Statistics

Yair Mau

Invalid Date

Table of contents

Preface

I read Mike X Cohen's excellent book "Modern Statistics", and now it's time to practice.

Part I

data

1 height data
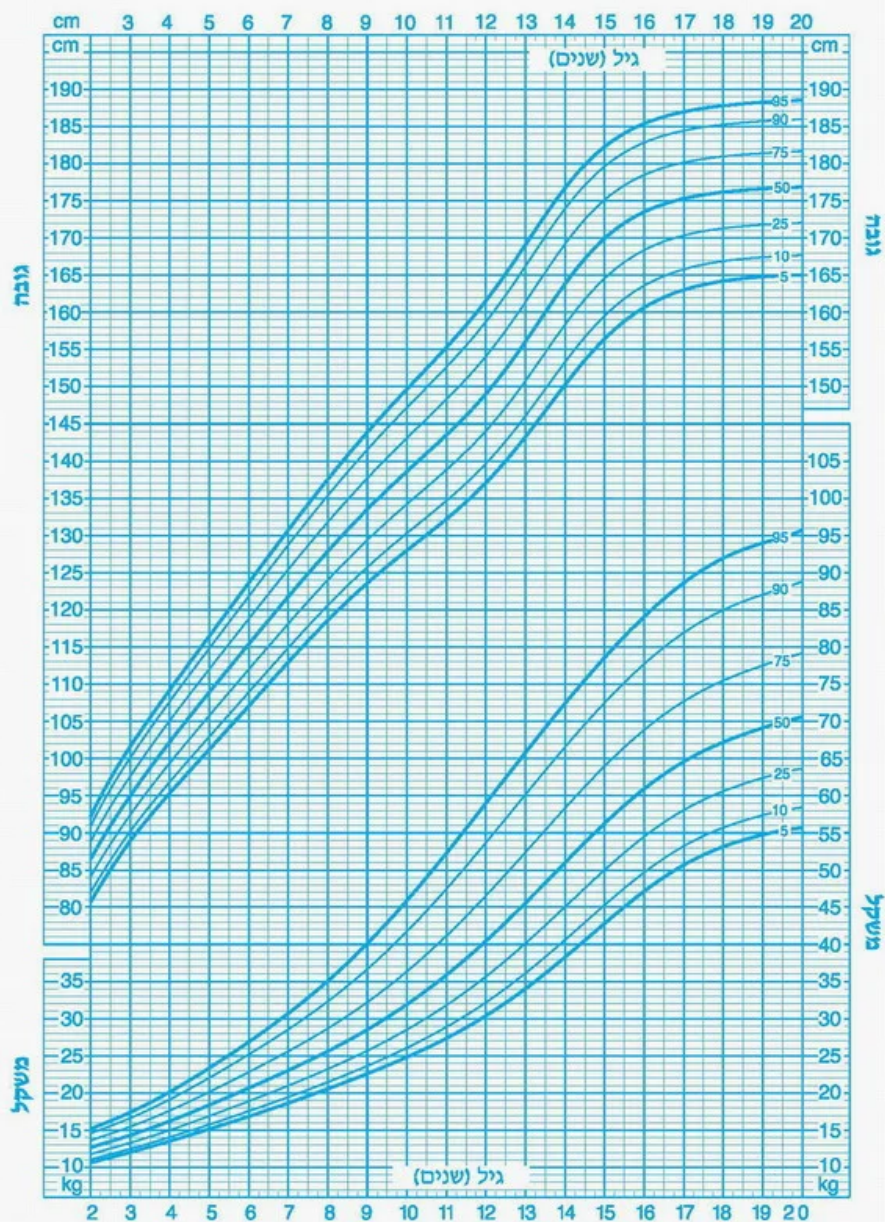
I found growth curves for girls and boys in Israel:

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

For example, see this:

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

בנים

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);
```

boys, 50th percentile
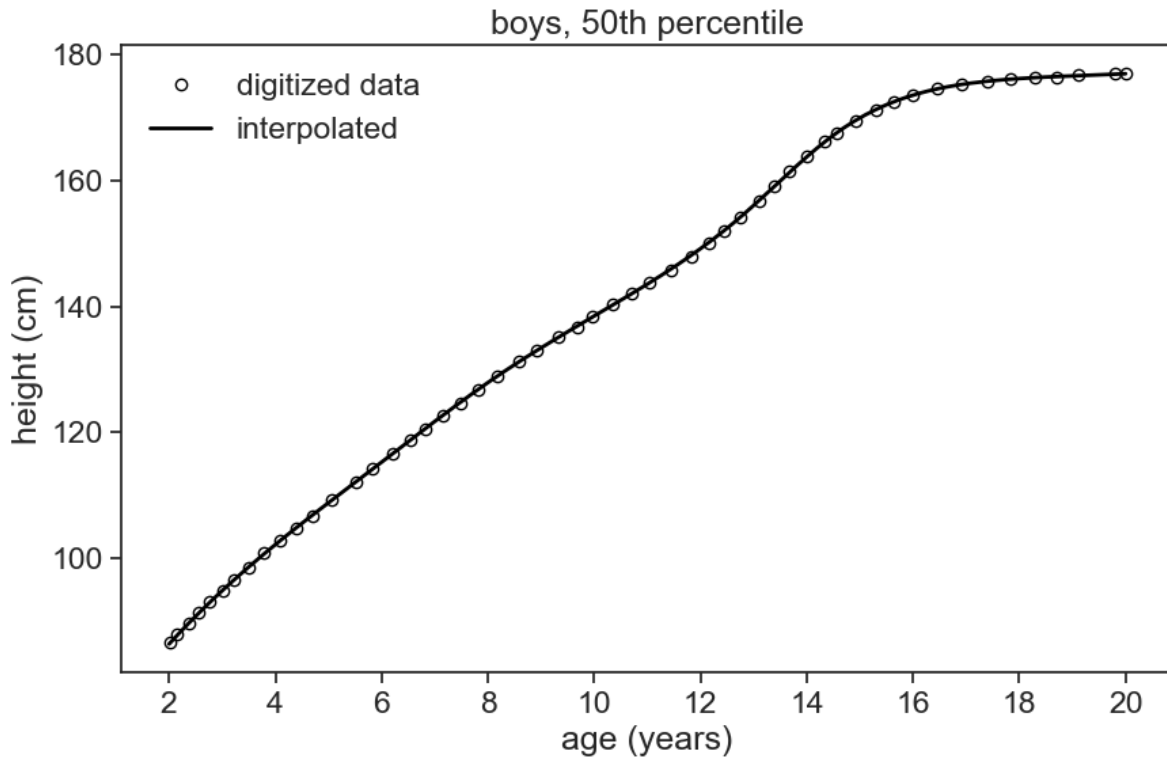
Let's do the same for all the other curves, and then save them to a file.

```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
```

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

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

growth curves for girls
percentile curves: 5, 10, 25, 50, 75, 90, 95
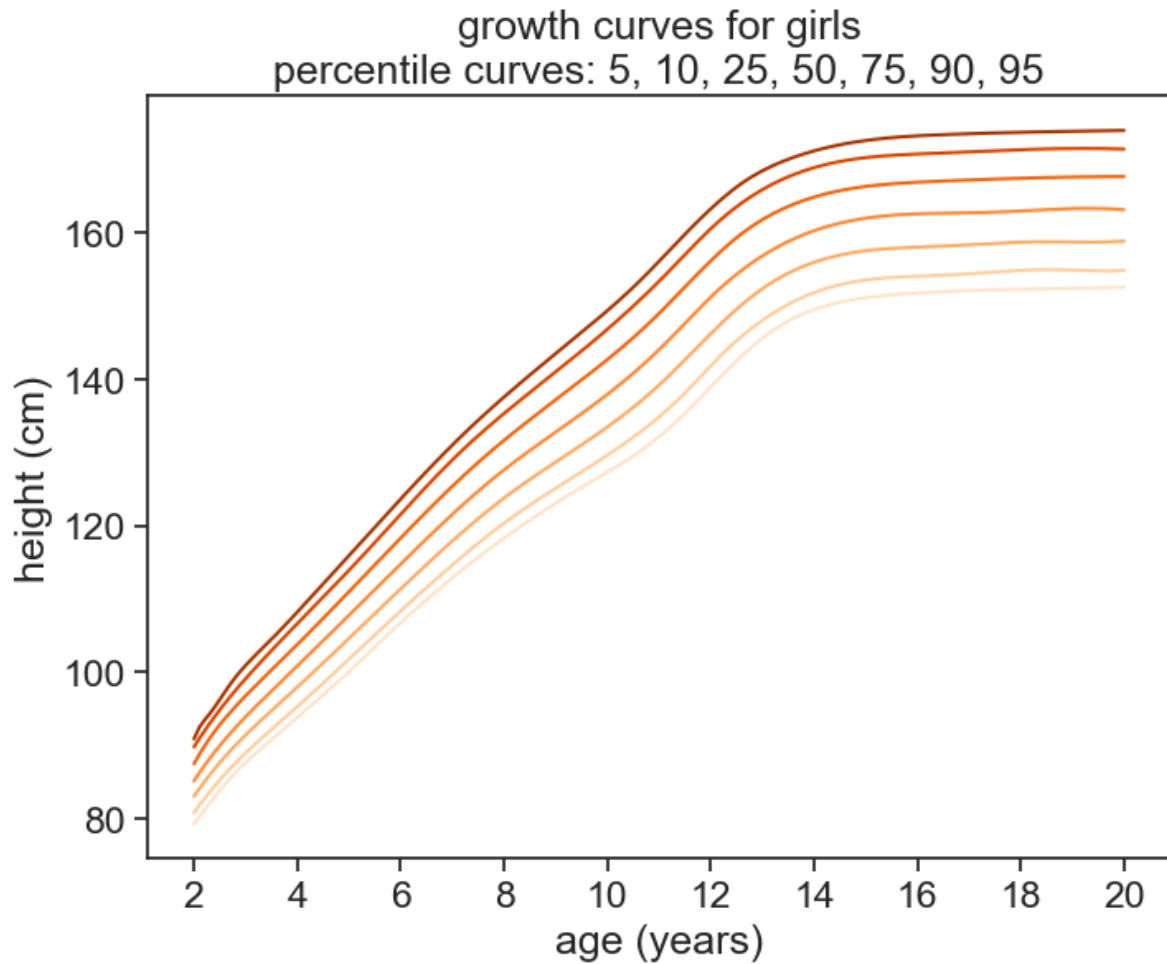
Let's now see the percentiles for girls age 20.

```
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"
       );
```

cdf for girls, age 20
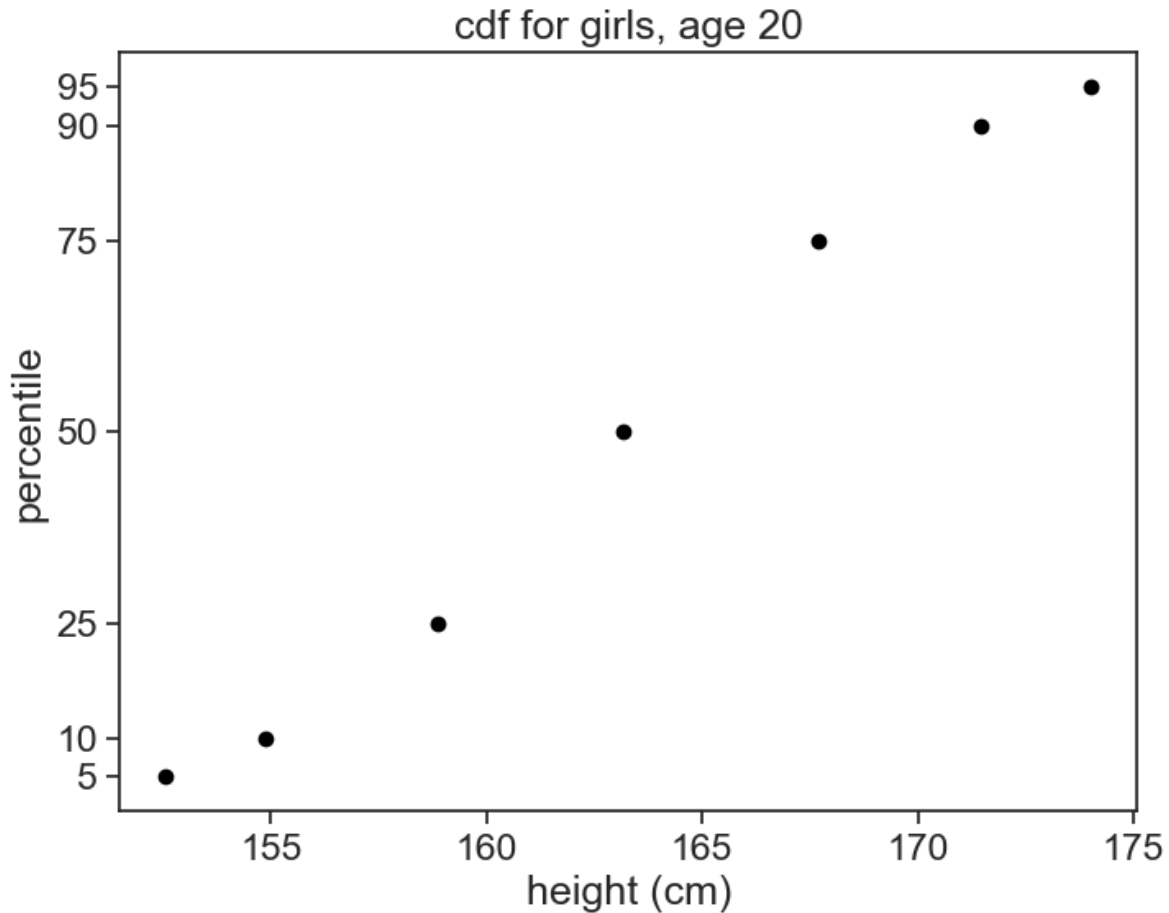
I suspect that the heights in the population are normally distributed. Let's check that. I'll fit the data to the integral of a gaussian, because the percentiles correspond to a cdf. If a pdf is a gaussian, its cumulative is given by

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

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

```python
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):
```

```python
        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)
```

```python
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)
```

```python
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"
         );
```

the data is very well fitted by a normal distribution

$\mu = 163.2$ cm
$\sigma = 6.5$ cm
$R^2 = 0.999947$
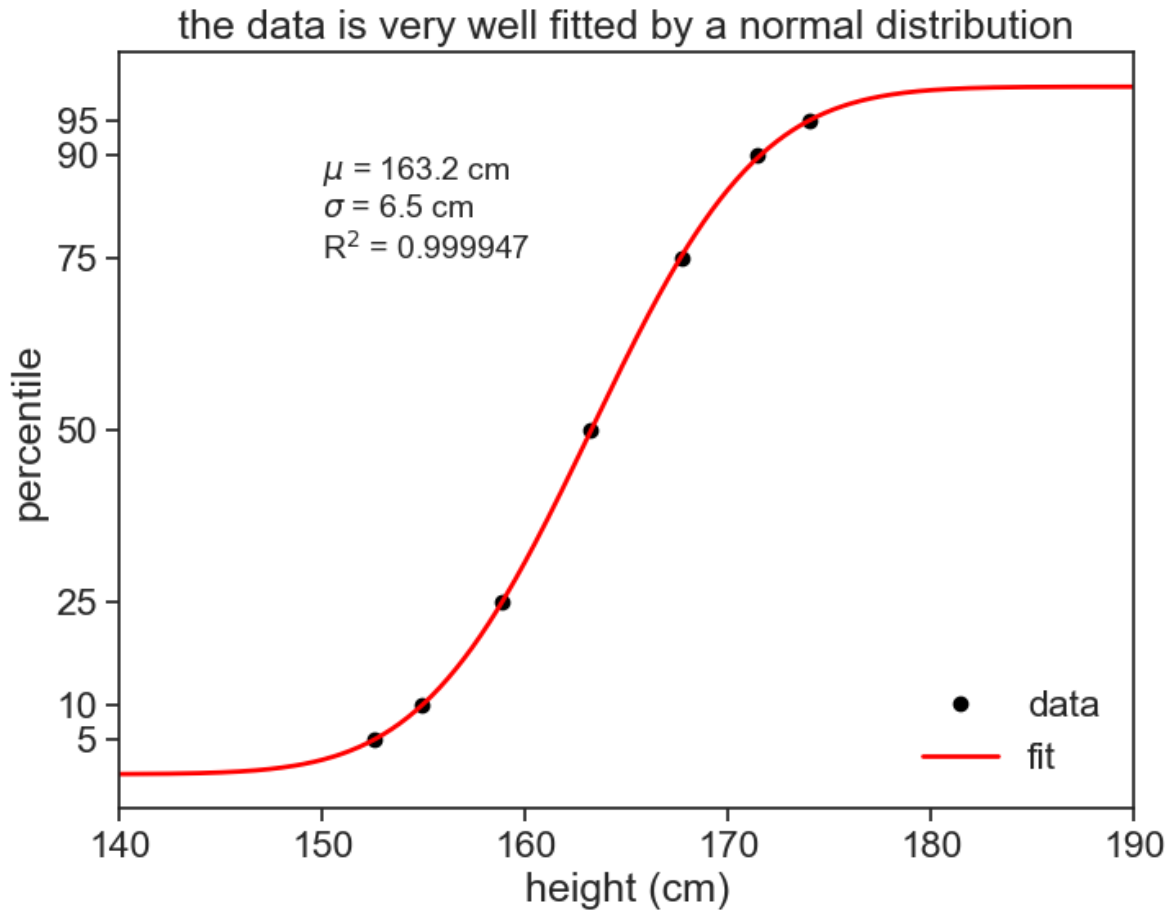
Another way of making sure that the model fits the data is to make a QQ plot. In this plot, the quantiles of the data are plotted against the quantiles of the normal distribution. If the data is normally distributed, the points should fall on a straight line.

```
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)
```


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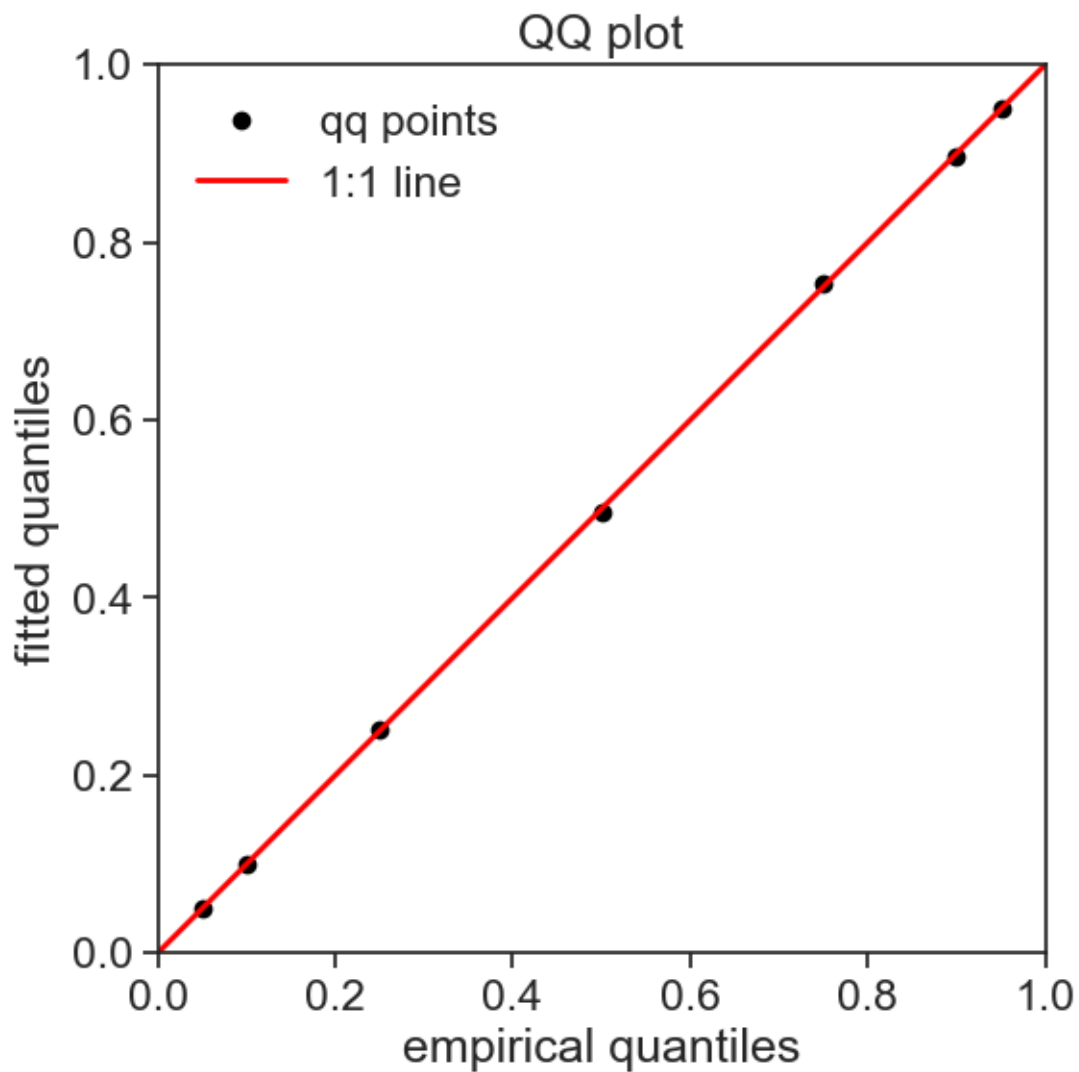
Great, now we just need to do exactly the same for both sexes, and all the ages. I chose to divide age from 2 to 20 into 0.1 intervals.

```
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'])
```

```python
df_stats_girls['mu'] = 0.0
df_stats_girls['sigma'] = 0.0
df_stats_girls['r2'] = 0.0
```

```python
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

```python
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),
         );
```
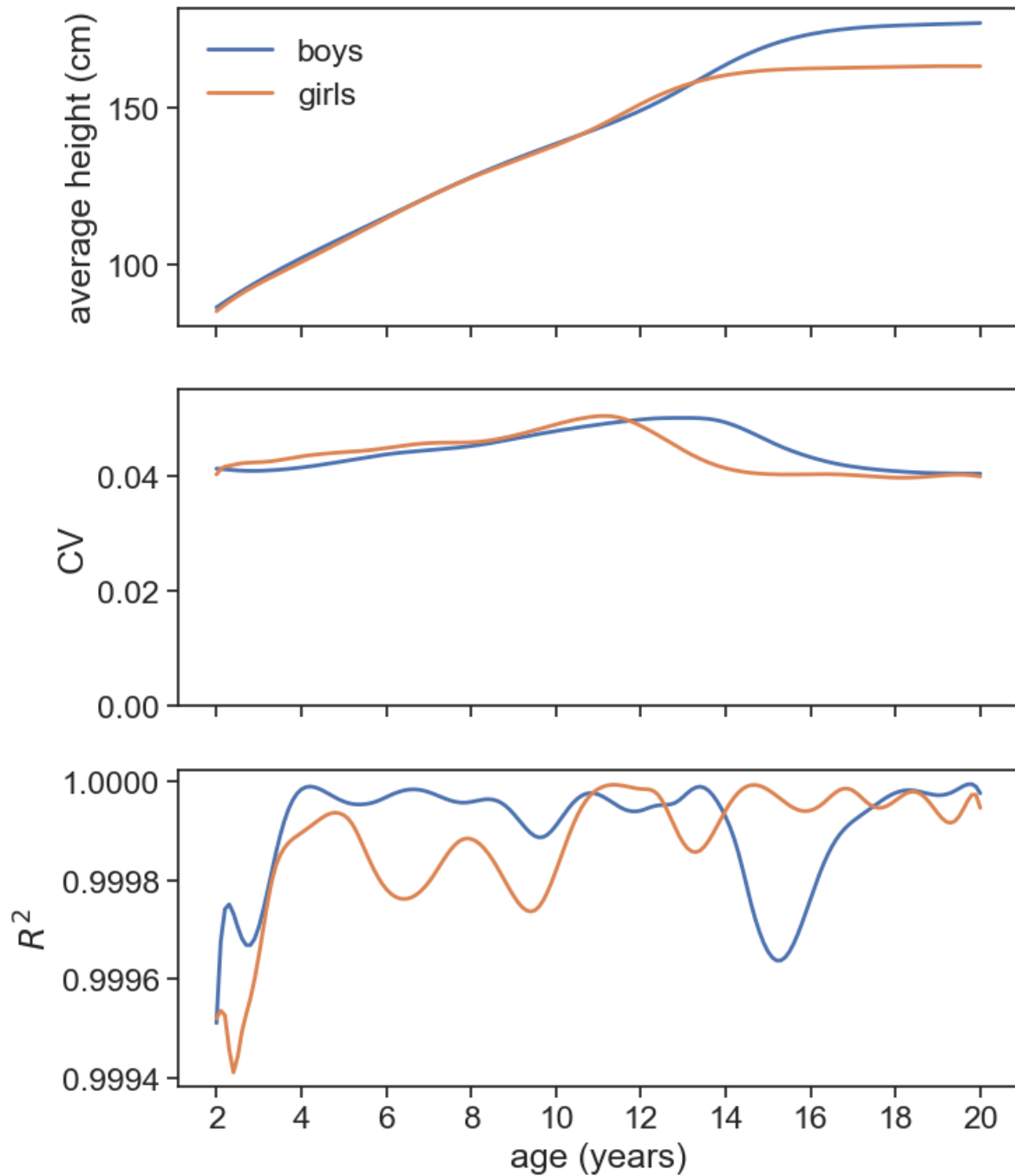
Let's see how the pdfs for boys and girls move and morph as age increases.

```
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.

```python
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),);
```
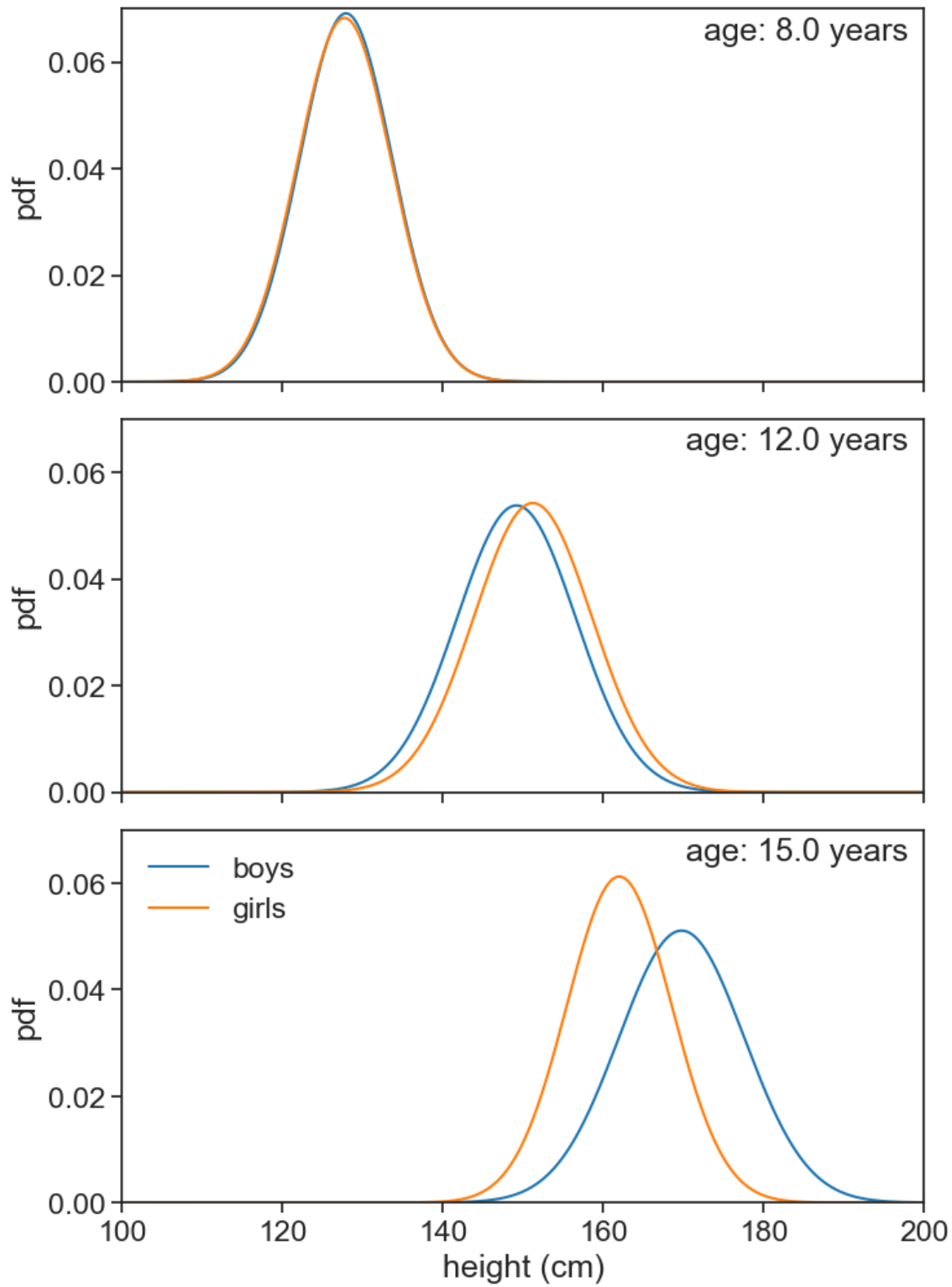
- Indeed, boys and girls age 8 have the exact same height distribution.
- 12-year-old girls are indeed taller than boys, on average. This difference is relatiely small, though.
- By age 15 boys have long surpassed girls in height, and the difference is quite large. Boys still have some growing to do, 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),
       );
```
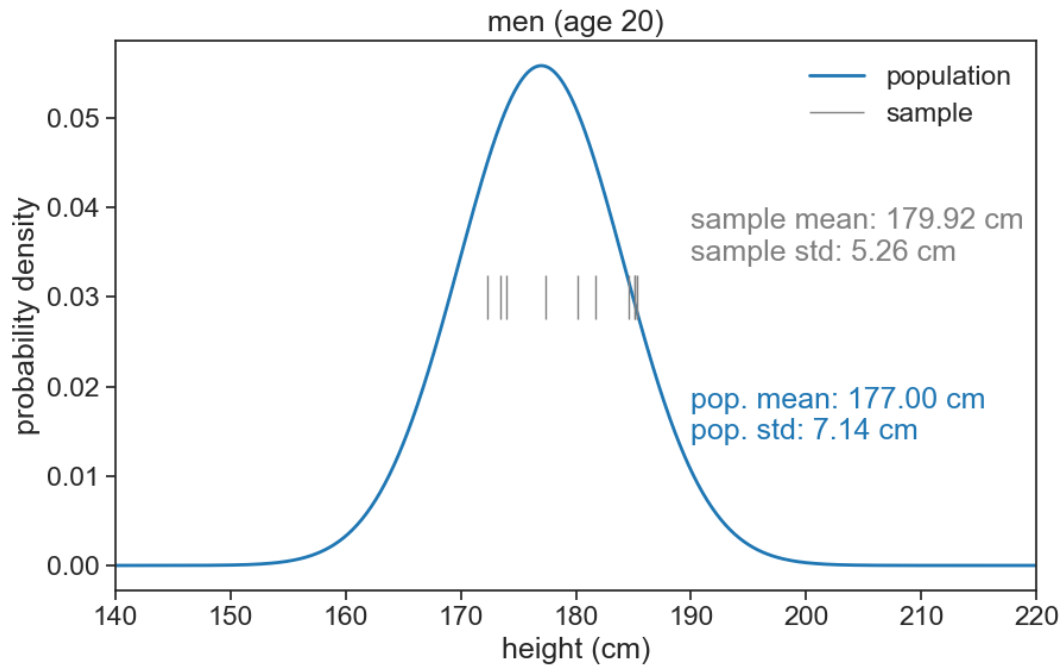
The t value is calculated as follows:

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

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.

```
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.

```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, 0.10),
                    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)",
```

```
    );
```

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),
       );
```

men (age 20)
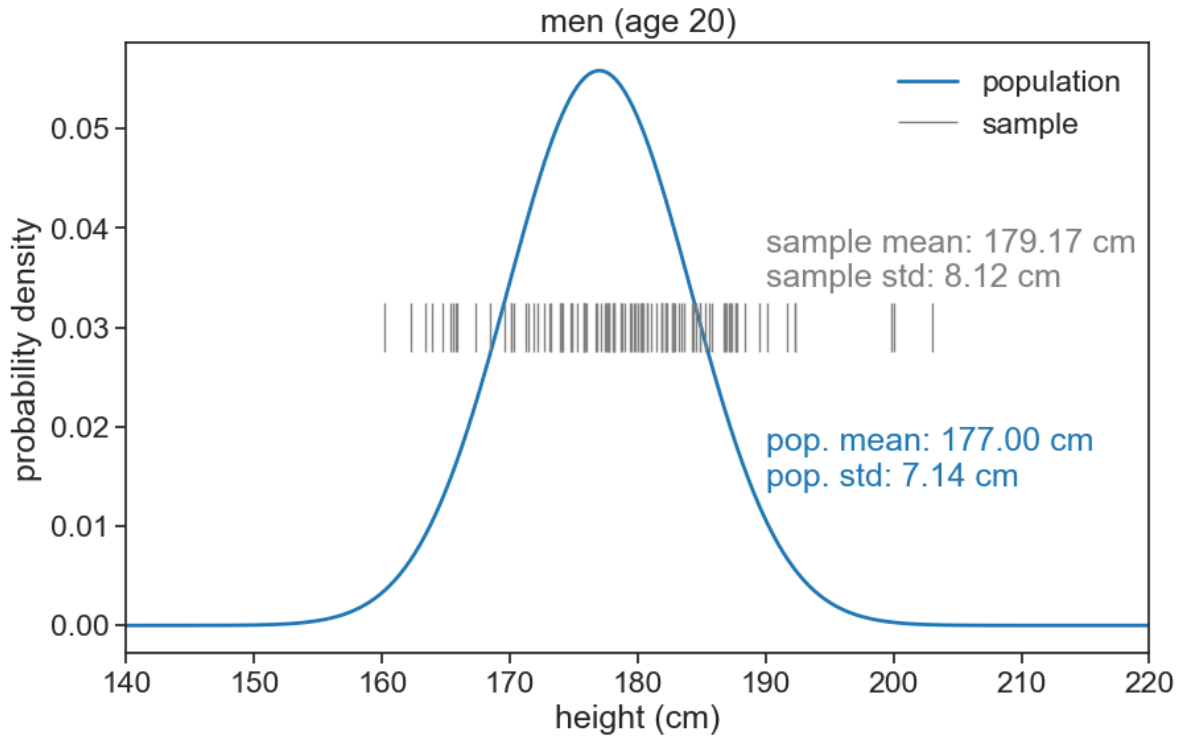

```
t_value_scipy = ttest_1samp(sample100, popmean=mu_boys)
print(f"t-value: {t_value_scipy.statistic:.3f}")
print(f"p-value: {t_value_scipy.pvalue:.3f}")
```

```
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,
```

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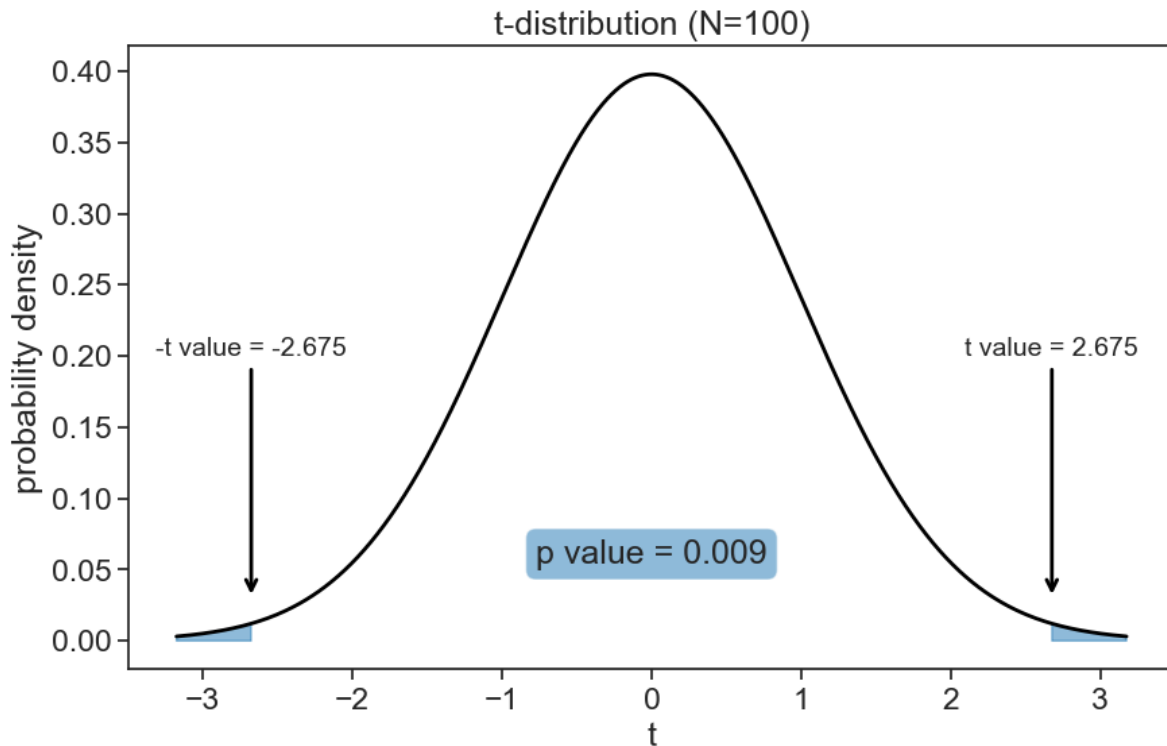
t-distribution (N=100)

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.

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

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

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

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

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

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

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

ax.set(xlabel='t',
            ylabel='probability density',
            title="t-distribution (N=100)",
            );
```
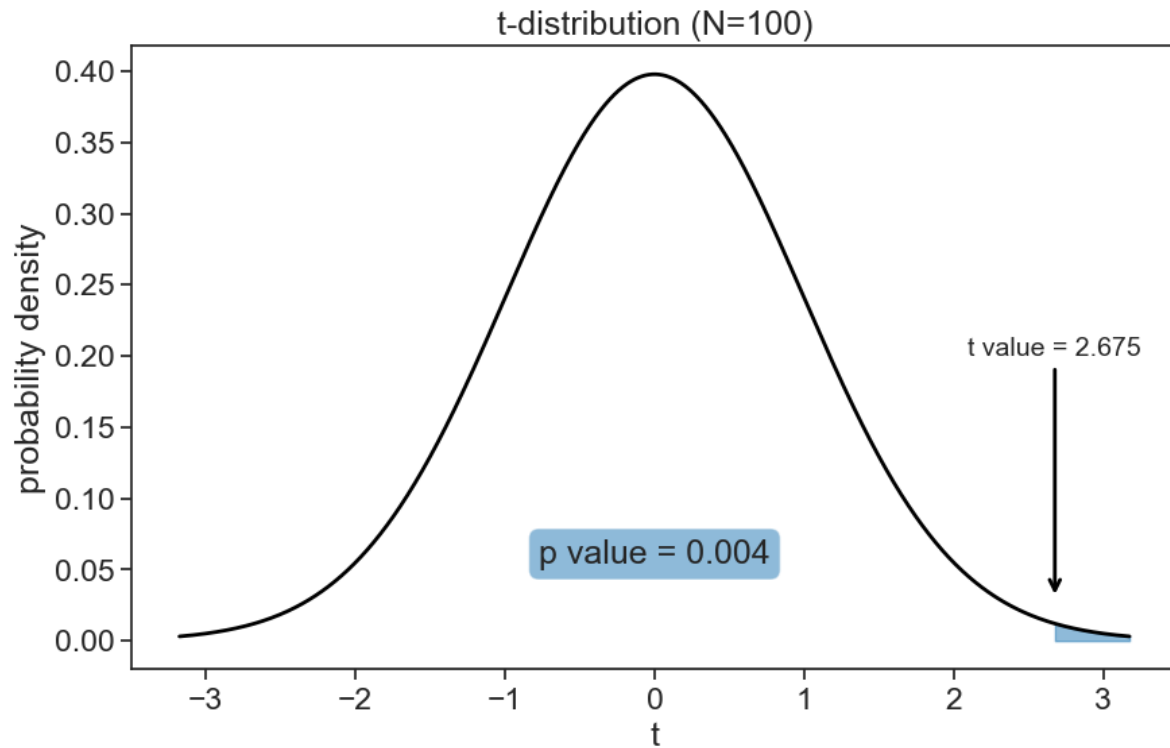
The answer is yes: the sampled men are significantly taller than the general population, since the p value is smaller than the significance level.

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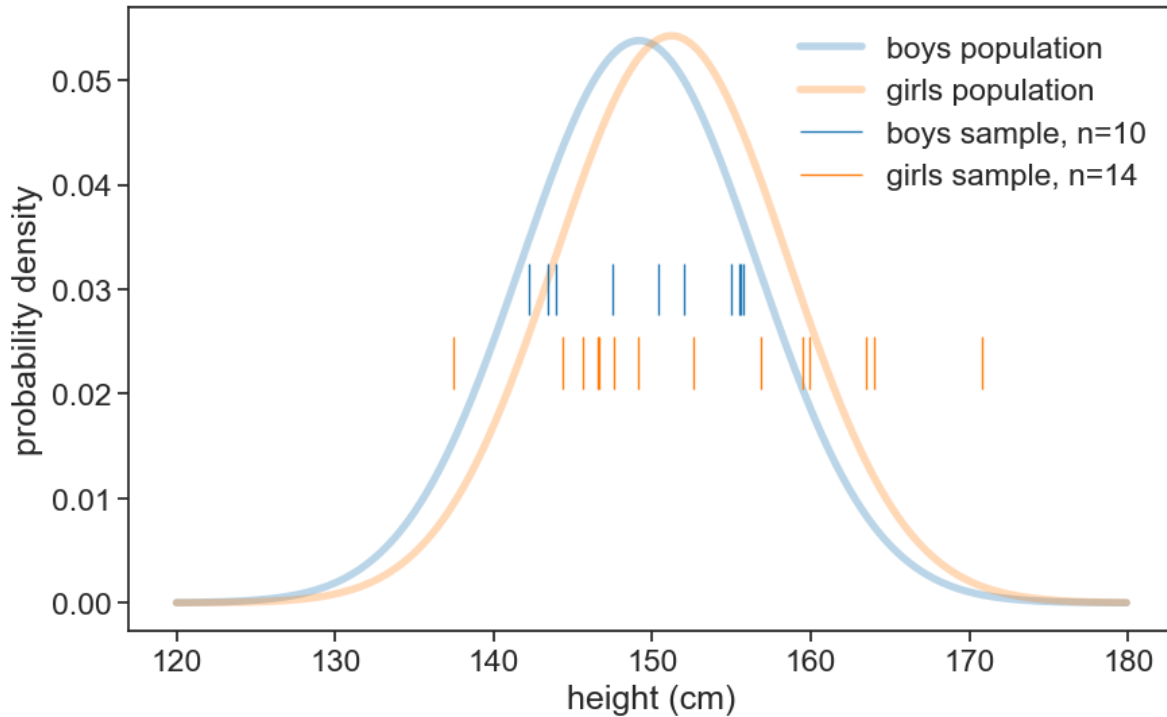
```python
N_boys = 10
N_girls = 14
# set scipy seed for reproducibility
np.random.seed(314)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
```

```python
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.
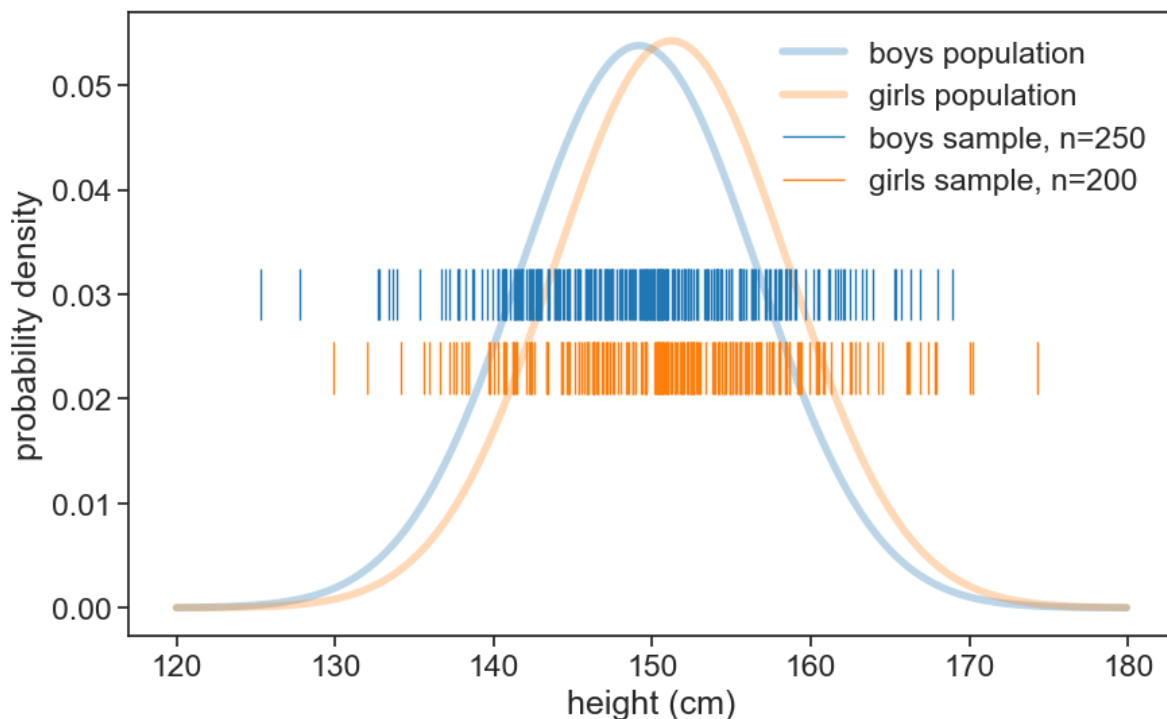
```
N_boys = 250
N_girls = 200
# set scipy seed for reproducibility
np.random.seed(314)
```

```
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',
    )
```

```python
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: -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.

```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
from matplotlib.lines import Line2D
import matplotlib.gridspec as gridspec
# %matplotlib widget
```

```python
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.

```python
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])
```

```python
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", coor
                       axesA=ax0, axesB=ax1, color="gray", linestyle='--', linewidth=1.5, alp
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=[],
        );
```

on average, the CI
of 1 out of 20 samples
($\alpha = 5\%$ significance level)
will not contain
the population mean
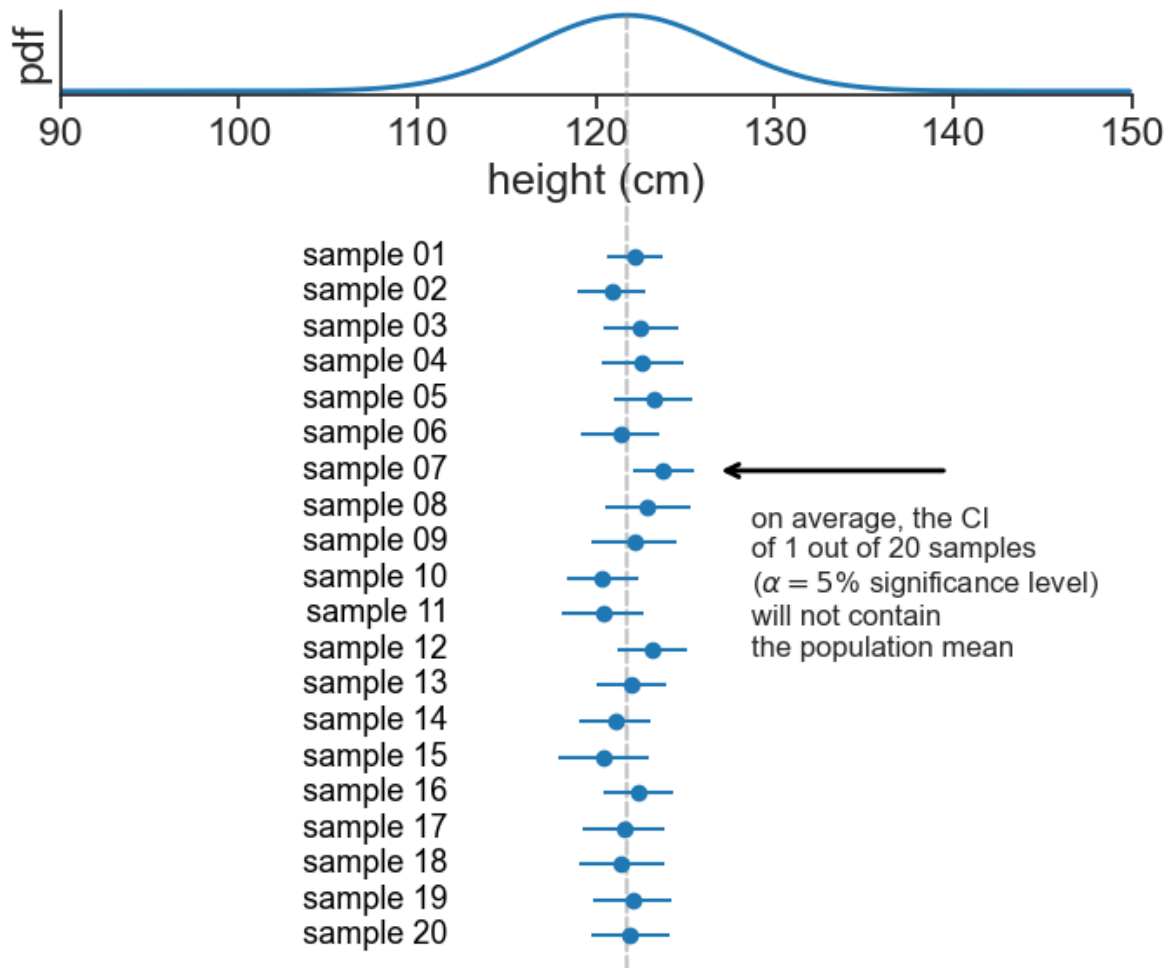
5 analytical confidence interval

```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.

```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
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', 
ax[1].hist(mean_list_30, bins=30, density=True, color=colors[1], label="N=10", align='mid', 
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");
```
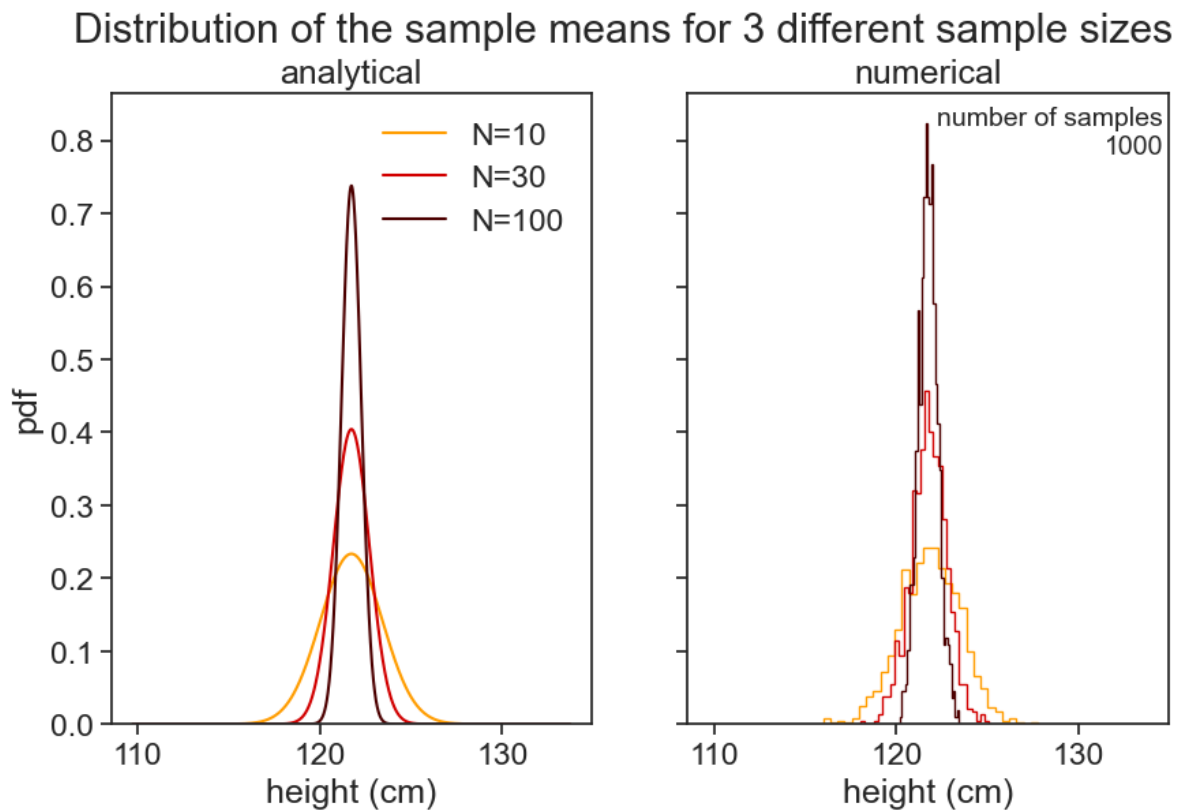

5.2 confidence interval 1

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.

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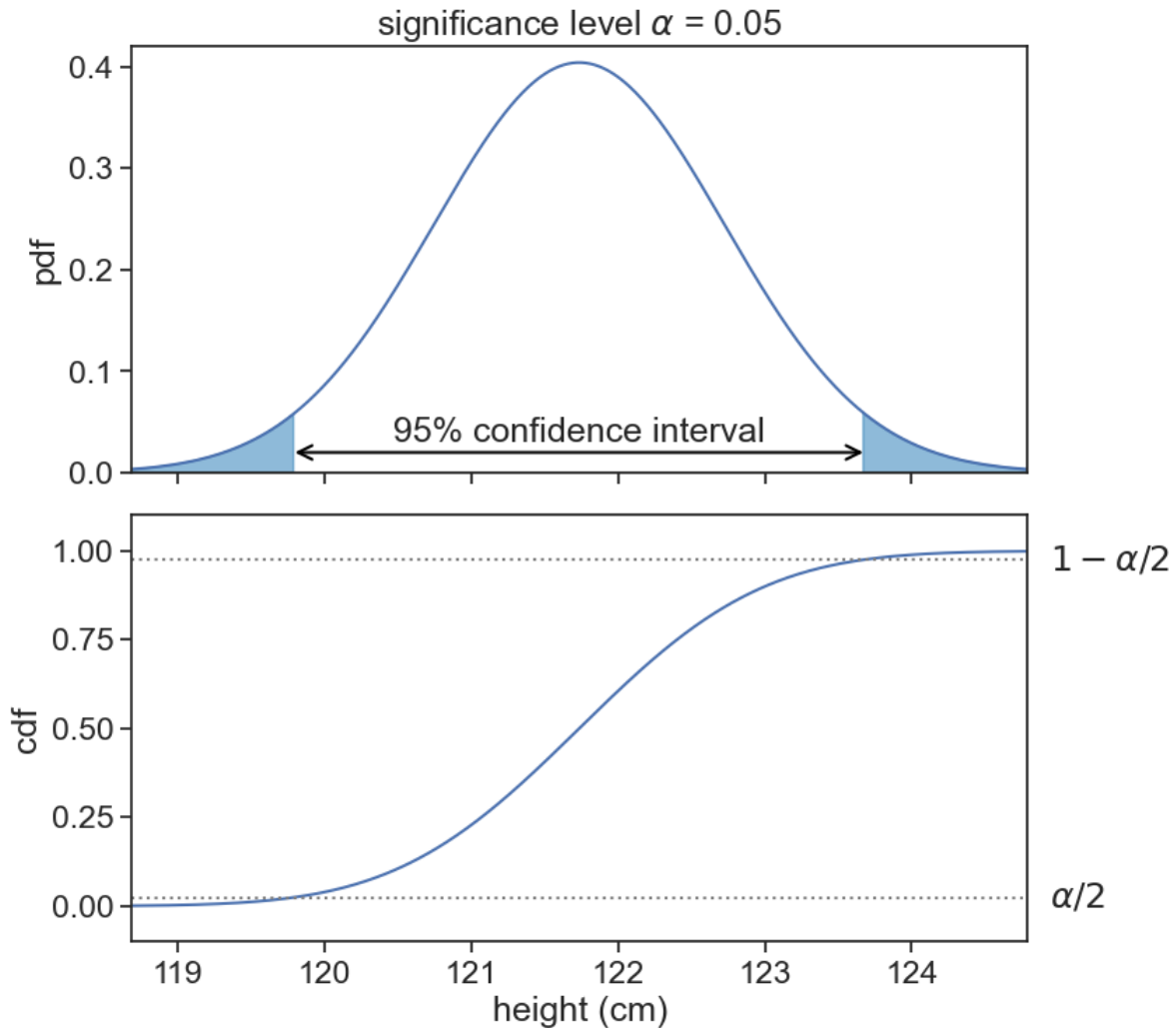
```
    );
```


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 = 100
B = 10000
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: 121.19 - 123.63 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")
```

```python
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)")
```

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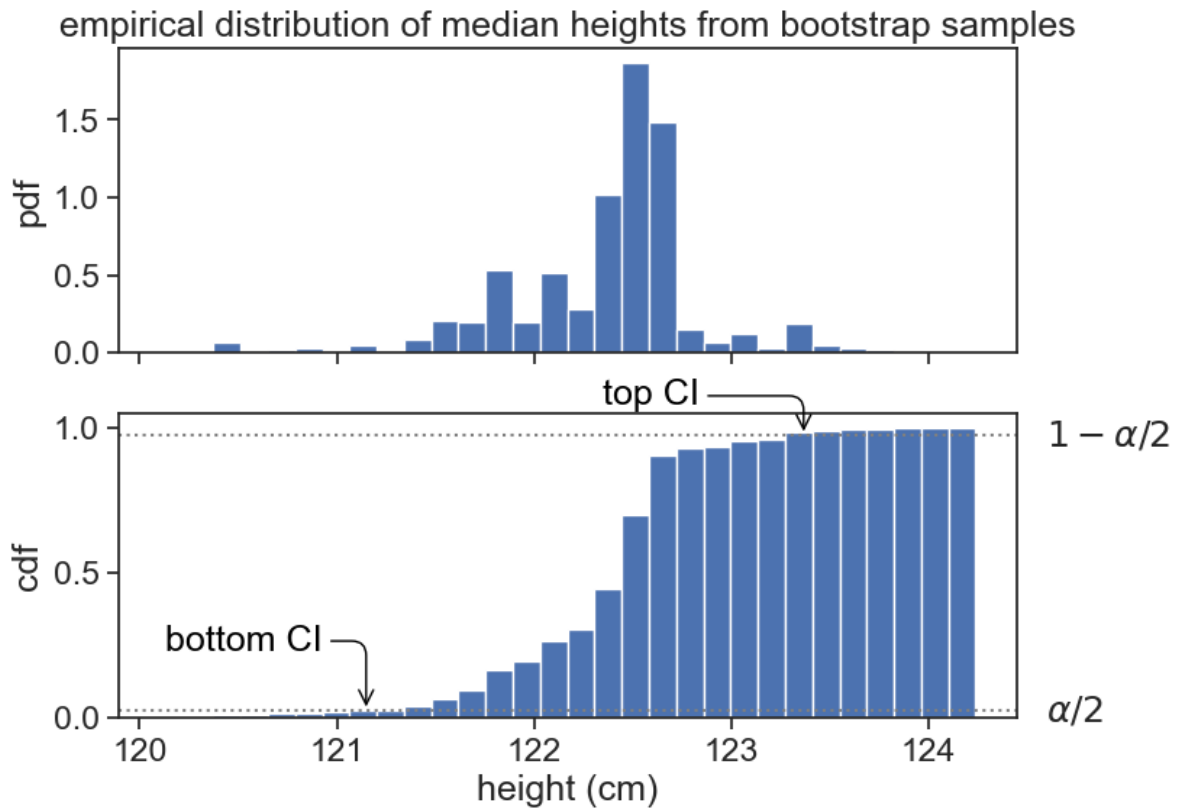
Clearly, the distribution of median height is not normal. The bootstrap method gives us a way to compute the confidence interval of the median height (or any other statistic of your choosing) without assuming normality.

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.

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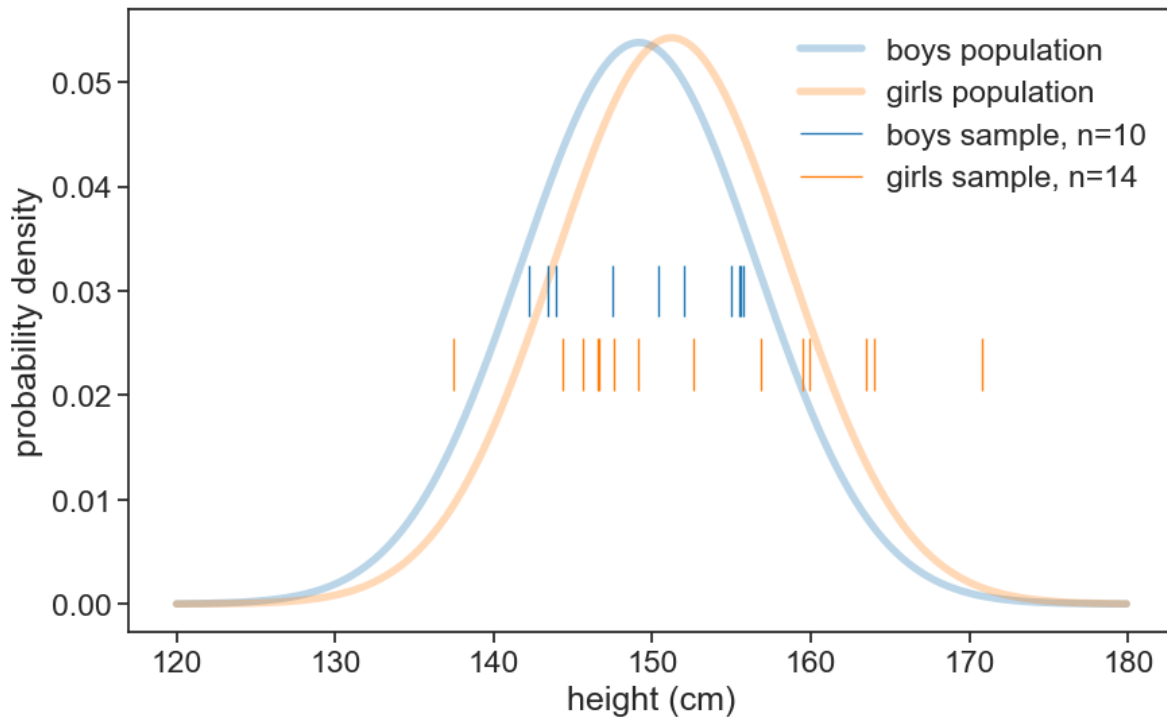
```python
N_boys = 10
N_girls = 14
# set scipy seed for reproducibility
np.random.seed(314)
sample_boys = norm.rvs(size=N_boys, loc=mu_boys, scale=sigma_boys)
sample_girls = norm.rvs(size=N_girls, loc=mu_girls, scale=sigma_girls)
```

```python
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
```

```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)
```

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

```python
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'),
```
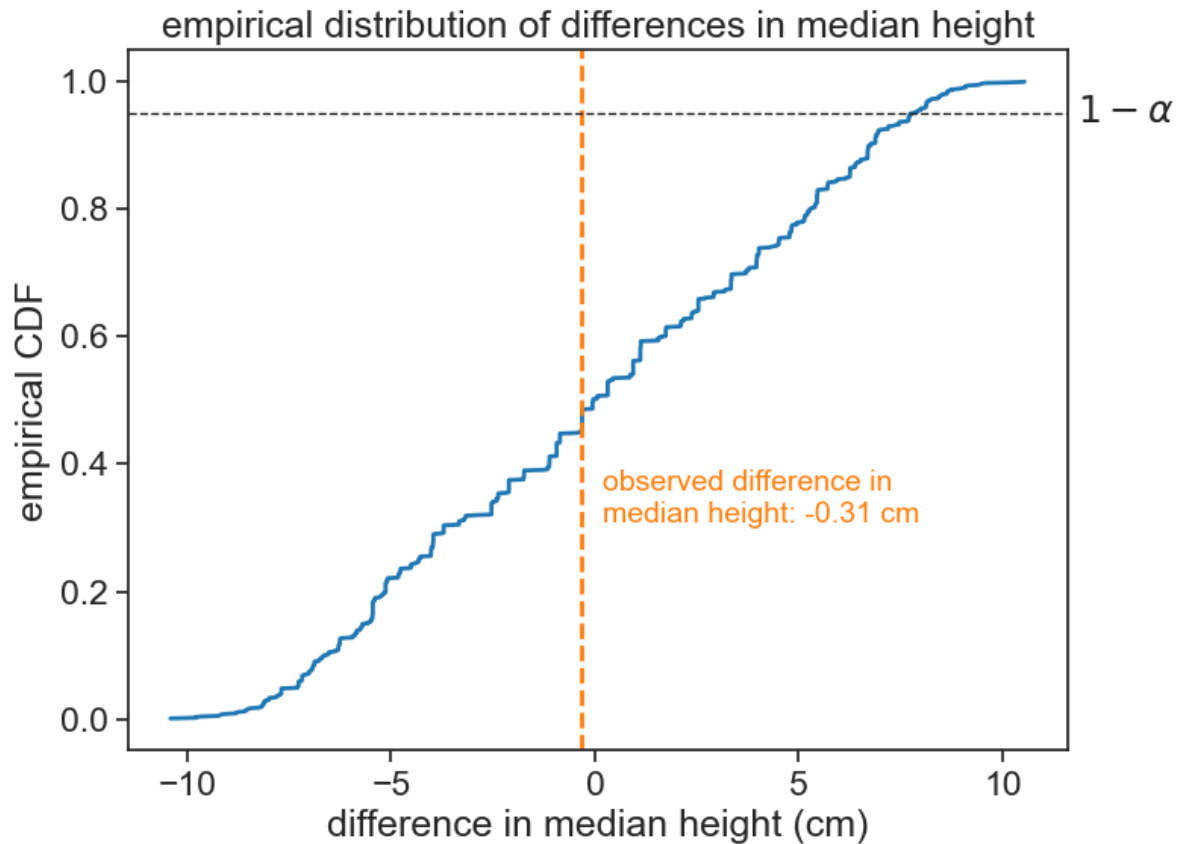
```
#                ha="left", va="center")
ax.set(xlabel='difference in median height (cm)',
       ylabel='empirical CDF',
       title=f'empirical distribution of differences in median height');
```


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
```

```python
# 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)
```

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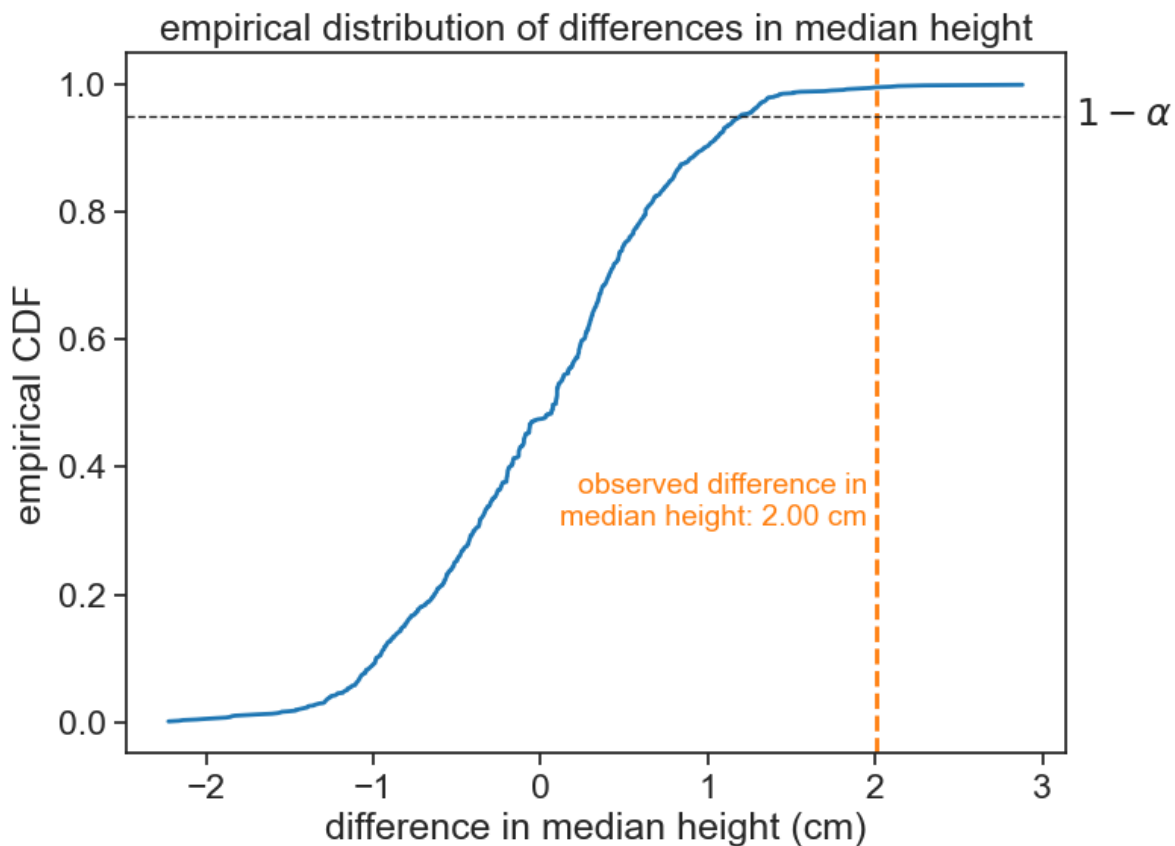
```
ax.axhline(1-alpha, color='k', lw=1, ls='--')
ax.annotate(r"$1-\alpha$", xy=(1.01, 1-alpha), xycoords=('axes fraction', 'data'),
            ha="left", va="center")
# for a two-tailed test
# ax.axhline(alpha/2, color='k', lw=1, ls='--')
# ax.axhline(1-alpha/2, color='k', lw=1, ls='--')
# ax.annotate(r"$\alpha/2$",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

```python
# 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

72

	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']


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


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


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

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


p_value = np.mean(diffs >= observed_diff)
# two-tailed p-value
# p_value = np.mean(np.abs(diffs) >= np.abs(observed_diff))
```

```
print("Monte Carlo permutation test 1")
print(f"observed difference: {observed_diff:.3f}")
print(f"p-value (one-tailed): {p_value:.4f}")
```

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

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

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

p_value = np.mean(diffs >= observed_diff)
# two-tailed p-value
# p_value = np.mean(np.abs(diffs) >= np.abs(observed_diff))
print("Monte Carlo permutation test 2")
print(f"observed difference: {observed_diff:.3f}")
print(f"p-value (one-tailed): {p_value:.4f}")
```

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

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

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

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))
```

```
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

svd

11 svd

```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
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation, FFMpegWriter
%load_ext manim

import os
%matplotlib widget
```

The manim module is not an IPython extension.

```python
fig, ax = plt.subplots(figsize=(8, 6))
x = np.array([1, 2, 3])
y = np.array([2, 2, 6])
intercept = np.ones_like(x)

# linear regression
slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(x, y)
# plot the regression line
ax.plot(x, intercept + slope * x, color='tab:orange', label='Regression Line')

ax.plot(x, y, marker='o', linestyle='None', color='tab:blue')
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');
```
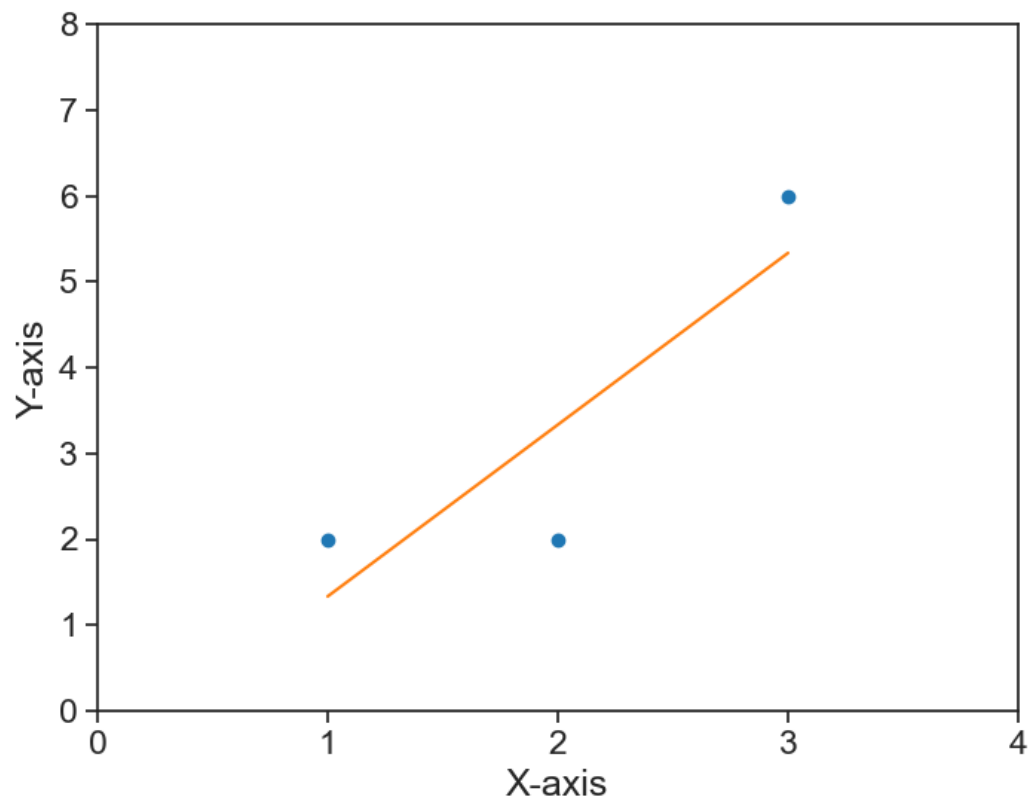
```
intercept, slope
```

```
(np.float64(-0.6666666666666665), np.float64(2.0))
```