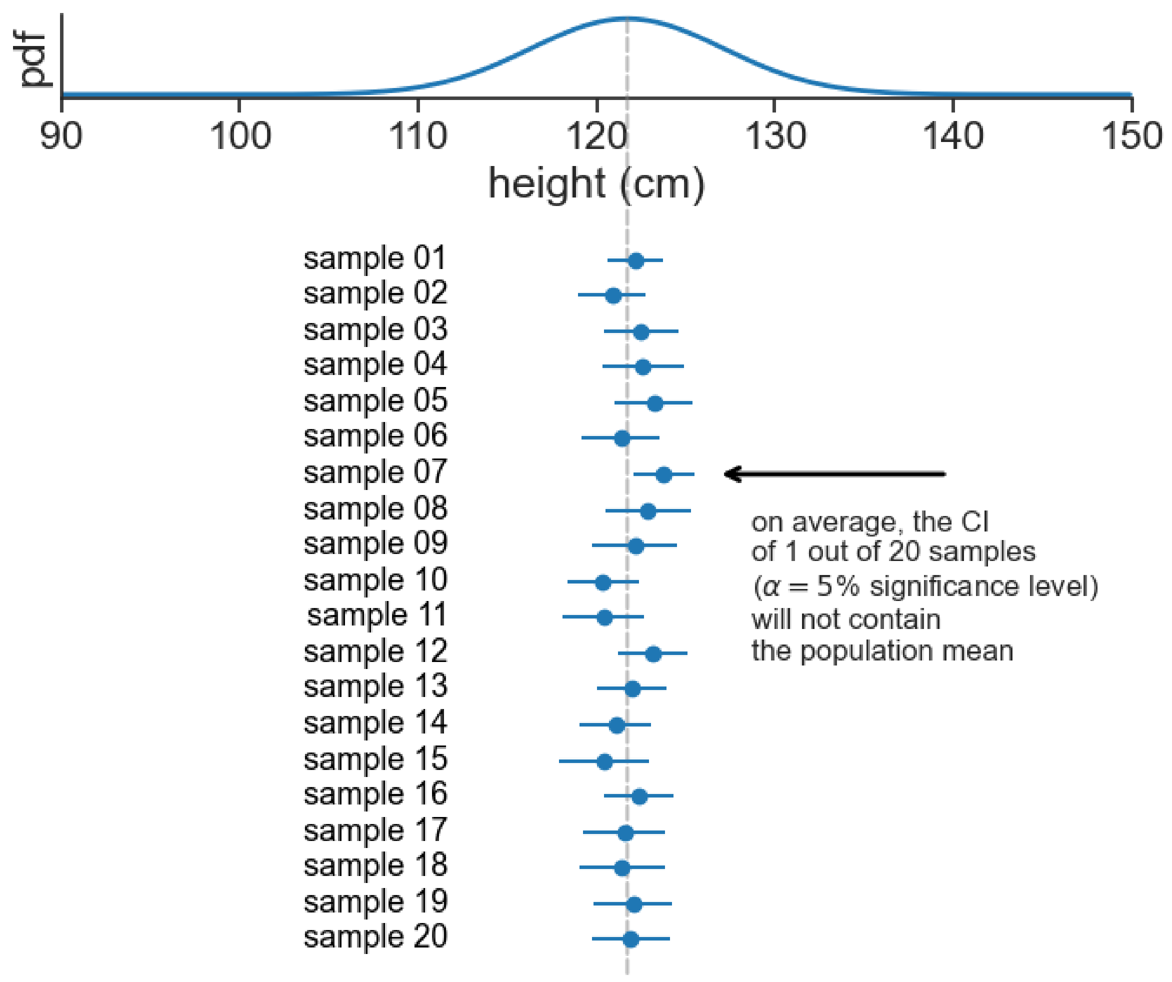
```
df_boys = pd.read_csv('../archive/data/height/boys_height_stats.csv', index_col=0)
age = 7.0
mu_boys = df_boys.loc[age, 'mu']
sigma_boys = df_boys.loc[age, 'sigma']
```

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", coo
                      axesA=ax0, axesB=ax1, color="gray", linestyle='--', linewidth=1.5, alpl
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
```

```python
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

5 analytical confidence interval

```python
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).

5.1 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:

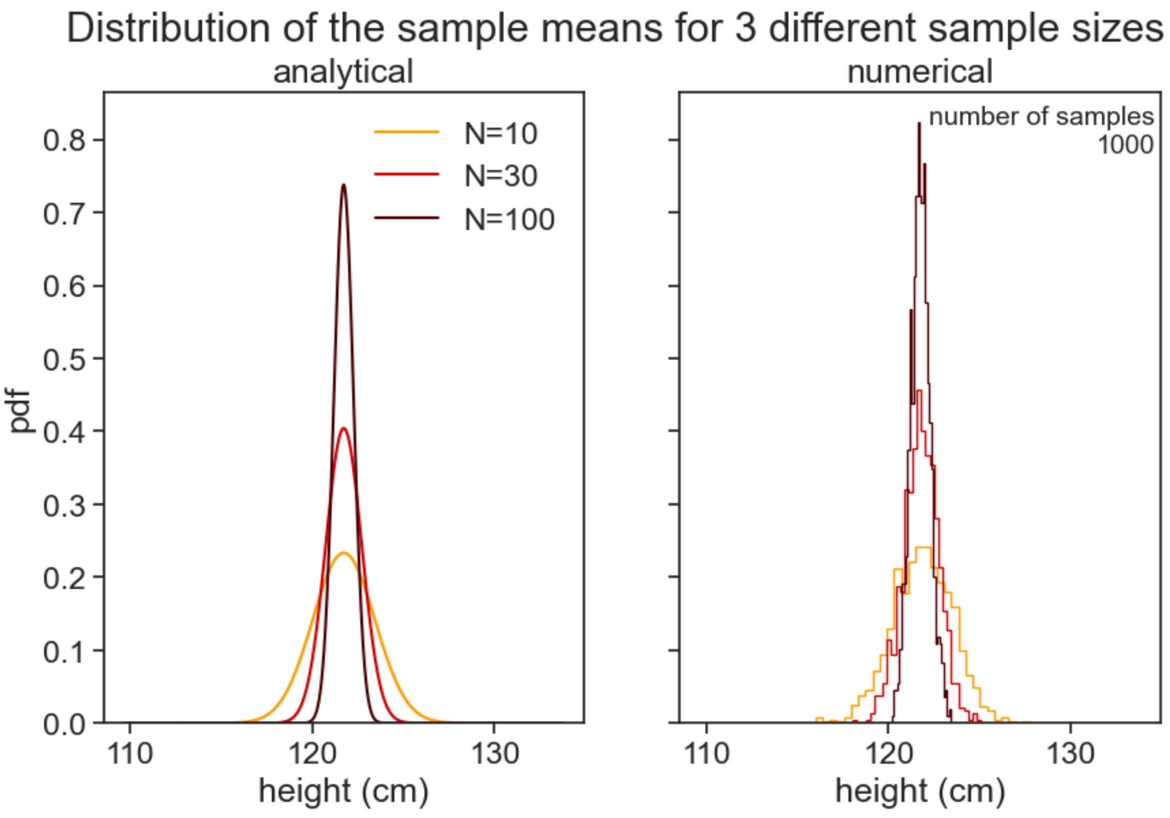
$$\bar{X} \sim N\left(\mu, \frac{\sigma^2}{N}\right),$$

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']
```

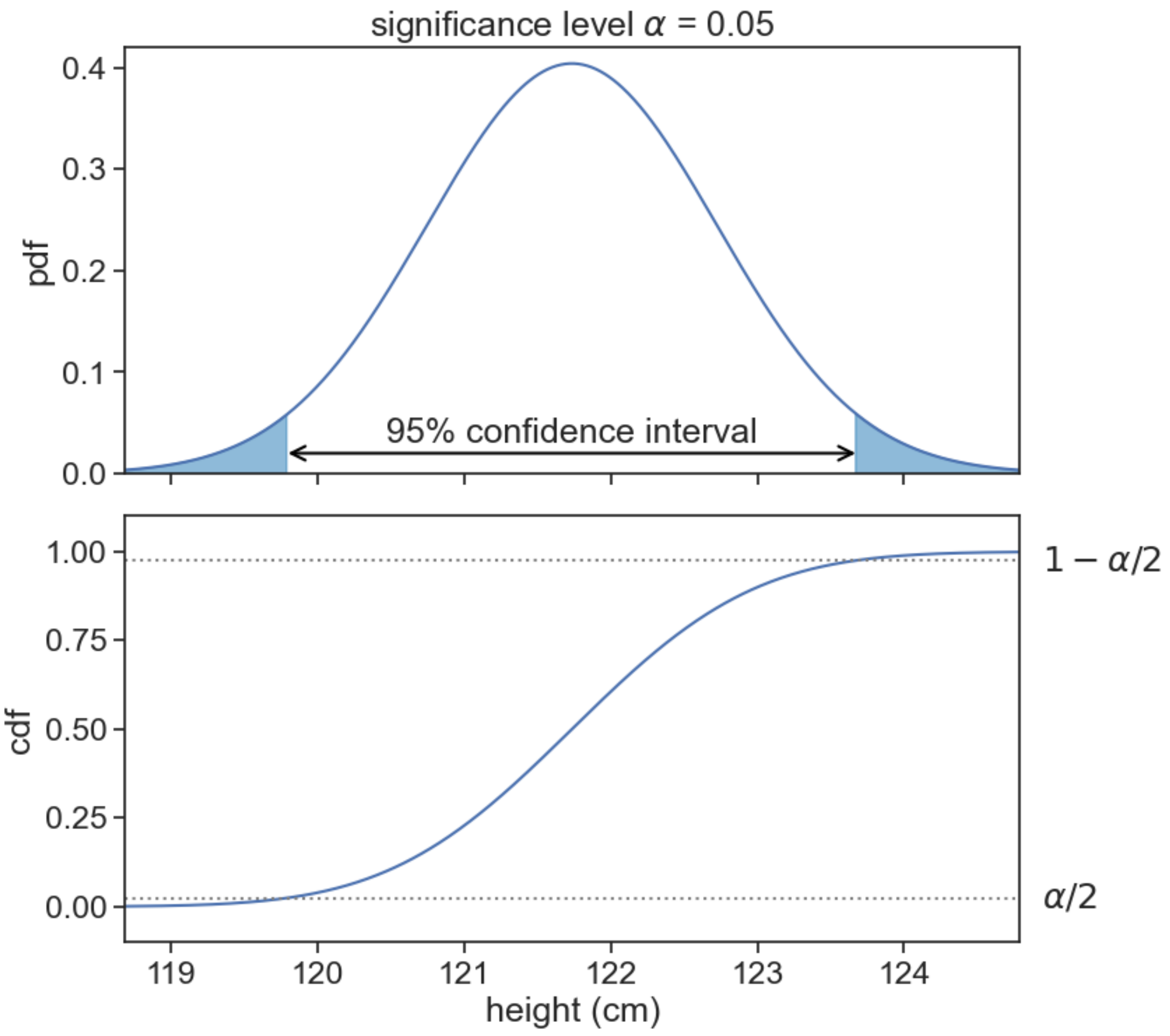
```
fig, ax = plt.subplots(1,2, figsize=(10, 6), sharex=True, sharey=True)

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

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



5.2 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_t
                   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=
               )
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.

5.3 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.

5.4 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
```

5.5 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).

6 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.

6.1 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.

6.2 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?

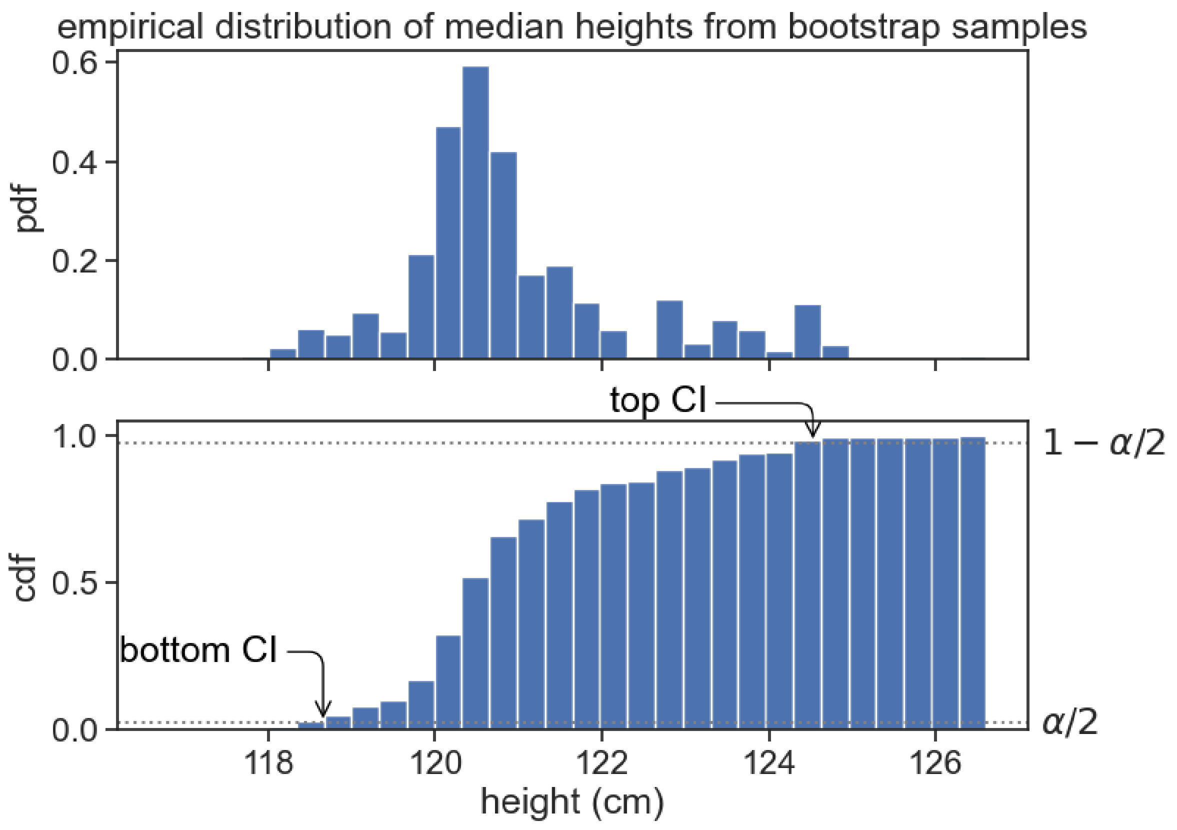
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="ticks", font_scale=1.5)
from scipy.stats import norm, ttest_ind, t
import scipy
# %matplotlib widget
```

```python
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']
```

```python
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
```

```python
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

7 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.

7.1 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.

7.2 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.

8 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?

8.1 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?

8.2 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.

8.3 example

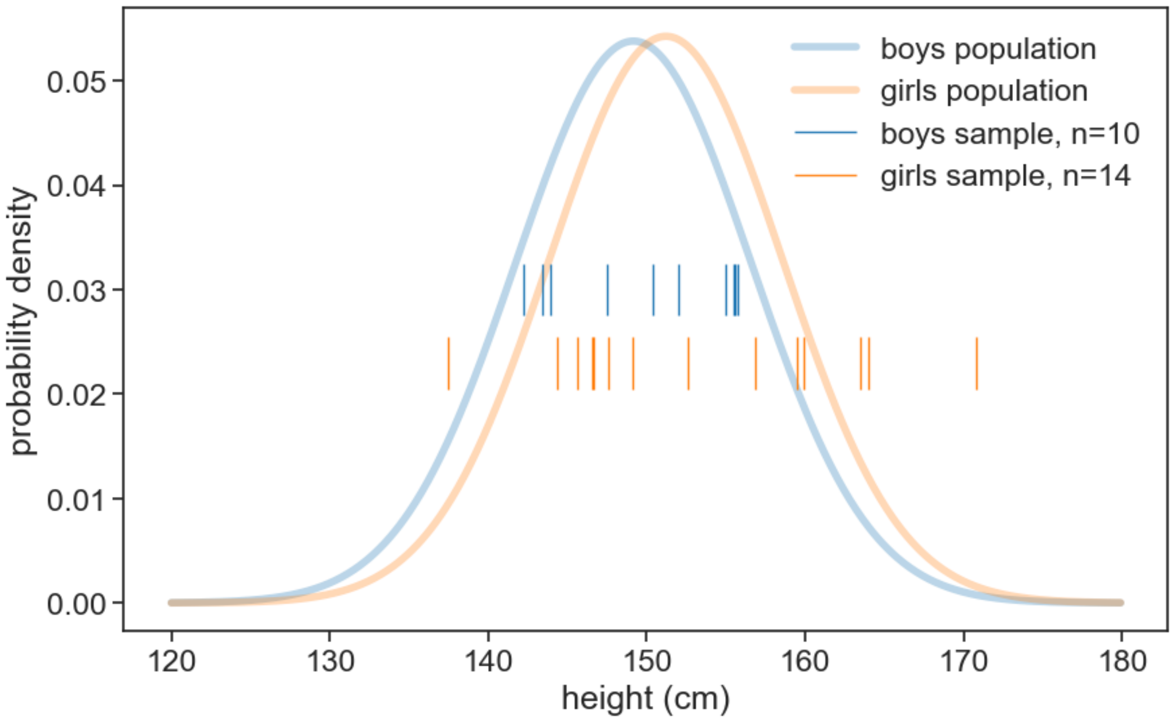
Let's use the very same example as in the independent samples t-test.

```
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.

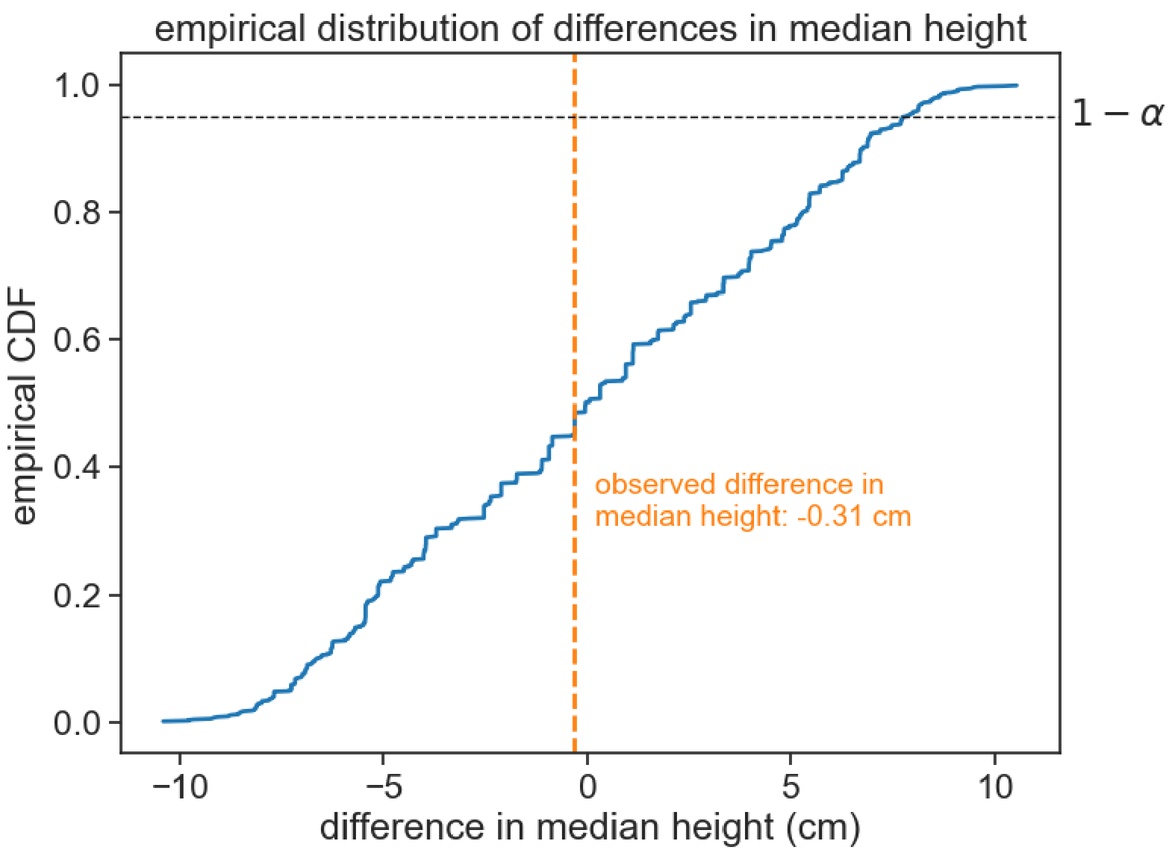
```
# define the desired statistic.
# in can be anything you want, you can even write your own function.
statistic = np.median
# compute the median for each sample and the difference
median_girls = statistic(sample_girls)
median_boys = statistic(sample_boys)
observed_diff = median_girls - median_boys
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:.2
        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.

8.4 increase sample size

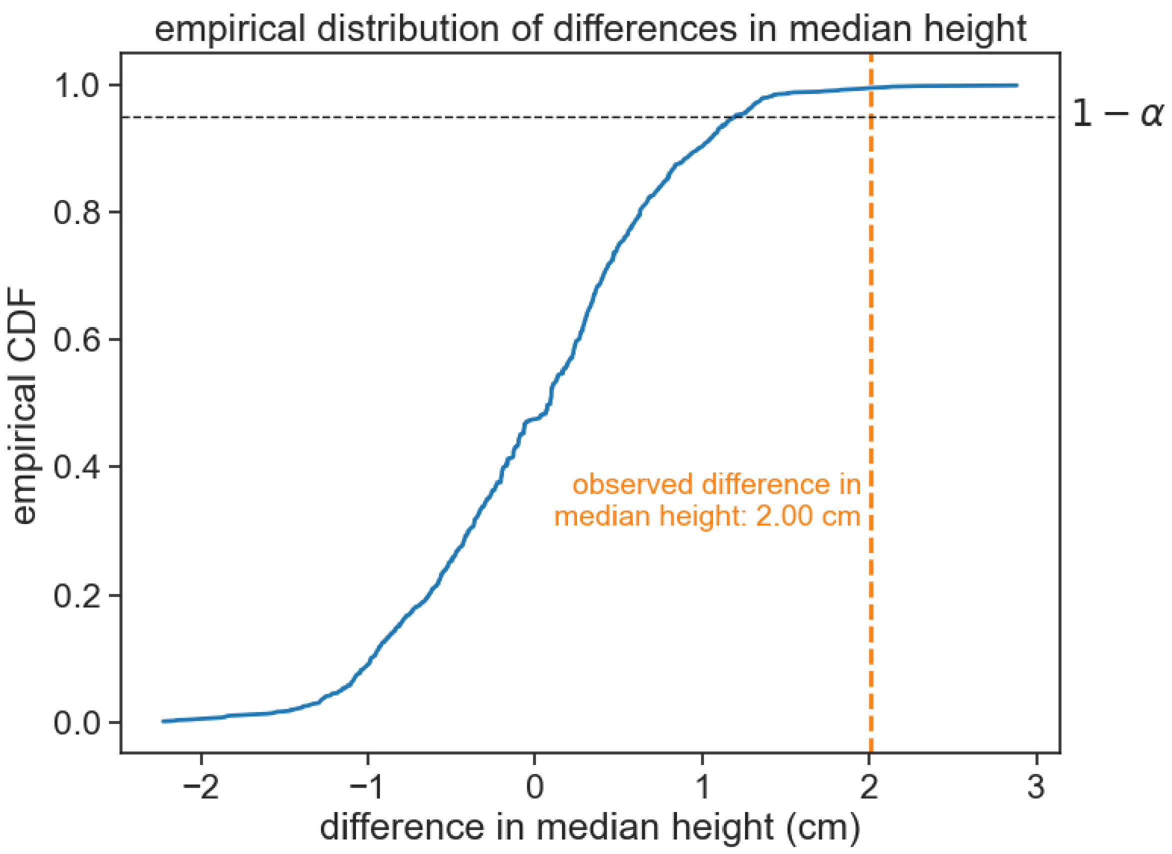
Let's increase the sample size to 200 girls and 300 boys.

```
# 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:.
        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.

8.5 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).

9 numpy vs pandas

9.1 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`.

9.2 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

	sex	height (cm)	weight (kg)
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)
```

10 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 imppractical 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.

10.1 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.

```python
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
```

```python
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']
```

```python
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)
```

```python
# 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
```

```python
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.

10.2 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))
```

```python
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

11 the geometry of regression

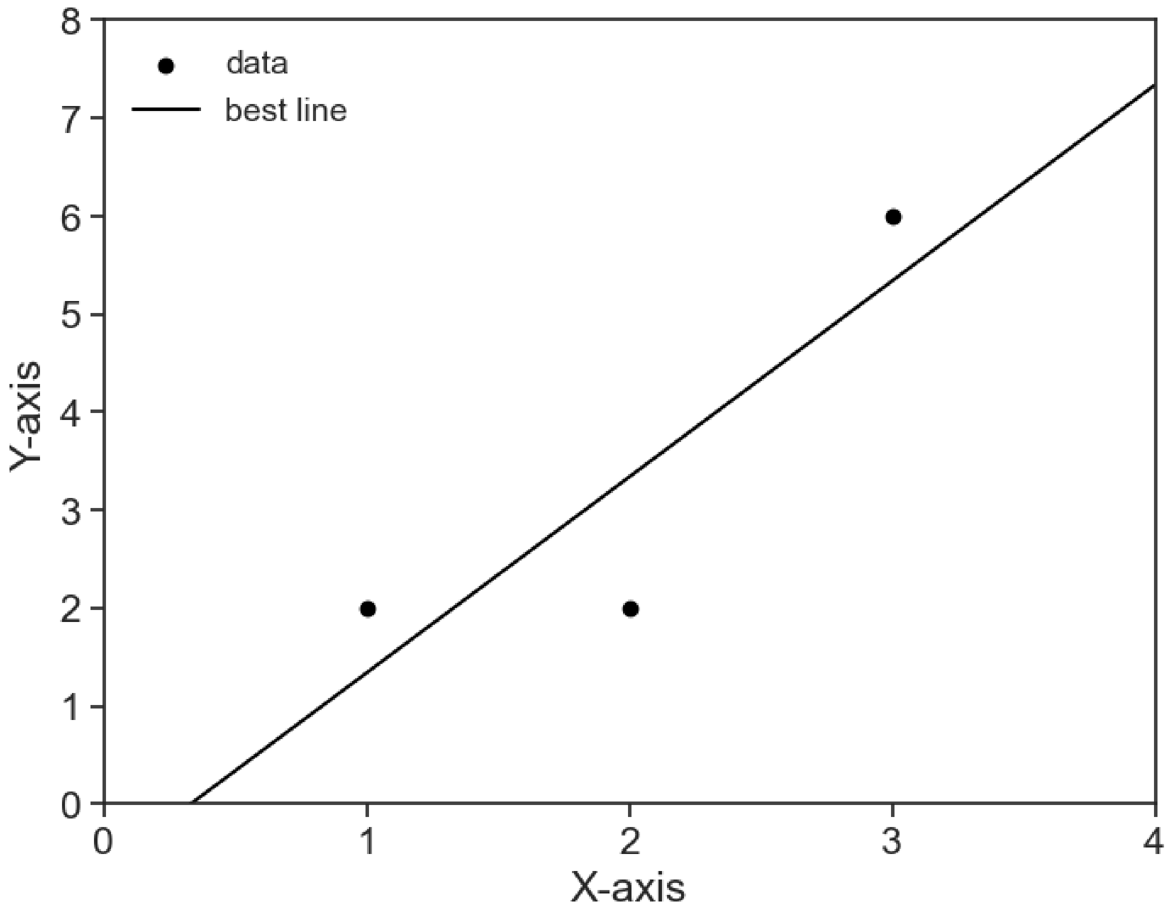
11.1 a very simple example

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:

```python
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
```

```python
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');
```



11.2 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?

11.3 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 $\hat{y}$:

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

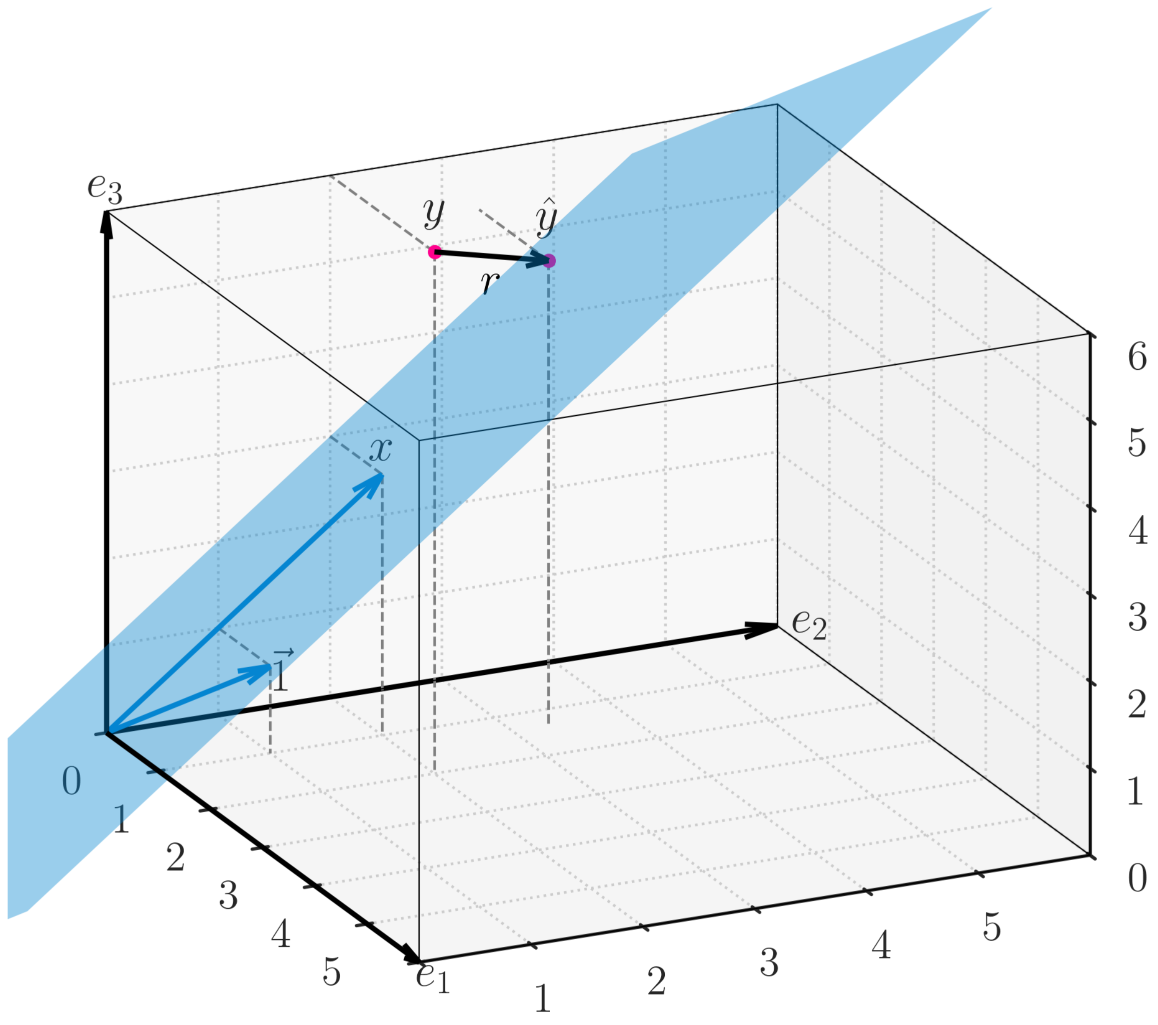
where β is the column vector $(\beta_0, \beta_1)^T$.

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.

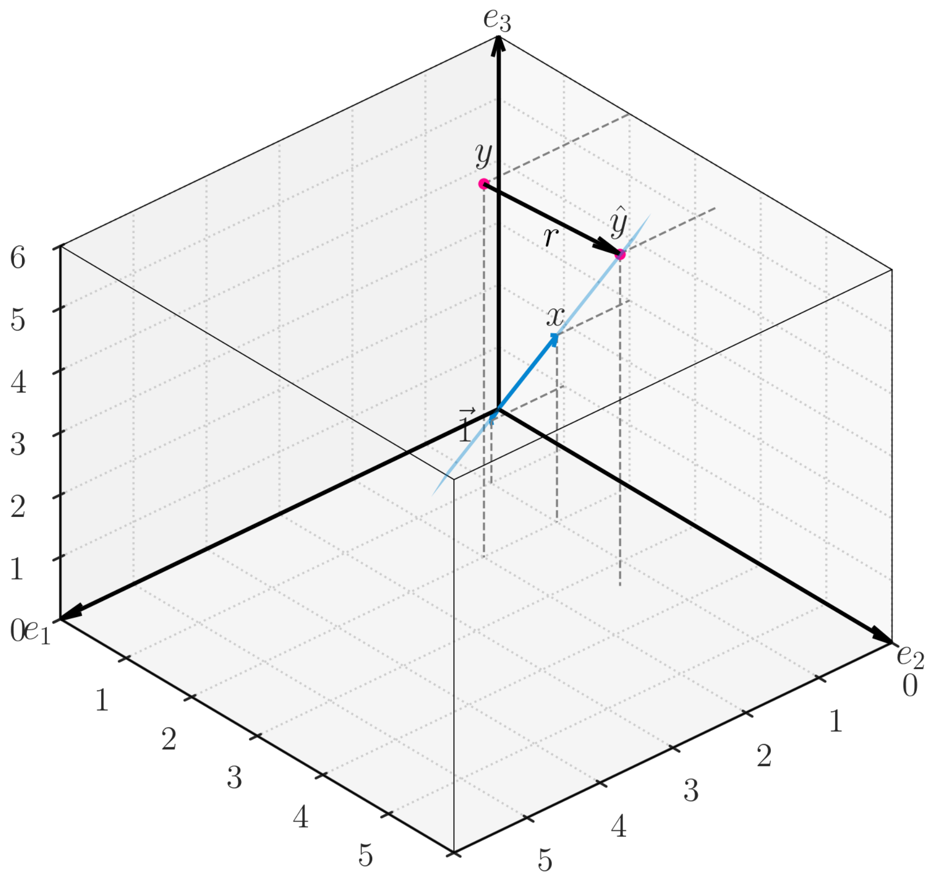
It will be extremely improbable that the vector y will also lie on this plane, so we will have to find the best prediction $\hat{y}$ that lies on this plane. Geometrically, our goal is to find the point $\hat{y}$ on the plane that is **closest** to the point y in the 3d space.

- 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.

11.4 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.

11.5 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

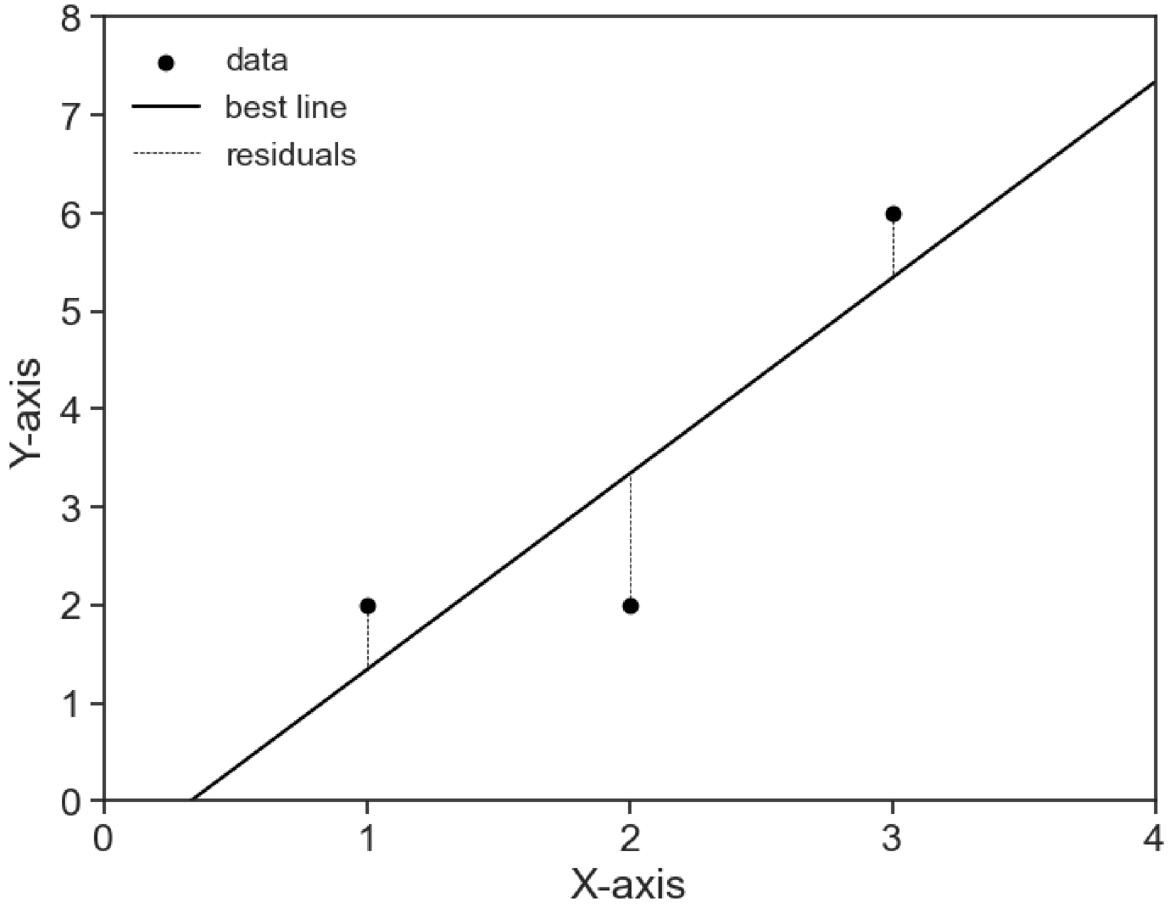
$$r = y - \hat{y}.$$

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.

$$
\begin{aligned}
\|r\|^2 &= \|y - \hat{y}\|^2 && (11.1)\\
&= (y_0 - \hat{y}_0)^2 + (y_1 - \hat{y}_1)^2 + \cdots + (y_{n-1} - \hat{y}_{n-1})^2 && (11.2)\\
&= r_0^2 + r_1^2 + \cdots + r_{n-1}^2 && (11.3)
\end{aligned}
$$

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');
```



11.6 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.

12 least squares

12.1 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)
```

12.2 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:

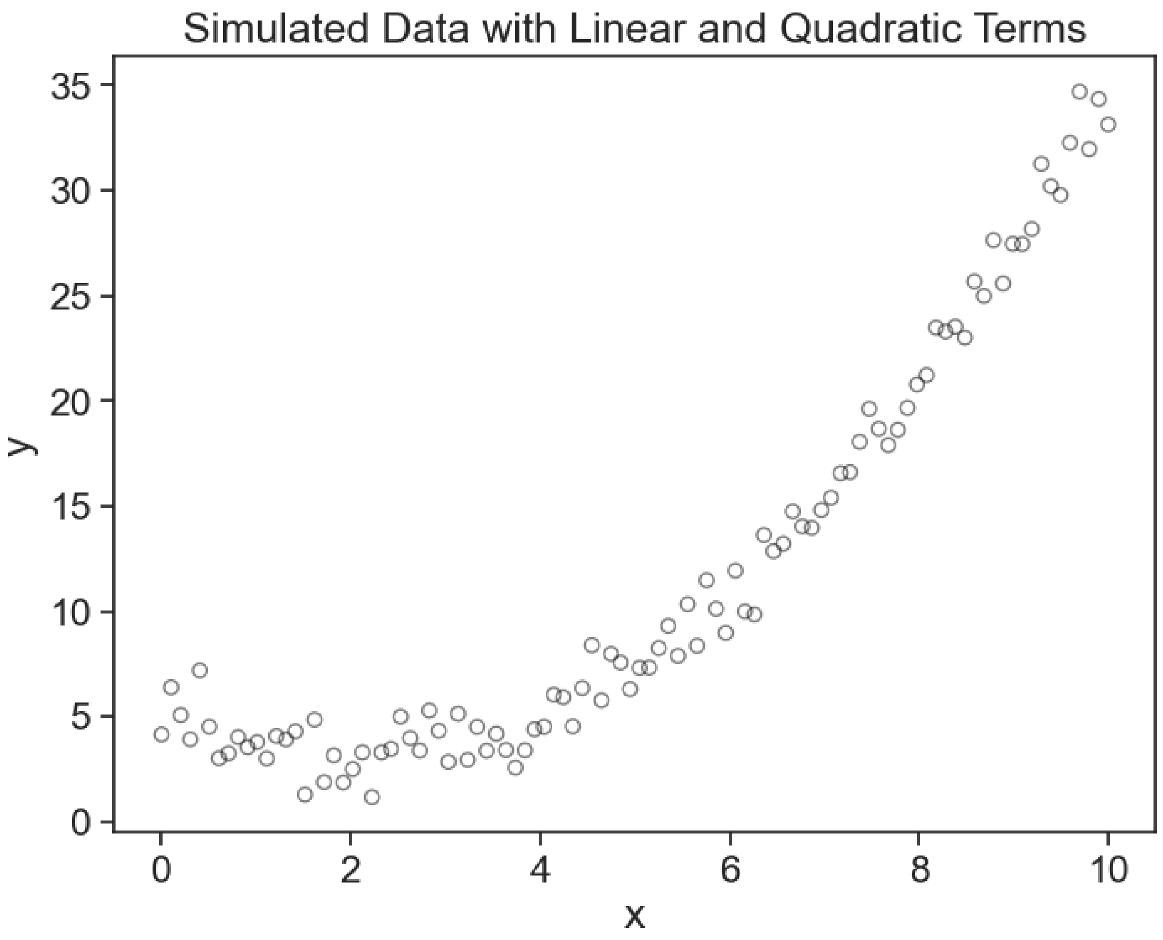
$$y = X\beta + e$$

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:

```python
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'
       );
```



12.3 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.

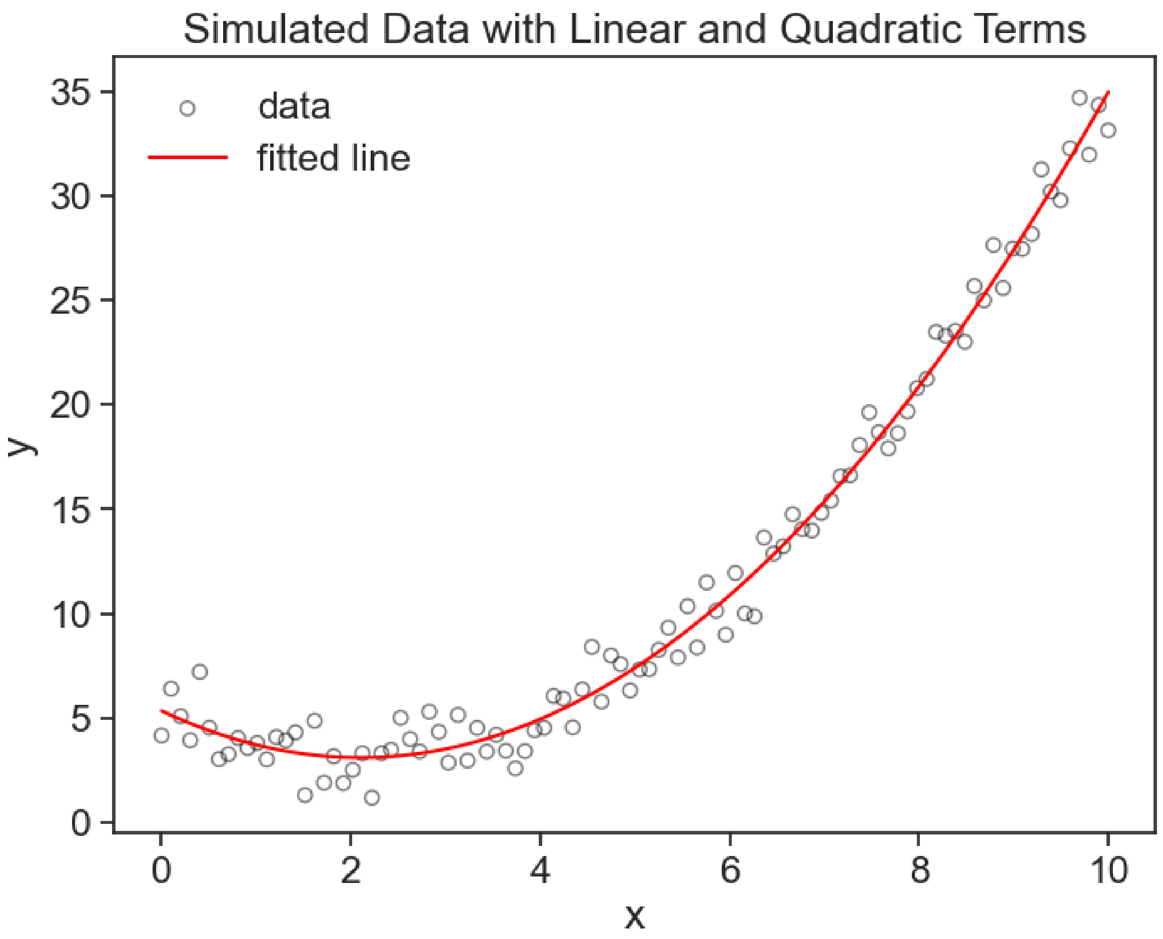
Now let's see if the parameters we found are any good.

```
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.

```python
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]
```

12.4 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.

12.5 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.

```python
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.

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

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.

12.6 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:

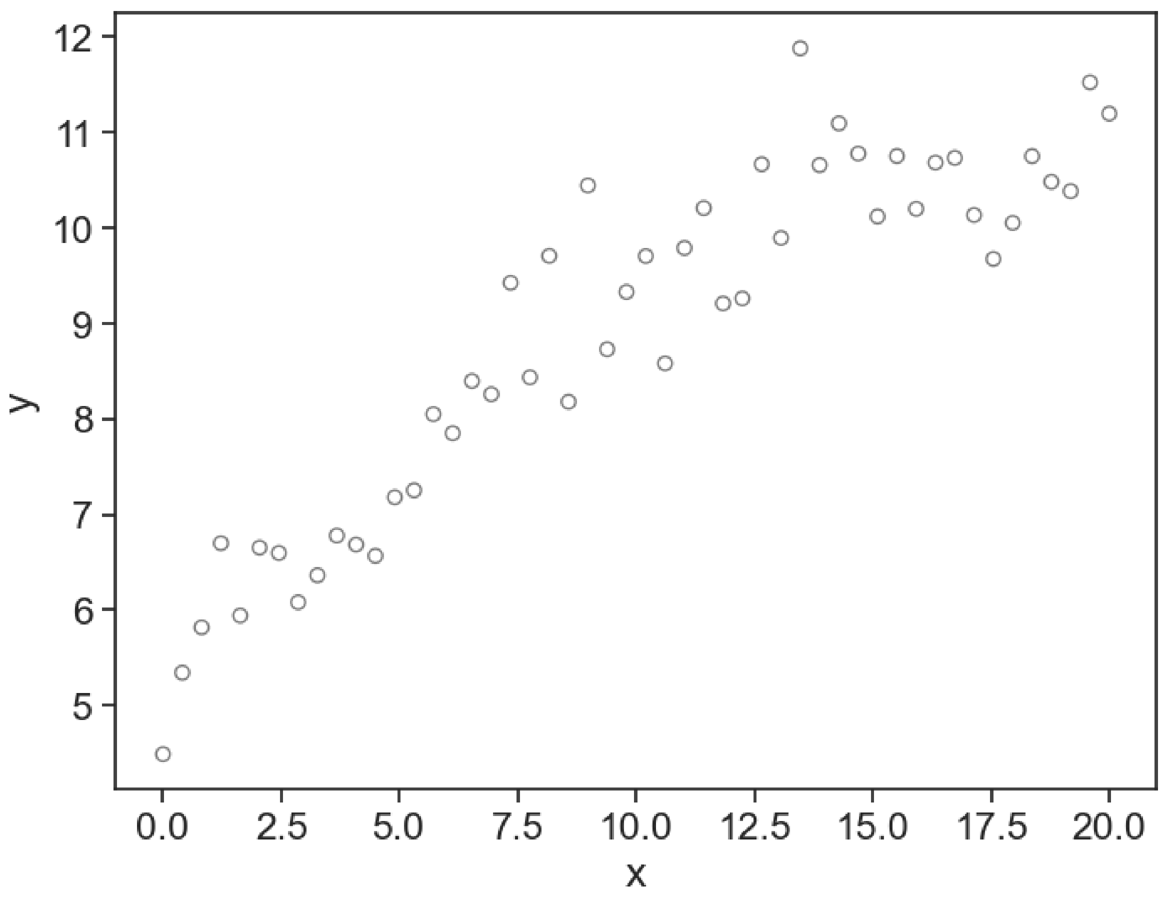
$$y = \beta_0 + e^{\beta_1 x} + \sin(\beta_2 x^2)$$

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())
```

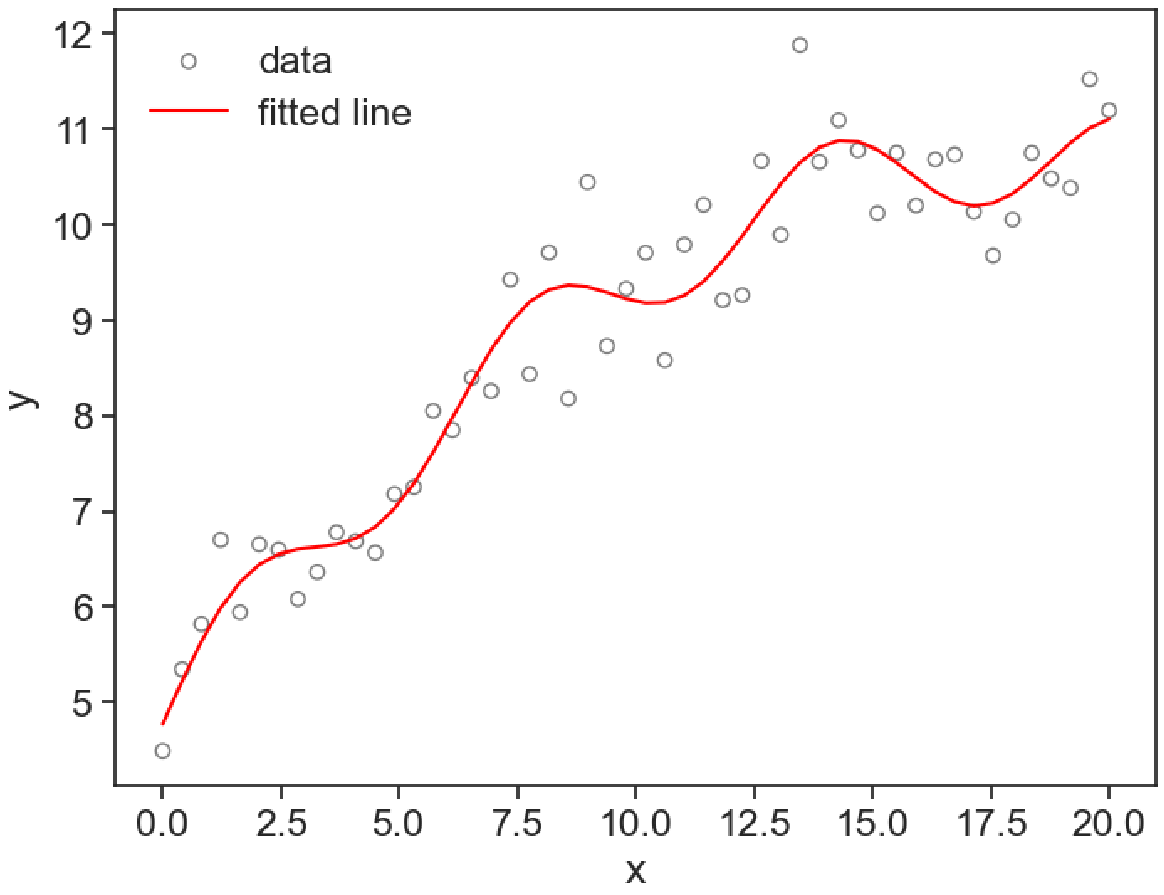
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.933
Model:                            OLS   Adj. R-squared:                  0.928
Method:                 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.

```python
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

13 equivalence

13.1 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.

13.2 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:

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

13.3 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 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.

14.1 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.norm
        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.

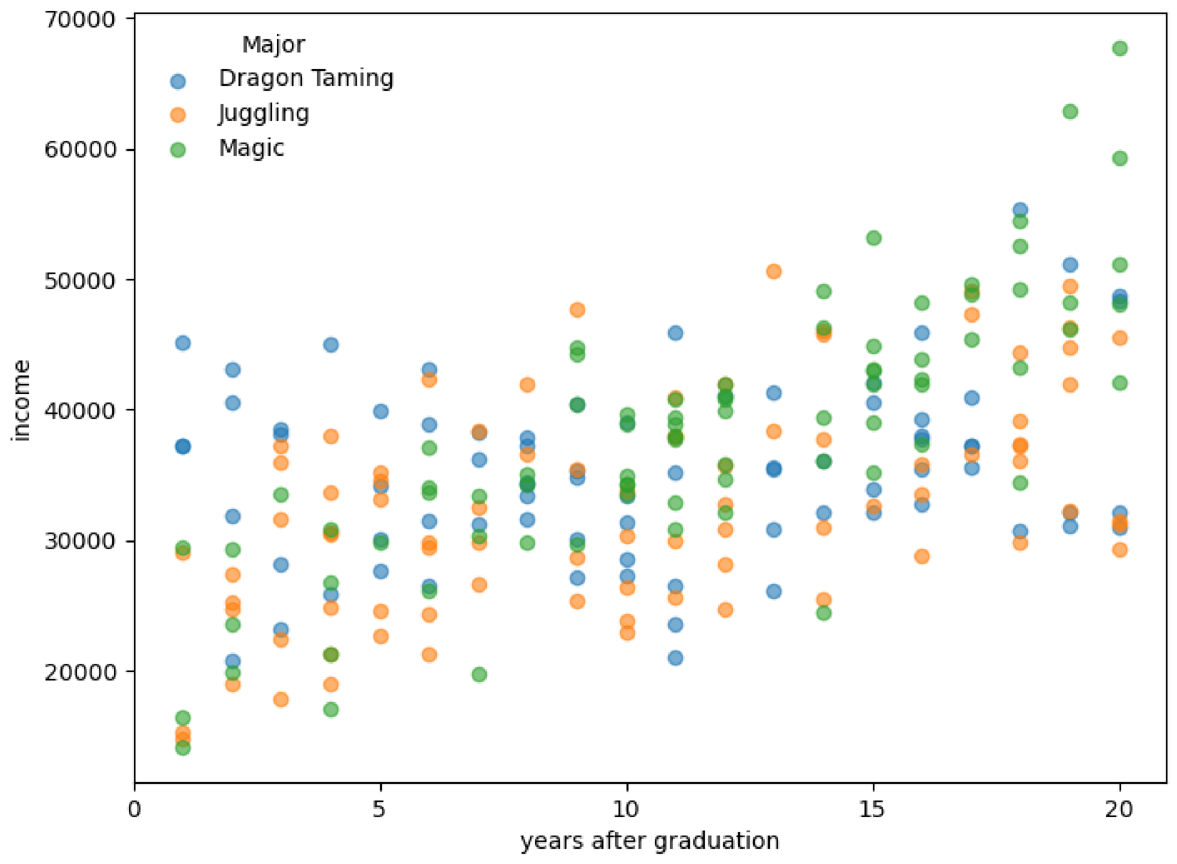
```
print(df[:5])
print(df[90:95])
print(df[190:195])
```

```
  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.

```python
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

14.2 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:

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

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.

14.3 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.

14.4 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=Tr
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:                         10690821.71
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 27851.5
major[T.Magic]                             -2999.754 1995.748 -1.503 0.133 -6911.348   911.8
major[T.Dragon Taming]                      5579.198 1954.661  2.854 0.004  1748.133  9410.2
years_after_grad                             723.745   72.028 10.048 0.000   582.573   864.9
years_after_grad:major[T.Magic]              635.180  109.599  5.795 0.000   420.370   849.9
years_after_grad:major[T.Dragon Taming]     -295.862  106.315 -2.783 0.005  -504.235   -87.4
Group Var                               33814137.626 2268.953
============================================================================================
```

14.5 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"
)
```

```python
result = model.fit()
print(result.summary())
```

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

```
                            Mixed Linear Model Regression Results
==================================================================================================
Model:                        MixedLM           Dependent Variable:            income
No. Observations:             239               Method:                        REML
No. Groups:                   90                Scale:                         10125672.168
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 27501.57
major[T.Magic]                           -2805.540 1811.051 -1.549 0.121 -6355.135   744.05
major[T.Dragon Taming]                    5980.367 1767.166  3.384 0.001  2516.786  9443.94
years_after_grad                           731.399   84.211  8.685 0.000   566.349   896.45
years_after_grad:major[T.Magic]            611.065  126.072  4.847 0.000   363.969   858.16
years_after_grad:major[T.Dragon Taming]   -329.530  122.977 -2.680 0.007  -570.561   -88.49
Group Var                             22392488.656 1835.422
Group x years_after_grad Cov             90328.607   75.664
years_after_grad Var                     39074.487    7.401
==================================================================================================
```

14.6 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.

```python
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.
--------------------------------------------------------------------------------------------
Intercept                               2.486e+04   1450.267     17.141      0.000      2.2e
major[T.Magic]                         -4402.0846   2281.475     -1.929      0.055   -8897.
major[T.Dragon Taming]                  7696.8705   2167.061      3.552      0.000    3427.
years_after_grad                         778.4674    123.280      6.315      0.000     535.
years_after_grad:major[T.Magic]          758.4393    185.397      4.091      0.000     393.
years_after_grad:major[T.Dragon Taming] -510.1096    183.456     -2.781      0.006    -871.
==============================================================================
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.
```

Part VI

miscellaneous

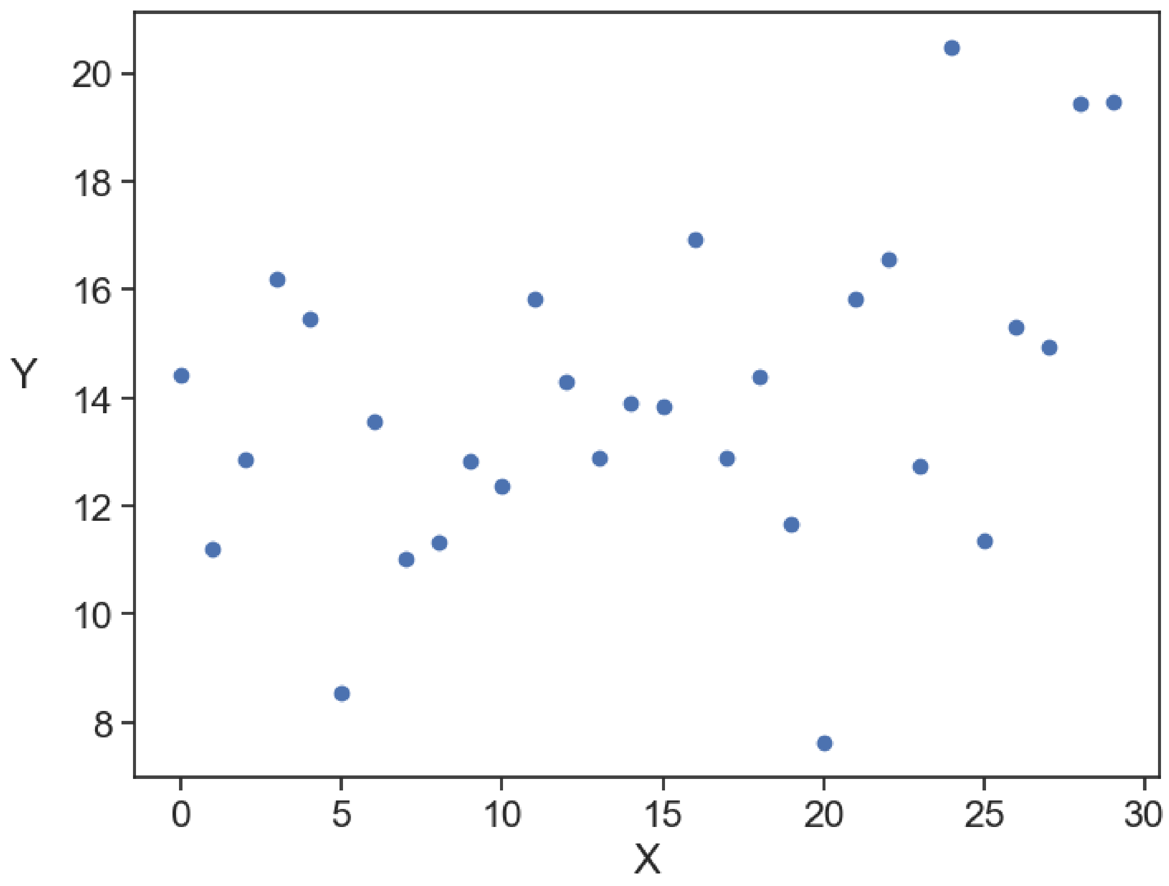
15 trend test

How to determine if there is a trend (positive or negative) between two variables?

```python
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)
```

```python
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)
```

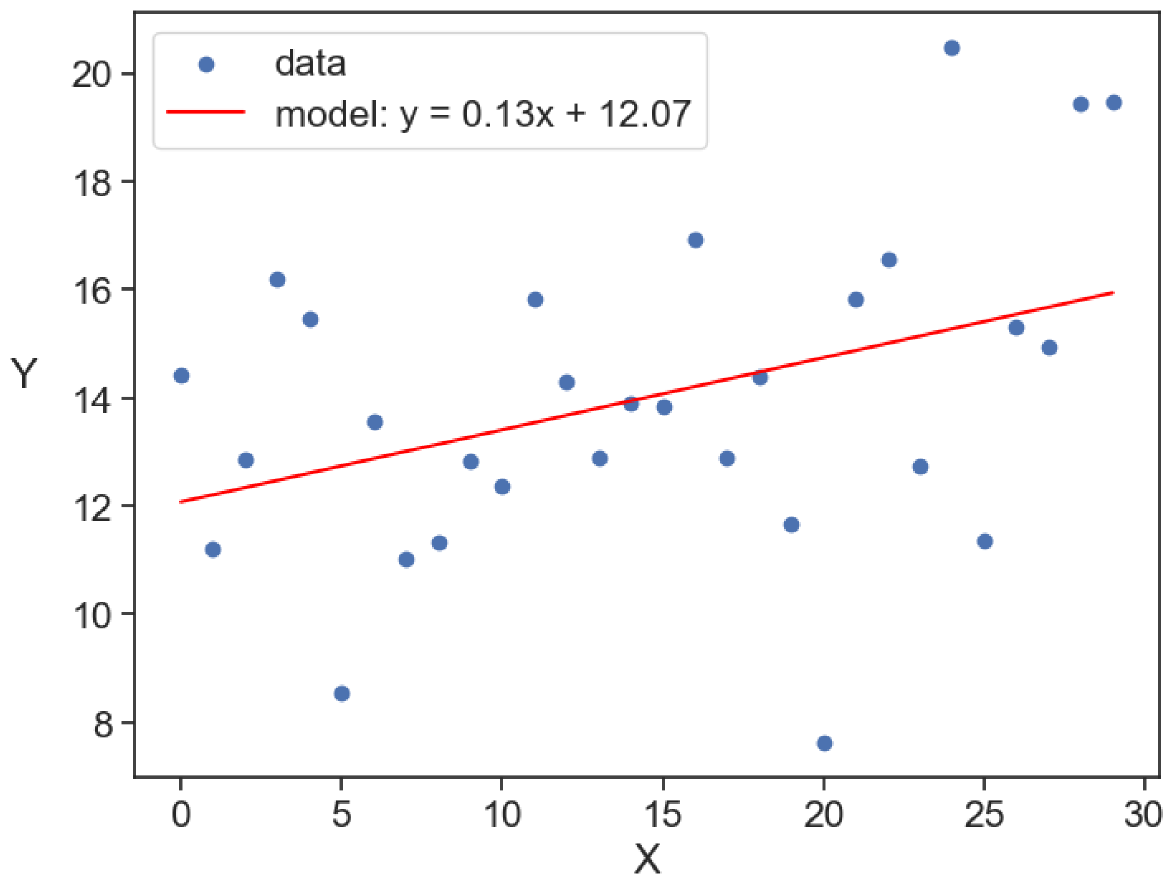
```
Text(0, 0.5, 'Y')
```



15.1 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.

```python
# 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
```

```python
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='grea
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.

15.2 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, Ta
```

The test concluded that there is an increasing trend, with a p-value of 0.0497.

15.3 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.

15.4 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.

```python
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
```