**Course Description**



After completing Statistical Thinking in Python (Part 1), you have the probabilistic mindset and foundational hacker stats skills to dive into data sets and extract useful information from them. In this course, you will do just that, expanding and honing your hacker stats toolbox to perform the two key tasks in statistical inference, parameter estimation and hypothesis testing. You will work with real data sets as you learn, culminating with analysis of measurements of the beaks of the Darwin's famous finches. You will emerge from this course with new knowledge and lots of practice under your belt, ready to attack your own inference problems out in the world.

1

#### Parameter estimation by optimization

0%

When doing statistical inference, we speak the language of probability. A probability distribution that describes your data has parameters. So, a major goal of statistical inference is to estimate the values of these parameters, which allows us to concisely and unambiguously describe our data and draw conclusions from it. In this chapter, you will learn how to find the optimal parameters, those that best describe your data.

**Daily XP550**

# Optimal parameters

**50 XP**

## 1. Optimal parameters

After completing the prequel to this course, you are now beginning to think probabilistically. Outcomes of measurements follow probability distributions defined by the story of how the data came to be. When we looked at Michelson's speed of light in air measurements, we assumed that the results were Normally distributed.

## 2. Histogram of Michelson's measurements

We verified that both by looking at the PDF and

1. 1 Data: Michelson, 1880

## 3. CDF of Michelson's measurements

the CDF, which was more effective because there is no binning bias. To compute and plot the CDF, we needed our old friends

1. 1 Data: Michelson, 1880

## 4. Checking Normality of Michelson data

NumPy and matplotlib dot pyplot, so the first step was to import them with their traditional aliases. To compute the theoretical CDF by sampling, we passed two parameters into np dot random dot normal, the mean and standard deviation. The values we chose for these parameters were in fact the mean and standard deviation we calculated directly from the data.

## 5. CDF of Michelson's measurements

The result was that the theoretical CDF overlayed beautifully with the empirical CDF. How did we know that the mean and standard deviation calculated from the data were the appropriate values for the Normal parameters? We could have chosen others.

1. 1 Data: Michelson, 1880

## 6. CDF with bad estimate of st. dev.

What if the standard deviation differs by 50%? The CDFs no longer match. Or if the mean

1. 1 Data: Michelson, 1880

## 7. CDF with bad estimate of mean

varies by just point-01%. So, if we believe that the process that generates our data gives Normally distributed results,

1. 1 Data: Michelson, 1880

## 8. Optimal parameters

the set of parameters that brings the model, in this case a Normal distribution, in closest agreement with the data uses the mean and standard deviation computed directly from the data. These are the optimal parameters. Remember though, the parameters are only optimal for

## 9. Mass of MA large mouth bass

the model you chose for your data. When your model is wrong, the optimal parameters are not really meaningful. Finding the optimal parameters is not always as easy as just computing the mean and standard deviation from the data. We will encounter this later in this chapter when we do linear regressions and we rely on built-in NumPy functions to find the optimal parameters for us. I pause to note that

1. 1 Source: Mass. Dept. of Environmental Protection

## 10. Packages to do statistical inference

there are great tools in the Python ecosystem for doing statistical inference, including by optimization, scipy dot stats and

## 11. Packages to do statistical inference

statsmodels being two good examples. In this course, however,

## 12. Packages to do statistical inference

we focus on hacker statistics because the technique is like a Swiss Army knife; the same simple principle is applicable to a wide variety of statistical problems.

1. 1 Knife image: D-M Commons, CC BY-SA 3.0

## 13. Let's practice!

Now it's time for you to do some exercises to demonstrate how choosing optimal parameters results in best agreement between the theoretical model distribution and your data.

# How often do we get no-hitters?

The number of games played between each no-hitter in the modern era (1901-2015) of Major League Baseball is stored in the array nohitter\_times.

If you assume that no-hitters are described as a Poisson process, then the time between no-hitters is Exponentially distributed. As you have seen, the Exponential distribution has a single parameter, which we will call

, the typical interval time. The value of the parameter

that makes the exponential distribution best match the data is the mean interval time (where time is in units of number of games) between no-hitters.

Compute the value of this parameter from the data. Then, use np.random.exponential() to "repeat" the history of Major League Baseball by drawing inter-no-hitter times from an exponential distribution with the

you found and plot the histogram as an approximation to the PDF.

NumPy, pandas, matplotlib.pyplot, and seaborn have been imported for you as np, pd, plt, and sns, respectively.

##### Instructions

**100 XP**

* Seed the random number generator with 42.
* Compute the mean time (in units of number of games) between no-hitters.
* Draw 100,000 samples from an Exponential distribution with the parameter you computed from the mean of the inter-no-hitter times.
* Plot the theoretical PDF using plt.hist(). Remember to use keyword arguments bins=50, normed=True, and histtype='step'. Be sure to label your axes.
* Show your plot.
* # Seed random number generator
* \_\_\_\_
* # Compute mean no-hitter time: tau
* tau = \_\_\_\_
* # Draw out of an exponential distribution with parameter tau: inter\_nohitter\_time
* inter\_nohitter\_time = \_\_\_\_(\_\_\_\_, 100000)
* # Plot the PDF and label axes
* \_ = \_\_\_\_(inter\_nohitter\_time,
* \_\_\_\_, \_\_\_\_, \_\_\_\_)
* \_ = plt.xlabel('Games between no-hitters')
* \_ = plt.ylabel('PDF')
* # Show the plot
* plt.show()
* # Seed random number generator
* np.random.seed(42)
* # Compute mean no-hitter time: tau
* tau = np.mean(nohitter\_times)
* # Draw out of an exponential distribution with parameter tau: inter\_nohitter\_time
* inter\_nohitter\_time = np.random.exponential(tau, size=100000)
* # Plot the PDF and label axes
* \_ = plt.hist(inter\_nohitter\_time,
* bins=50, normed=True, histtype='step')
* \_ = plt.xlabel('Games between no-hitters')
* \_ = plt.ylabel('PDF')
* # Show the plot
* plt.show()

# Seed random number generator np.random.seed(42) # Compute mean no-hitter time: tau tau = np.mean(nohitter\_times) # Draw out of an exponential distribution with parameter tau: inter\_nohitter\_time inter\_nohitter\_time = np.random.exponential(tau, size=100000) # Plot the PDF and label axes \_ = plt.hist(inter\_nohitter\_time, bins=50, normed=True, histtype='step') \_ = plt.xlabel('Games between no-hitters') \_ = plt.ylabel('PDF') # Show the plot plt.show()

Nice work! We see the typical shape of the Exponential distribution, going from a maximum at 0 and decaying to the right.

# Do the data follow our story?

You have modeled no-hitters using an Exponential distribution. Create an ECDF of the real data. Overlay the theoretical CDF with the ECDF from the data. This helps you to verify that the Exponential distribution describes the observed data.

It may be helpful to remind yourself of the [function you created in the previous course](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-1/graphical-exploratory-data-analysis?ex=12) to compute the ECDF, as well as the code you wrote to [plot it](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-1/graphical-exploratory-data-analysis?ex=13).

##### Instructions

* Compute an ECDF from the actual time between no-hitters (nohitter\_times). Use the ecdf() function you wrote in the prequel course.
* Create a CDF from the theoretical samples you took in the last exercise (inter\_nohitter\_time).
* Plot x\_theor and y\_theor as a line using plt.plot(). Then overlay the ECDF of the real data x and y as points. To do this, you have to specify the keyword arguments marker = '.' and linestyle = 'none' in addition to x and y inside plt.plot().
* Set a 2% margin on the plot.
* Show the plot.
* # Create an ECDF from real data: x, y
* x, y = \_\_\_\_
* # Create a CDF from theoretical samples: x\_theor, y\_theor
* x\_theor, y\_theor = \_\_\_\_
* # Overlay the plots
* plt.plot(\_\_\_\_, \_\_\_\_)
* plt.plot(\_\_\_\_, \_\_\_\_, marker=\_\_\_\_, linestyle=\_\_\_\_)
* # Margins and axis labels
* plt.margins(\_\_\_\_)
* plt.xlabel('Games between no-hitters')
* plt.ylabel('CDF')
* # Show the plot
* plt.show()

# Create an ECDF from real data: x, y

x, y = ecdf(nohitter\_times)

# Create a CDF from theoretical samples: x\_theor, y\_theor

x\_theor, y\_theor = ecdf(inter\_nohitter\_time)

# Overlay the plots

plt.plot(x\_theor, y\_theor)

plt.plot(x, y, marker='.', linestyle='none')

# Margins and axis labels

plt.margins(0.02)

plt.xlabel('Games between no-hitters')

plt.ylabel('CDF')

# Show the plot

plt.show()

# Create an ECDF from real data: x, y x, y = ecdf(nohitter\_times) # Create a CDF from theoretical samples: x\_theor, y\_theor x\_theor, y\_theor = ecdf(inter\_nohitter\_time) # Overlay the plots plt.plot(x\_theor, y\_theor) plt.plot(x, y, marker='.', linestyle='none') # Margins and axis labels plt.margins(0.02) plt.xlabel('Games between no-hitters') plt.ylabel('CDF') # Show the plot plt.show()

It looks like no-hitters in the modern era of Major League Baseball are Exponentially distributed. Based on the story of the Exponential distribution, this suggests that they are a random process; when a no-hitter will happen is independent of when the last no-hitter was.

# How is this parameter optimal?

Now sample out of an exponential distribution with

being twice as large as the optimal . Do it again for half as large. Make CDFs of these samples and overlay them with your data. You can see that they do not reproduce the data as well. Thus, the

you computed from the mean inter-no-hitter times is optimal in that it best reproduces the data.

Note: In this and all subsequent exercises, the random number generator is pre-seeded for you to save you some typing.

##### Instructions

##### Take 10000 samples out of an Exponential distribution with parameter

 = tau/2.

 Take 10000 samples out of an Exponential distribution with parameter

 = 2\*tau.

 Generate CDFs from these two sets of samples using your ecdf() function.

 Add these two CDFs as lines to your plot. This has been done for you, so hit submit to view the plot!

# Plot the theoretical CDFs

plt.plot(x\_theor, y\_theor)

plt.plot(x, y, marker='.', linestyle='none')

plt.margins(0.02)

plt.xlabel('Games between no-hitters')

plt.ylabel('CDF')

# Take samples with half tau: samples\_half

samples\_half = np.random.exponential(tau/2, size=10000)

# Take samples with double tau: samples\_double

samples\_double = np.random.exponential(2\*tau, size=10000)

# Generate CDFs from these samples

x\_half, y\_half = ecdf(samples\_half)

x\_double, y\_double = ecdf(samples\_double)

# Plot these CDFs as lines

\_ = plt.plot(x\_half, y\_half)

\_ = plt.plot(x\_double, y\_double)

# Show the plot

plt.show()

# Plot the theoretical CDFs plt.plot(x\_theor, y\_theor) plt.plot(x, y, marker='.', linestyle='none') plt.margins(0.02) plt.xlabel('Games between no-hitters') plt.ylabel('CDF') # Take samples with half tau: samples\_half samples\_half = np.random.exponential(tau/2, size=10000) # Take samples with double tau: samples\_double samples\_double = np.random.exponential(2\*tau, size=10000) # Generate CDFs from these samples x\_half, y\_half = ecdf(samples\_half) x\_double, y\_double = ecdf(samples\_double) # Plot these CDFs as lines \_ = plt.plot(x\_half, y\_half) \_ = plt.plot(x\_double, y\_double) # Show the plot plt.show()

Great work! Notice how the value of tau given by the mean matches the data best. In this way, tau is an optimal parameter.

# Linear regression by least squares

**50 XP**

## 1. Linear regression by least squares

Sometimes two variables are related. You may recall from the prequel to this course that we computed the Pearson correlation coefficient between

## 2. 2008 US swing state election results

Obama's vote share in each county in swings states and the total vote count of the respective counties. The Pearson correlation coefficient is important to compute, but we might like to get a fuller understanding of how the data are related to each other. Specifically, we might suspect some underlying function gives the data its shape.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. 2008 US swing state election results

Often times a linear function is appropriate to describe the data, and this is what we will focus on in this course. The parameters of the function are

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 4. 2008 US swing state election results

the slope and intercept. The slope sets how steep the line is, and the intercept sets where the line crosses the y-axis. How do we figure out which slope and intercept best describe the data? A simple answer is that we want to choose the slope and intercept such that the data points collectively lie as close as possible to the line. This is easiest to think about by first considering one data point,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 5. 2008 US swing state election results

say this one. The vertical distance between the data point and the line is called

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 6. Residuals

the residual. In this case, the residual has a negative value because the data point lies below the line. Each data point has a residual associated with it.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 7. Least squares

We define the line that is closest to the data to be the line for which the sum of the squares of all of the residuals is minimal. This process, finding the parameters for which the sum of the squares of the residuals is minimal, is called "least squares". There are many algorithms to do this in practice.

## 8. Least squares with np.polyfit()

We will use the NumPy function polyfit, which performs least squares analysis with polynomial functions. We can use it because a linear function is a first degree polynomial. The first two arguments to this function are the x and y data. The third argument is the degree of the polynomial you wish to fit; for linear functions, we enter one. The function returns the slope and intercept of the best fit line. The slope tells us that we get about 4 more percent votes for Obama for every 100,000 additional voters in a county. Now that you know how to perform a linear regression,

## 9. Let's practice!

let's do it with some real data in the exercises!

# EDA of literacy/fertility data

In the next few exercises, we will look at the correlation between female literacy and fertility (defined as the average number of children born per woman) throughout the world. For ease of analysis and interpretation, we will work with the illiteracy rate.

It is always a good idea to do some EDA ahead of our analysis. To this end, plot the fertility versus illiteracy and compute the Pearson correlation coefficient. The NumPy array illiteracy has the illiteracy rate among females for most of the world's nations. The array fertility has the corresponding fertility data.

Here, it may be useful to refer back to the [function you wrote in the previous course](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-1/quantitative-exploratory-data-analysis?ex=15) to compute the Pearson correlation coefficient.

##### Instructions

* Plot fertility (y-axis) versus illiteracy (x-axis) as a scatter plot.
* Set a 2% margin.
* Compute and print the Pearson correlation coefficient between illiteracy and fertility.
* # Plot the illiteracy rate versus fertility
* \_ = plt.plot(\_\_\_\_, \_\_\_\_, \_\_\_\_='.', \_\_\_\_='none')
* # Set the margins and label axes
* plt.margins(\_\_\_\_)
* \_ = plt.xlabel('percent illiterate')
* \_ = plt.ylabel('fertility')
* # Show the plot
* plt.show()
* # Show the Pearson correlation coefficient
* print(pearson\_r(\_\_\_\_, \_\_\_\_))
* # Plot the illiteracy rate versus fertility
* \_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')
* # Set the margins and label axes
* plt.margins(0.02)
* \_ = plt.xlabel('percent illiterate')
* \_ = plt.ylabel('fertility')
* # Show the plot
* plt.show()
* # Show the Pearson correlation coefficient
* print(pearson\_r(illiteracy, fertility))

# Plot the illiteracy rate versus fertility

\_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')

# Set the margins and label axes

plt.margins(0.02)

\_ = plt.xlabel('percent illiterate')

\_ = plt.ylabel('fertility')

# Show the plot

plt.show()

# Show the Pearson correlation coefficient

print(pearson\_r(illiteracy, fertility))

0.8041324026815346

You can see the correlation between illiteracy and fertility by eye, and by the substantial Pearson correlation coefficient of 0.8. It is difficult to resolve in the scatter plot, but there are many points around near-zero illiteracy and about 1.8 children/woman.

# Linear regression

We will assume that fertility is a linear function of the female illiteracy rate. That is,

, where is the slope and

is the intercept. We can think of the intercept as the minimal fertility rate, probably somewhere between one and two. The slope tells us how the fertility rate varies with illiteracy. We can find the best fit line using np.polyfit().

Plot the data and the best fit line. Print out the slope and intercept. (Think: what are their units?)

##### Instructions

* Compute the slope and intercept of the regression line using np.polyfit(). Remember, fertility is on the y-axis and illiteracy on the x-axis.
* Print out the slope and intercept from the linear regression.
* To plot the best fit line, create an array x that consists of 0 and 100 using np.array(). Then, compute the theoretical values of y based on your regression parameters. I.e., y = a \* x + b.
* Plot the data and the regression line on the same plot. Be sure to label your axes.
* Hit submit to display your plot.

**# Plot the illiteracy rate versus fertility**

**\_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')**

**plt.margins(0.02)**

**\_ = plt.xlabel('percent illiterate')**

**\_ = plt.ylabel('fertility')**

**# Perform a linear regression using np.polyfit(): a, b**

**a, b = \_\_\_\_**

**# Print the results to the screen**

**print('slope =', a, 'children per woman / percent illiterate')**

**print('intercept =', b, 'children per woman')**

**# Make theoretical line to plot**

**x = \_\_\_\_**

**y = \_\_\_\_ \* \_\_\_\_ + \_\_\_\_**

**# Add regression line to your plot**

**\_ = plt.plot(\_\_\_\_, \_\_\_\_)**

**# Draw the plot**

**plt.show()**

**# Plot the illiteracy rate versus fertility**

**\_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')**

**plt.margins(0.02)**

**\_ = plt.xlabel('percent illiterate')**

**\_ = plt.ylabel('fertility')**

**# Perform a linear regression using np.polyfit(): a, b**

**a, b = np.polyfit(illiteracy,fertility,1)**

**# Print the results to the screen**

**print('slope =', a, 'children per woman / percent illiterate')**

**print('intercept =', b, 'children per woman')**

**# Make theoretical line to plot**

**x = np.array([0,100])**

**y = a \* x + b**

**# Add regression line to your plot**

**\_ = plt.plot(x, y)**

**# Draw the plot**

**Pl. Show()**

# Plot the illiteracy rate versus fertility

\_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')

plt.margins(0.02)

\_ = plt.xlabel('percent illiterate')

\_ = plt.ylabel('fertility')

# Perform a linear regression using np.polyfit(): a, b

a, b = np.polyfit(illiteracy,fertility,1)

# Print the results to the screen

print('slope =', a, 'children per woman / percent illiterate')

print('intercept =', b, 'children per woman')

# Make theoretical line to plot

x = np.array([0,100])

y = a \* x + b

# Add regression line to your plot

\_ = plt.plot(x, y)

# Draw the plot

plt.show()

slope = 0.04979854809063421 children per woman / percent illiterate

intercept = 1.8880506106365573 children per woman

# How is it optimal?

The function np.polyfit() that you used to get your regression parameters finds the optimal slope and intercept. It is optimizing the sum of the squares of the residuals, also known as RSS (for residual sum of squares). In this exercise, you will plot the function that is being optimized, the RSS, versus the slope parameter a. To do this, fix the intercept to be what you found in the optimization. Then, plot the RSS versus the slope. Where is it minimal?

##### Instructions

* Specify the values of the slope to compute the RSS. Use np.linspace() to get 200 points in the range between 0 and 0.1. For example, to get 100 points in the range between 0 and 0.5, you could use np.linspace() like so: np.linspace(0, 0.5, 100).
* Initialize an array, rss, to contain the RSS using np.empty\_like() and the array you created above. The empty\_like() function returns a new array with the same shape and type as a given array (in this case, a\_vals).
* Write a for loop to compute the sum of RSS of the slope. Hint: the RSS is given by np.sum((y\_data - a \* x\_data - b)\*\*2). The variable b you computed in the last exercise is already in your namespace. Here, fertility is the y\_data and illiteracy the x\_data.
* Plot the RSS (rss) versus slope (a\_vals).
* Hit submit to see the plot!
* # Specify slopes to consider: a\_vals
* a\_vals = np.linspace(0,0.1,200)
* # Initialize sum of square of residuals: rss
* rss = np.empty\_like(a\_vals)
* # Compute sum of square of residuals for each value of a\_vals
* for i, a in enumerate(a\_vals):
* rss[i] = np.sum((fertility - a\*illiteracy - b)\*\*2)
* # Plot the RSS
* plt.plot(a\_vals, rss,'-')
* plt.xlabel('slope (children per woman / percent illiterate)')
* plt.ylabel('sum of square of residuals')
* plt.show()

# Specify slopes to consider: a\_vals a\_vals = np.linspace(0,0.1,200) # Initialize sum of square of residuals: rss rss = np.empty\_like(a\_vals) # Compute sum of square of residuals for each value of a\_vals for i, a in enumerate(a\_vals): rss[i] = np.sum((fertility - a\*illiteracy - b)\*\*2) # Plot the RSS plt.plot(a\_vals, rss,'-') plt.xlabel('slope (children per woman / percent illiterate)') plt.ylabel('sum of square of residuals') plt.show()

Great work! Notice that the minimum on the plot, that is the value of the slope that gives the minimum sum of the square of the residuals, is the same value you got when performing the regression.

**Daily XP1250**

# The importance of EDA: Anscombe's quartet

**50 XP**

## 1. The importance of EDA: Anscombe's quartet

In 1973, statistician Francis Anscombe published a paper that contained

## 2. Anscombe's quartet

four fictitious x-y data sets, plotted here. He uses these data sets to make an important point. That point becomes clear if we blindly go about doing parameter estimation on these data sets. First, let's look at the average x-values of the four data sets.

1. 1 Data: Anscombe, The American Statistician, 1973

## 3. Anscombe's quartet

They are all the same. How about the average y-values?

1. 1 Data: Anscombe, The American Statistician, 1973

## 4. Anscombe's quartet

Again, all the same. And what if we do a linear regression on each of the data sets?

1. 1 Data: Anscombe, The American Statistician, 1973

## 5. Anscombe's quartet

They all have the same line! Surely some of the fits are less optimal than others. Let's look at the sum of the squares of the residuals.

1. 1 Data: Anscombe, The American Statistician, 1973

## 6. Anscombe's quartet

Oh my, they are all basically the same as well. Of course, Anscombe constructed the data sets so that this would happen. The point he was making is very important. You already have some powerful tools for statistical inference. You can compute summary statistics and optimal parameters, including linear regression parameters, and by the end of the course, you will able to construct confidence intervals with quantify uncertainty about the parameter estimates. These are crucial skills for any data analysis, no doubt.

1. 1 Data: Anscombe, The American Statistician, 1973

## 7. Look before you leap!

But look before you leap! This is a powerful reminder to do some graphic exploratory data analysis before you start computing and making judgments about your data. For example,

## 8. Anscombe's quartet

this data set might be well modeled with a line, and the regression parameters will be meaningful. The same is true of

1. 1 Data: Anscombe, The American Statistician, 1973

## 9. Anscombe's quartet

this data set, but the outlier throws off the slope and intercept. After doing EDA, you should look into what is causing that outlier.

1. 1 Data: Anscombe, The American Statistician, 1973

## 10. Anscombe's quartet

This data set might also have a linear relationship between x and y, but from the plot, you can conclude that you should try to acquire more data for intermediate x values to make sure that it does.

1. 1 Data: Anscombe, The American Statistician, 1973

## 11. Anscombe's quartet

And this data set is definitely not linear, and you need to choose another model. Explore your data first. I'll let you prove to yourself

1. 1 Data: Anscombe, The American Statistician, 1973

## 12. Let's practice!

that these data sets give the same regression parameters. It will be good practice, and seeing is believing!

# The importance of EDA

Why should exploratory data analysis be the first step in an analysis of data (after getting your data imported and cleaned, of course)?

##### Answer the questions

* **You can be protected from misinterpretation of the type demonstrated by Anscombe's quartet**
* **EDA provides a good starting point for planning the rest of your analysis.**
* **EDA is not really any more difficult than any of the subsequent analysis, so there is no excuse for not exploring the data.**
* **All of these reasons!**

**Yes! Always do EDA as you jump into a data set.**

# Linear regression on appropriate Anscombe data

For practice, perform a linear regression on the data set from Anscombe's quartet that is most reasonably interpreted with linear regression.

##### Instructions

* Compute the parameters for the slope and intercept using np.polyfit(). The Anscombe data are stored in the arrays x and y.
* Print the slope a and intercept b.
* Generate theoretical

and data from the linear regression. Your array, which you can create with np.array(), should consist of 3 and 15. To generate the

* data, multiply the slope by x\_theor and add the intercept.
* Plot the Anscombe data as a scatter plot and then plot the theoretical line. Remember to include the marker='.' and linestyle='none' keyword arguments in addition to x and y when to plot the Anscombe data as a scatter plot. You do not need these arguments when plotting the theoretical line.
* Hit submit to see the plot!
* # Perform linear regression: a, b
* a, b = \_\_\_\_
* # Print the slope and intercept
* print(\_\_\_\_, \_\_\_\_)
* # Generate theoretical x and y data: x\_theor, y\_theor
* x\_theor = np.array([\_\_\_\_, \_\_\_\_])
* y\_theor = \_\_\_\_ \* \_\_\_\_ + \_\_\_\_
* # Plot the Anscombe data and theoretical line
* \_ = \_\_\_\_
* \_ = \_\_\_\_
* # Label the axes
* plt.xlabel('x')
* plt.ylabel('y')
* # Show the plot
* plt.show()

# Perform linear regression: a, b

a, b = np.polyfit(x,y,1)

# Print the slope and intercept

print(a, b)

# Generate theoretical x and y data: x\_theor, y\_theor

x\_theor = np.array([3, 15])

y\_theor = a \* x\_theor + b

# Plot the Anscombe data and theoretical line

\_ = plt.plot(x, y, marker=".", linestyle='none')

\_ = plt.plot(x\_theor,y\_theor, linestyle='none', marker='.')

# Label the axes

plt.xlabel('x')

plt.ylabel('y')

# Show the plot

plt.show()

# Perform linear regression: a, b

a, b = np.polyfit(x,y,1)

# Print the slope and intercept

print(a, b)

# Generate theoretical x and y data: x\_theor, y\_theor

x\_theor = np.array([3, 15])

y\_theor = a \* x\_theor + b

# Plot the Anscombe data and theoretical line

\_ = plt.plot(x, y, marker=".", linestyle='none')

\_ = plt.plot(x\_theor,y\_theor, linestyle='none', marker='.')

# Label the axes

plt.xlabel('x')

plt.ylabel('y')

# Show the plot

plt.show()

0.5000909090909096 3.000090909090909

Great work! You're getting to be a linear regression pro!

# Linear regression on all Anscombe data

Now, to verify that all four of the Anscombe data sets have the same slope and intercept from a linear regression, you will compute the slope and intercept for each set. The data are stored in lists; anscombe\_x = [x1, x2, x3, x4] and anscombe\_y = [y1, y2, y3, y4], where, for example, x2 and y2 are the

and

values for the second Anscombe data set.

##### Instructions

* Write a for loop to do the following for each Anscombe data set.
  + Compute the slope and intercept.
  + Print the slope and intercept.
* # Iterate through x,y pairs
* for x, y in zip(\_\_\_\_, \_\_\_\_):
* # Compute the slope and intercept: a, b
* a, b = \_\_\_\_
* # Print the result
* print('slope:', a, 'intercept:', b)

# Iterate through x,y pairs

for x, y in zip(anscombe\_x, anscombe\_y):

    # Compute the slope and intercept: a, b

    a, b = np.polyfit(x,y,1)

    # Print the result

    print('slope:', a, 'intercept:', b)

# Iterate through x,y pairs

for x, y in zip(anscombe\_x, anscombe\_y):

# Compute the slope and intercept: a, b

a, b = np.polyfit(x,y,1)

# Print the result

print('slope:', a, 'intercept:', b)

slope: 0.5000909090909096 intercept: 3.000090909090909

slope: 0.5000000000000003 intercept: 3.0009090909090896

slope: 0.4997272727272729 intercept: 3.0024545454545457

slope: 0.49990909090909086 intercept: 3.0017272727272735

<script.py> output:

slope: 0.5000909090909096 intercept: 3.000090909090909

slope: 0.5000000000000003 intercept: 3.0009090909090896

slope: 0.4997272727272729 intercept: 3.0024545454545457

slope: 0.49990909090909086 intercept: 3.0017272727272735

Great work! Indeed, they all have the same slope and intercept.

**Daily XP1550**

# Generating bootstrap replicates

**50 XP**

## 1. Generating bootstrap replicates

In the prequel to this course we computed summary statistics of measurements, including the mean, median, and standard deviation. But remember, we need to think probabilistically. What if we acquired the data again? Would we get the same mean? The same median? The same standard deviation? Probably not. In inference problems, it is rare that we are interested in the result from a single experiment or data acquisition. We want to say something more general.

## 2. Michelson's speed of light measurements

Michelson was not interested in what the measured speed of light was in the specific 100 measurements conducted in the summer of 1879. He wanted to know what the speed of light actually is. Statistically speaking, that means he wanted to know what speed of light he would observe if he did the experiment over and over again an infinite number of times. Unfortunately, actually repeating the experiment lots and lots of times is just not possible. But, as hackers, we can simulate getting the data again.

1. 1 Data: Michelson, 1880

## 3. Resampling an array

The idea is that we resample the data we have and recompute the summary statistic of interest, say the mean. To resample an array of measurements, we randomly

## 4. Resampling an array

select one entry and

## 5. Resampling an array

store it. Importantly, we

## 6. Resampling an array

replace the entry in the original array, or equivalently, we just don't delete it. This is called sampling with replacement. Then, we then randomly

## 7. Resampling an array

select another

## 8. Resampling an array

one and store it. We do this n times,

## 9. Resampling an array

where n is the total number of measurements, five in this case. We then have a resampled array of data. Using this new resampled array, we compute the summary statistic and store the result. Resampling the speed of light data is as if we repeated Michelson's set of measurements.

## 10. Mean of resampled Michelson measurements

We do this over and over again to get a large number of summary statistics from resampled data sets. We can use these results to plot an ECDF, for example, to get a picture of the probability distribution describing the summary statistic. This process is an example of

## 11. Bootstrapping

bootstrapping, which more generally is the use of resampled data to perform statistical inference. To make sure we have our terminology down, each resampled array is called

## 12. Bootstrap sample

a bootstrap sample. A bootstrap replicate

## 13. Bootstrap replicate

is the value of the summary statistic computed from the bootstrap sample. The name makes sense; it's a simulated replica of the original data acquired by bootstrapping. Let's look at how we can generate a bootstrap sample and compute a bootstrap replicate from it using Python. We will use Michelson's measurements of the speed of light.

## 14. Resampling engine: np.random.choice()

First, we need a function to perform the resampling. The NumPy function random dot choice provides this functionality. Conveniently, like many of the other functions in the NumPy random module, it has a size keyword argument, which allows us to specify how many samples we want to take out of the array. Notice that it chose the number five three times; the function does not delete an entry when it samples it out of the array. Now, we can draw 100 samples out of the Michelson speed of light data.

## 15. Computing a bootstrap replicate

This is a bootstrap sample, since there were 100 data points and we are choosing 100 of them with replacement. Now that we have a bootstrap sample, we can compute a bootstrap replicate. We can pick whatever summary statistic we like. We'll compute the mean, median, and standard deviation. It's as simple as treating the bootstrap sample as though it were a data set.

## 16. Let's practice!

Now it's time for you to do some bootstrap sampling yourself!

# Getting the terminology down

Getting tripped up over terminology is a common cause of frustration in students. Unfortunately, you often will read and hear other data scientists using different terminology for bootstrap samples and replicates. This is even more reason why we need everything to be clear and consistent for this course. So, before going forward discussing bootstrapping, let's get our terminology down. If we have a data set with

repeated measurements, a **bootstrap sample** is an array of length

that was drawn from the original data with replacement. What is a **bootstrap replicate**?

##### Answer the question

#### Possible Answers

* Just another name for a bootstrap sample.
* **A single value of a statistic computed from a bootstrap sample.**
* An actual repeat of the measurements.

**Bootstrapping by hand**

To help you gain intuition about how bootstrapping works, imagine you have a data set that has only three points, [-1, 0, 1]. How many unique bootstrap samples can be drawn (e.g., [-1, 0, 1] and [1, 0, -1] *are* unique), and what is the maximum mean you can get from a bootstrap sample? It might be useful to jot down the samples on a piece of paper.

(These are too few data to get meaningful results from bootstrap procedures, but this example is useful for intuition.)

**Instructions**

**Possible Answers**

* There are 3 unique samples, and the maximum mean is 0.
* There are 10 unique samples, and the maximum mean is 0.
* There are 10 unique samples, and the maximum mean is 1.
* There are 27 unique samples, and the maximum mean is 0.
* **There are 27 unique samples, and the maximum mean is 1.**

**Correct! There are 27 total bootstrap samples, and one of them, [1,1,1] has a mean of 1. Conversely, 7 of them have a mean of zero.**

# Visualizing bootstrap samples

In this exercise, you will generate bootstrap samples from the set of annual rainfall data measured at the Sheffield Weather Station in the UK from 1883 to 2015. The data are stored in the NumPy array rainfall in units of millimeters (mm). By graphically displaying the bootstrap samples with an ECDF, you can get a feel for how bootstrap sampling allows probabilistic descriptions of data.

##### Instructions

**100 XP**

* Write a for loop to acquire 50 bootstrap samples of the rainfall data and plot their ECDF.
  + Use np.random.choice() to generate a bootstrap sample from the NumPy array rainfall. Be sure that the size of the resampled array is len(rainfall).
  + Use the function ecdf() that you wrote in the prequel to this course to generate the x and y values for the ECDF of the bootstrap sample bs\_sample.
  + Plot the ECDF values. Specify color='gray' (to make gray dots) and alpha=0.1 (to make them semi-transparent, since we are overlaying so many) in addition to the marker='.' and linestyle='none' keyword arguments.
* Use ecdf() to generate x and y values for the ECDF of the original rainfall data available in the array rainfall.
* Plot the ECDF values of the original data.
* Hit submit to visualize the samples!
* for \_ in range(50):
* # Generate bootstrap sample: bs\_sample
* bs\_sample = \_\_\_\_(\_\_\_\_, size=\_\_\_\_)
* # Compute and plot ECDF from bootstrap sample
* x, y = \_\_\_\_
* \_ = plt.plot(\_\_\_\_, \_\_\_\_, \_\_\_\_='.', \_\_\_\_='none',
* \_\_\_\_='gray', \_\_\_\_=0.1)
* # Compute and plot ECDF from original data
* x, y = \_\_\_\_
* \_ = plt.plot(\_\_\_\_, \_\_\_\_, \_\_\_\_='.')
* # Make margins and label axes
* plt.margins(0.02)
* \_ = plt.xlabel('yearly rainfall (mm)')
* \_ = plt.ylabel('ECDF')
* # Show the plot
* plt.show()

for \_ in range(50):

    # Generate bootstrap sample: bs\_sample

    bs\_sample = np.random.choice(rainfall, size=len(rainfall))

    # Compute and plot ECDF from bootstrap sample

    x, y = ecdf(bs\_sample)

    \_ = plt.plot(x, y, marker='.', linestyle='none',

                 color='gray', alpha=0.1)

# Compute and plot ECDF from original data

x, y = ecdf(rainfall)

\_ = plt.plot(x, y, marker='.')

# Make margins and label axes

plt.margins(0.02)

\_ = plt.xlabel('yearly rainfall (mm)')

\_ = plt.ylabel('ECDF')

# Show the plot

plt.show()

for \_ in range(50): # Generate bootstrap sample: bs\_sample bs\_sample = np.random.choice(rainfall, size=len(rainfall)) # Compute and plot ECDF from bootstrap sample x, y = ecdf(bs\_sample) \_ = plt.plot(x, y, marker='.', linestyle='none', color='gray', alpha=0.1) # Compute and plot ECDF from original data x, y = ecdf(rainfall) \_ = plt.plot(x, y, marker='.') # Make margins and label axes plt.margins(0.02) \_ = plt.xlabel('yearly rainfall (mm)') \_ = plt.ylabel('ECDF') # Show the plot plt.show()

Good job! Notice how the bootstrap samples give an idea of how the distribution of rainfalls is spread.

**Daily XP1800**

# Bootstrap confidence intervals

**50 XP**

## 1. Bootstrap confidence intervals

In the last video, we learned how to take a set of data, create a bootstrap sample, and then compute a bootstrap replicate of a given statistic. Since we will repeat the replicates over and over again, we can write a function to generate a bootstrap replicate.

## 2. Bootstrap replicate function

We will call the function bootstrap\_replicate\_1d, since it works on one-dimensional arrays. We pass in the data and also a function that computes the statistic of interest. We could pass np dot mean or np dot median, for example. Generating a replicate takes two steps. First, we choose entries out of the data array so that the bootstrap sample has the same number of entries as the original data. Then, we compute the statistic using the specified function. If we call the function, we get a bootstrap replicate. And we can do this over and over again. So, how do we do it over and over again?

## 3. Many bootstrap replicates

With a for loop! First, we have to initialize an array to store our bootstrap replicates. We will make 10,000 replicates, so we use np dot empty to create an empty array. Next, we write a for loop to generate a replicate and store it in the bs\_replicates array. Now that we have the replicates,

## 4. Plotting a histogram of bootstrap replicates

we can make a histogram to see what we might expect to get for the mean of repeated measurements of the speed of light. Note that we use the normed equals True keyword argument. This sets the height of the bars of the histogram such that the total area of the bars is equal to one. This is called

## 5. Bootstrap estimate of the mean

normalization, and we do it so that the histogram approximates a probability density function. You'll recall from the prequel to this course that the area under the PDF gives a probability. So, we have computed the approximate PDF of the mean speed of light we would expect to get if we performed the measurements again. Now we're thinking probabilistically! If we repeat the experiment again and again, we are likely to only see the sample mean vary by about 30 km/s. Now it is useful to summarize this result without having to resort to a graphical method like a histogram. To do this,

## 6. Confidence interval of a statistic

we will compute the 95% confidence interval of the mean. The p% confidence interval is defined as follows. If we repeated measurements over and over again, p% of the observed values would lie within the p% confidence interval. In our case, if we repeated the 100 measurements of the speed of light over and over again, 95% of the sample means would lie within the 95% confidence interval.

## 7. Bootstrap confidence interval

By doing bootstrap replicas, we just "repeated" the experiment over and over again. So, we just use np dot percentile to compute the 2-point-5th and 97-point-5th percentiles to get the 95% confidence interval. This is indeed commensurate with what we see in the histogram.

## 8. Let's practice!

Now it's time for you get some of your own bootstrap confidence intervals.

# Generating many bootstrap replicates

The function bootstrap\_replicate\_1d() from the video is available in your namespace. Now you'll write another function, draw\_bs\_reps(data, func, size=1), which generates many bootstrap replicates from the data set. This function will come in handy for you again and again as you compute confidence intervals and later when you do hypothesis tests.

For your reference, the bootstrap\_replicate\_1d() function is provided below:

def bootstrap\_replicate\_1d(data, func):

"""Generate bootstrap replicate of 1D data."""

bs\_sample = np.random.choice(data, len(data))

return func(bs\_sample)

##### Instructions

**100 XP**

* Define a function with call signature draw\_bs\_reps(data, func, size=1).
  + Using np.empty(), initialize an array called bs\_replicates of size size to hold all of the bootstrap replicates.
  + Write a for loop that ranges over size and computes a replicate using bootstrap\_replicate\_1d(). Refer to the exercise description above to see the function signature of bootstrap\_replicate\_1d(). Store the replicate in the appropriate index of bs\_replicates.
  + Return the array of replicates bs\_replicates. This has already been done for you.
* def draw\_bs\_reps(data, func, size=1):
* """Draw bootstrap replicates."""
* # Initialize array of replicates: bs\_replicates
* bs\_replicates = \_\_\_\_
* # Generate replicates
* for i in \_\_\_\_:
* bs\_replicates[i] = \_\_\_\_
* return bs\_replicates

def draw\_bs\_reps(data, func, size=1):

    """Draw bootstrap replicates."""

    # Initialize array of replicates: bs\_replicates

    bs\_replicates = np.empty(size)

    # Generate replicates

    for i in range(size):

        bs\_replicates[i] = bootstrap\_replicate\_1d(data, func)

    return bs\_replicates

def draw\_bs\_reps(data, func, size=1): """Draw bootstrap replicates.""" # Initialize array of replicates: bs\_replicates bs\_replicates = np.empty(size) # Generate replicates for i in range(size): bs\_replicates[i] = bootstrap\_replicate\_1d(data, func) return bs\_replicates

Good job! This function will be a workhorse for you!

**Bootstrap replicates of the mean and the SEM**

In this exercise, you will compute a bootstrap estimate of the probability density function of the mean annual rainfall at the Sheffield Weather Station. Remember, we are estimating the mean annual rainfall we would get if the Sheffield Weather Station could repeat all of the measurements from 1883 to 2015 over and over again. This is a *probabilistic* estimate of the mean. You will plot the PDF as a histogram, and you will see that it is Normal.

In fact, it can be shown theoretically that under not-too-restrictive conditions, the value of the mean will always be Normally distributed. (This does not hold in general, just for the mean and a few other statistics.) The standard deviation of this distribution, called the **standard error of the mean**, or SEM, is given by the standard deviation of the data divided by the square root of the number of data points. I.e., for a data set, sem = np.std(data) / np.sqrt(len(data)). Using hacker statistics, you get this same result without the need to derive it, but you will verify this result from your bootstrap replicates.

The dataset has been pre-loaded for you into an array called rainfall.

**Instructions**

* Draw 10000 bootstrap replicates of the **mean** annual rainfall using your draw\_bs\_reps() function and the rainfall array. *Hint*: Pass in np.mean for func to compute the mean.
  + As a reminder, draw\_bs\_reps() accepts 3 arguments: data, func, and size.
* Compute and print the standard error of the mean of rainfall.
  + The formula to compute this is np.std(data) / np.sqrt(len(data)).
* Compute and print the standard deviation of your bootstrap replicates bs\_replicates.
* Make a histogram of the replicates using the normed=True keyword argument and 50 bins.

Hit submit to see the plot!

# Take 10,000 bootstrap replicates of the mean: bs\_replicates

bs\_replicates = draw\_bs\_reps(rainfall, np.mean, size=10000)

# Compute and print SEM

sem = np.std(rainfall) / np.sqrt(len(rainfall))

print(sem)

# Compute and print standard deviation of bootstrap replicates

bs\_std = np.std(bs\_replicates)

print(bs\_std)

# Make a histogram of the results

\_ = plt.hist(bs\_replicates, bins=50, normed=True)

\_ = plt.xlabel('mean annual rainfall (mm)')

\_ = plt.ylabel('PDF')

# Show the plot

plt.show()

# Take 10,000 bootstrap replicates of the mean: bs\_replicates

bs\_replicates = draw\_bs\_reps(rainfall, np.mean, size=10000)

# Compute and print SEM

sem = np.std(rainfall) / np.sqrt(len(rainfall))

print(sem)

# Compute and print standard deviation of bootstrap replicates

bs\_std = np.std(bs\_replicates)

print(bs\_std)

# Make a histogram of the results

\_ = plt.hist(bs\_replicates, bins=50, normed=True)

\_ = plt.xlabel('mean annual rainfall (mm)')

\_ = plt.ylabel('PDF')

# Show the plot

plt.show()

10.51054915050619

10.465927071184412

<script.py> output:

10.51054915050619

10.465927071184412

Great work! Notice that the SEM we got from the known expression and the bootstrap replicates is the same and the distribution of the bootstrap replicates of the mean is Normal.

**Confidence intervals of rainfall data**

A *confidence interval* gives upper and lower bounds on the range of parameter values you might expect to get if we repeat our measurements. For named distributions, you can compute them analytically or look them up, but one of the many beautiful properties of the bootstrap method is that you can take percentiles of your bootstrap replicates to get your confidence interval. Conveniently, you can use the np.percentile() function.

Use the bootstrap replicates you just generated to compute the 95% confidence interval. That is, give the 2.5th and 97.5th percentile of your bootstrap replicates stored as bs\_replicates. What is the 95% confidence interval?

**Instructions**

* (765, 776) mm/year
* (780, 821) mm/year
* (761, 817) mm/year
* (761, 841) mm/year
* **np.percentile(bs\_replicates,[2.5,97.5])**
* **array([779.76992481, 820.95043233])**

Correct! See, it's simple to get confidence intervals using bootstrap!

**Bootstrap replicates of other statistics**

We saw in a previous exercise that the mean is Normally distributed. This does not necessarily hold for other statistics, but no worry: as hackers, we can always take bootstrap replicates! In this exercise, you'll generate bootstrap replicates for the variance of the annual rainfall at the Sheffield Weather Station and plot the histogram of the replicates.

Here, you will make use of the draw\_bs\_reps() function you defined a few exercises ago. It is provided below for your reference:

def draw\_bs\_reps(data, func, size=1):

"""Draw bootstrap replicates."""

# Initialize array of replicates

bs\_replicates = np.empty(size)

# Generate replicates

for i in range(size):

bs\_replicates[i] = bootstrap\_replicate\_1d(data, func)

return bs\_replicates

**Instructions**

Draw 10000 bootstrap replicates of the **variance** in annual rainfall, stored in the rainfall dataset, using your draw\_bs\_reps() function. *Hint*: Pass in np.var for computing the variance.

Divide your variance replicates (bs\_replicates) by 100 to put the variance in units of square centimeters for convenience.

Make a histogram of bs\_replicates using the normed=True keyword argument and 50 bins.

# Generate 10,000 bootstrap replicates of the variance: bs\_replicates

bs\_replicates = \_\_\_\_

# Put the variance in units of square centimeters

\_\_\_\_

# Make a histogram of the results

\_ = plt.hist(\_\_\_\_, \_\_\_\_, \_\_\_\_)

\_ = plt.xlabel('variance of annual rainfall (sq. cm)')

\_ = plt.ylabel('PDF')

# Show the plot

plt.show()

# Generate 10,000 bootstrap replicates of the variance: bs\_replicates

bs\_replicates = draw\_bs\_reps(rainfall, np.var, size=10000)

# Put the variance in units of square centimeters

bs\_replicates =bs\_replicates / 100

# Make a histogram of the results

\_ = plt.hist(bs\_replicates, bins=50, normed=True)

\_ = plt.xlabel('variance of annual rainfall (sq. cm)')

\_ = plt.ylabel('PDF')

# Show the plot

plt.show()

# Generate 10,000 bootstrap replicates of the variance: bs\_replicates bs\_replicates = draw\_bs\_reps(rainfall, np.var, size=10000) # Put the variance in units of square centimeters bs\_replicates =bs\_replicates / 100 # Make a histogram of the results \_ = plt.hist(bs\_replicates, bins=50, normed=True) \_ = plt.xlabel('variance of annual rainfall (sq. cm)') \_ = plt.ylabel('PDF') # Show the plot plt.show()

Great work! This is not normally distributed, as it has a longer tail to the right. Note that you can also compute a confidence interval on the variance, or any other statistic, using np.percentile() with your bootstrap replicates.

**np.percentile(bs\_replicates, [2.5, 97.5])**

**array([114.87926838, 179.45385568])**

**Confidence interval on the rate of no-hitters**

Consider again the inter-no-hitter intervals for the modern era of baseball. Generate 10,000 bootstrap replicates of the optimal parameter

Plot a histogram of your replicates and report a 95% confidence interval.

**Instructions**

Generate 10000 bootstrap replicates of from the nohitter\_times data using your draw\_bs\_reps() function. Recall that the optimal is calculated as the **mean** of the data.

Compute the 95% confidence interval using np.percentile() and passing in two arguments: The array bs\_replicates, and the list of percentiles - in this case 2.5 and 97.5.

Print the confidence interval.

Plot a histogram of your bootstrap replicates. This has been done for you, so hit submit to see the plot!

# Draw bootstrap replicates of the mean no-hitter time (equal to tau): bs\_replicates

bs\_replicates = \_\_\_\_

# Compute the 95% confidence interval: conf\_int

conf\_int = \_\_\_\_

# Print the confidence interval

print('95% confidence interval =', \_\_\_\_, 'games')

# Plot the histogram of the replicates

\_ = plt.hist(bs\_replicates, bins=50, normed=True)

\_ = plt.xlabel(r'$\tau$ (games)')

\_ = plt.ylabel('PDF')

# Show the plot

plt.show()

# Draw bootstrap replicates of the mean no-hitter time (equal to tau): bs\_replicates

bs\_replicates = draw\_bs\_reps(nohitter\_times, np.mean, size=10000)

# Compute the 95% confidence interval: conf\_int

conf\_int = np.percentile(bs\_replicates, [2.5, 97.5])

# Print the confidence interval

print('95% confidence interval =', conf\_int, 'games')

# Plot the histogram of the replicates

\_ = plt.hist(bs\_replicates, bins=50, normed=True)

\_ = plt.xlabel(r'$\tau$ (games)')

\_ = plt.ylabel('PDF')

# Show the plot

plt.show()

# Draw bootstrap replicates of the mean no-hitter time (equal to tau): bs\_replicates

bs\_replicates = draw\_bs\_reps(nohitter\_times, np.mean, size=10000)

# Compute the 95% confidence interval: conf\_int

conf\_int = np.percentile(bs\_replicates, [2.5, 97.5])

# Print the confidence interval

print('95% confidence interval =', conf\_int, 'games')

# Plot the histogram of the replicates

\_ = plt.hist(bs\_replicates, bins=50, normed=True)

\_ = plt.xlabel(r'$\tau$ (games)')

\_ = plt.ylabel('PDF')

# Show the plot

plt.show()

95% confidence interval = [660.67280876 871.63077689] games

<script.py> output:

95% confidence interval = [660.67280876 871.63077689] games

This gives you an estimate of what the typical time between no-hitters is. It could be anywhere between 660 and 870 games.

**Daily XP100**

# Pairs bootstrap

**50 XP**

## 1. Pairs bootstrap

When we computed bootstrap confidence intervals on summary statistics, we did so

## 2. Nonparametric inference

nonparametrically. By this, I mean that we did not assume any model underlying the data; the estimates were done using the data alone.

## 3. 2008 US swing state election results

When we performed a linear least squares regression, however, we were using a linear model, which has two parameters, the slope and intercept. This was a parametric estimate. The optimal parameter values we compute for our parametric model are like other statistics, in that we would get different values for them if we acquired the data again. We can perform bootstrap estimates to get confidence intervals on the slope and intercept as well. Remember: we need to think probabilistically. Let's consider the swing state election data from the prequel to this course. What if we had the election again, under identical conditions? How would the slope and intercept change? This is kind of a tricky question; there are several ways to get bootstrap estimates of the confidence intervals on these parameters, each of which makes difference assumptions about the data. We will do a method that makes the least assumptions,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 4. Pairs bootstrap for linear regression

called pairs bootstrap. Since we cannot resample individual data because each county has two variables associated with it, the vote share for Obama and the total number of votes, we resample pairs. For the election data, we could randomly select a given county, and keep its total votes and Democratic share as a pair. So our bootstrap sample consists of a set (x,y) pairs. We then compute the slope and intercept from this pairs bootstrap sample to get the bootstrap replicates. You can get confidence intervals from many bootstrap replicates of the slope and intercept, just like before. Let's see how this works in practice.

## 5. Generating a pairs bootstrap sample

Because np dot random dot choice must sample a 1D array, we will sample the indices of the data points. We can generate the indices of a NumPy array using the np dot arrange function. It give us an array of sequential integers. We then sample the indices with replacement. The bootstrap sample is generated by slicing out the respective values from the original data arrays. With these in hand,

## 6. Computing a pairs bootstrap replicate

we can perform a linear regression using np dot polyfit on the pairs bootstrap sample to get a bootstrap replicate. If we compare the result to the linear regression on the original data, they are close, but not equal. As we have seen before, you can use many of these replicates to generate bootstrap confidence intervals for the slope and intercept using np dot percentile. You can also

## 7. 2008 US swing state election results

plot the lines you get from your bootstrap replicates to get a graphic idea how the regression line may change if the data were collected again. You will work through this whole procedure in the exercises.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 8. Let's practice!

When you do, always keep in mind that you are thinking probabilistically. Getting an optimal parameter value is the first step. Now, you are finding out how that parameter is likely to change upon repeated measurements. Happy coding!

**A function to do pairs bootstrap**

As discussed in the video, pairs bootstrap involves resampling pairs of data. Each collection of pairs fit with a line, in this case using np.polyfit(). We do this again and again, getting bootstrap replicates of the parameter values. To have a useful tool for doing pairs bootstrap, you will write a function to perform pairs bootstrap on a set of x,y data.

**Instructions**

Define a function with call signature draw\_bs\_pairs\_linreg(x, y, size=1) to perform pairs bootstrap estimates on linear regression parameters.

Use np.arange() to set up an array of indices going from 0 to len(x). These are what you will resample and use them to pick values out of the x and y arrays.

Use np.empty() to initialize the slope and intercept replicate arrays to be of size size.

Write a for loop to:

Resample the indices inds. Use np.random.choice() to do this.

Make new x and y arrays bs\_x and bs\_y using the the resampled indices bs\_inds. To do this, slice x and y with bs\_inds.

Use np.polyfit() on the new x and y arrays and store the computed slope and intercept.

Return the pair bootstrap replicates of the slope and intercept.

def draw\_bs\_pairs\_linreg(x, y, size=1):

    """Perform pairs bootstrap for linear regression."""

    # Set up array of indices to sample from: inds

    inds = \_\_\_\_

    # Initialize replicates: bs\_slope\_reps, bs\_intercept\_reps

    bs\_slope\_reps = \_\_\_\_

    bs\_intercept\_reps = \_\_\_\_

    # Generate replicates

    for i in range(size):

        bs\_inds = np.random.choice(\_\_\_\_, size=\_\_\_\_)

        bs\_x, bs\_y = x[\_\_\_\_], y[\_\_\_\_]

        bs\_slope\_reps[i], bs\_intercept\_reps[i] = \_\_\_\_

    return bs\_slope\_reps, bs\_intercept\_reps

def draw\_bs\_pairs\_linreg(x, y, size=1):

    """Perform pairs bootstrap for linear regression."""

    # Set up array of indices to sample from: inds

    inds = np.arange(0, len(x))

    # Initialize replicates: bs\_slope\_reps, bs\_intercept\_reps

    bs\_slope\_reps = np.empty(size)

    bs\_intercept\_reps = np.empty(size)

    # Generate replicates

    for i in range(size):

        bs\_inds = np.random.choice(inds, size=len(inds))

        bs\_x, bs\_y = x[bs\_inds], y[bs\_inds]

        bs\_slope\_reps[i], bs\_intercept\_reps[i] = np.polyfit(bs\_x, bs\_y, 1)

    return bs\_slope\_reps, bs\_intercept\_reps

def draw\_bs\_pairs\_linreg(x, y, size=1): """Perform pairs bootstrap for linear regression.""" # Set up array of indices to sample from: inds inds = np.arange(0, len(x)) # Initialize replicates: bs\_slope\_reps, bs\_intercept\_reps bs\_slope\_reps = np.empty(size) bs\_intercept\_reps = np.empty(size) # Generate replicates for i in range(size): bs\_inds = np.random.choice(inds, size=len(inds)) bs\_x, bs\_y = x[bs\_inds], y[bs\_inds] bs\_slope\_reps[i], bs\_intercept\_reps[i] = np.polyfit(bs\_x, bs\_y, 1) return bs\_slope\_reps, bs\_intercept\_reps

# Pairs bootstrap of literacy/fertility data

Using the function you just wrote, perform pairs bootstrap to plot a histogram describing the estimate of the slope from the illiteracy/fertility data. Also report the 95% confidence interval of the slope. The data is available to you in the NumPy arrays illiteracy and fertility.

As a reminder, draw\_bs\_pairs\_linreg() has a function signature of draw\_bs\_pairs\_linreg(x, y, size=1), and it returns two values: bs\_slope\_reps and bs\_intercept\_reps.

##### Instructions

**100 XP**

* Use your draw\_bs\_pairs\_linreg() function to take 1000 bootstrap replicates of the slope and intercept. The x-axis data is illiteracy and y-axis data is fertility.
* Compute and print the 95% bootstrap confidence interval for the slope.
* Plot and show a histogram of the slope replicates. Be sure to label your axes. This has been done for you, so click submit to see your histogram!
* # Generate replicates of slope and intercept using pairs bootstrap
* bs\_slope\_reps, bs\_intercept\_reps = \_\_\_\_
* # Compute and print 95% CI for slope
* print(np.percentile(\_\_\_\_, \_\_\_\_))
* # Plot the histogram
* \_ = plt.hist(bs\_slope\_reps, bins=50, normed=True)
* \_ = plt.xlabel('slope')
* \_ = plt.ylabel('PDF')
* plt.show()

# Generate replicates of slope and intercept using pairs bootstrap

bs\_slope\_reps, bs\_intercept\_reps = draw\_bs\_pairs\_linreg(illiteracy, fertility, size=1000)

# Compute and print 95% CI for slope

print(np.percentile(bs\_slope\_reps, [2.5, 97.5]))

# Plot the histogram

\_ = plt.hist(bs\_slope\_reps, bins=50, normed=True)

\_ = plt.xlabel('slope')

\_ = plt.ylabel('PDF')

plt.show()

**# Generate replicates of slope and intercept using pairs bootstrap**

**bs\_slope\_reps, bs\_intercept\_reps = draw\_bs\_pairs\_linreg(illiteracy, fertility, size=1000)**

**# Compute and print 95% CI for slope**

**print(np.percentile(bs\_slope\_reps, [2.5, 97.5]))**

**# Plot the histogram**

**\_ = plt.hist(bs\_slope\_reps, bins=50, normed=True)**

**\_ = plt.xlabel('slope')**

**\_ = plt.ylabel('PDF')**

**plt.show()**

**[0.04378061 0.0551616 ]**

**<script.py> output:**

**[0.04378061 0.0551616** ]

# Plotting bootstrap regressions

A nice way to visualize the variability we might expect in a linear regression is to plot the line you would get from each bootstrap replicate of the slope and intercept. Do this for the first 100 of your bootstrap replicates of the slope and intercept (stored as bs\_slope\_reps and bs\_intercept\_reps).

##### Instructions

**100 XP**

* Generate an array of
* -values consisting of 0 and 100 for the plot of the regression lines. Use the np.array() function for this.
* Write a for loop in which you plot a regression line with a slope and intercept given by the pairs bootstrap replicates. Do this for 100 lines.
  + When plotting the regression lines in each iteration of the for loop, recall the regression equation y = a\*x + b. Here, a is bs\_slope\_reps[i] and b is bs\_intercept\_reps[i].
  + Specify the keyword arguments linewidth=0.5, alpha=0.2, and color='red' in your call to plt.plot().
* Make a scatter plot with illiteracy on the x-axis and fertility on the y-axis. Remember to specify the marker='.' and linestyle='none' keyword arguments.
* Label the axes, set a 2% margin, and show the plot. This has been done for you, so hit submit to visualize the bootstrap regressions!
* # Generate array of x-values for bootstrap lines: x
* x = \_\_\_\_
* # Plot the bootstrap lines
* for i in range(\_\_\_\_):
* \_ = plt.plot(\_\_\_\_,
* \_\_\_\_\*x + \_\_\_\_,
* \_\_\_\_=0.5, \_\_\_\_=0.2, \_\_\_\_='red')
* # Plot the data
* \_ = \_\_\_\_
* # Label axes, set the margins, and show the plot
* \_ = plt.xlabel('illiteracy')
* \_ = plt.ylabel('fertility')
* plt.margins(0.02)
* plt.show()

# Generate array of x-values for bootstrap lines: x

x = np.array([0, 100])

# Plot the bootstrap lines

for i in range(100):

    \_ = plt.plot(x,

                 bs\_slope\_reps[i]\*x + bs\_intercept\_reps[i],

                 linewidth=0.5, alpha=0.2, color='red')

# Plot the data

\_ = plt.plot(illiteracy, fertility,marker='.', linestyle='none')

# Label axes, set the margins, and show the plot

\_ = plt.xlabel('illiteracy')

\_ = plt.ylabel('fertility')

plt.margins(0.02)

plt.show()

# Generate array of x-values for bootstrap lines: x x = np.array([0, 100]) # Plot the bootstrap lines for i in range(100): \_ = plt.plot(x, bs\_slope\_reps[i]\*x + bs\_intercept\_reps[i], linewidth=0.5, alpha=0.2, color='red') # Plot the data \_ = plt.plot(illiteracy, fertility,marker='.', linestyle='none') # Label axes, set the margins, and show the plot \_ = plt.xlabel('illiteracy') \_ = plt.ylabel('fertility') plt.margins(0.02) plt.show()

Great work! You now have some serious chops for parameter estimation. Let's move on to hypothesis testing!

**Daily XP450**

# Formulating and simulating a hypothesis

**50 XP**

## 1. Formulating and simulating a hypothesis

When we studied linear regression, we assumed a linear model

## 2. 2008 US swing state election results

for how the data are generated and then estimated the parameters that are defined by that model. But, how to we assess how reasonable it is that our observed data are actually described by the model? This is the realm of hypothesis testing. Let's start by thinking about a simpler scenario. Consider the following.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. Insert title here...

Ohio and Pennsylvania are similar states. They are neighbors and they both have liberal urban counties and also lots of rural conservative counties. I hypothesize that county-level voting in these two states have identical probability distributions. We have voting data to help test if this hypothesis. Stated more concretely,

## 4. Hypothesis testing

we are going to assess how reasonable the observed data are assuming the hypothesis is true. The hypothesis we are testing is

## 5. Null hypothesis

typically called the null hypothesis. We might start by just plotting the two ECDFs of

## 6. ECDFs of swing state election results

the county-level votes. Whew! It is pretty tough to make a judgment here. Pennsylvania seems to be slightly more toward Obama in the middle part of the ECDFs, but not much. We can't really draw a conclusion here.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 7. Percent vote for Obama

We could just compare some summary statistics. Again, this is a tough call. The means and medians of the two states are really close, and the standard deviations are almost identical. So eyeballing the data is not enough. To resolve this issue,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 8. Simulating the hypothesis

we can simulate what the data would look like if the county-level voting trends in the two states were identically distributed. We can do this by putting the Democratic share of the vote for all of Pennsylvania's 67 counties and Ohio's 88 counties together.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 9. Simulating the hypothesis

We then ignore what state they belong to. Next, we randomly scramble

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 10. Simulating the hypothesis

the ordering of the counties.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 11. Simulating the hypothesis

We then re-label the first 67 to be "Pennsylvania" and the remaining ones to be "Ohio." So, we just redid the election as if there was no difference between Pennsylvania and Ohio.

## 12. Permutation

This technique, of scrambling the order of an array, is called a permutation. It is at the heart of simulating a null hypothesis were we assume two quantities are identically distributed.

## 13. Generating a permutation sample

Let's look at how we can implement this in Python. First, we need to make a single array with all of the counties in it. We do this using the np dot concatenate function. Notice that this function takes a tuple of the arrays you wish to concatenate as an argument. Next, we use the function np dot random dot permutation to conveniently permute the entries of the array. We then assign the first 67 to be labeled Pennsylvania and the last 88 to be labeled Ohio. These samples are called permutation samples.

## 14. Let's practice!

Now, let's practice doing some permutation sampling of real data!

# Generating a permutation sample

In the video, you learned that permutation sampling is a great way to simulate the hypothesis that two variables have identical probability distributions. This is often a hypothesis you want to test, so in this exercise, you will write a function to generate a permutation sample from two data sets.

Remember, a permutation sample of two arrays having respectively n1 and n2 entries is constructed by concatenating the arrays together, scrambling the contents of the concatenated array, and then taking the first n1 entries as the permutation sample of the first array and the last n2 entries as the permutation sample of the second array.

##### Instructions

**100 XP**

* Concatenate the two input arrays into one using np.concatenate(). Be sure to pass in data1 and data2 as one argument (data1, data2).
* Use np.random.permutation() to permute the concatenated array.
* Store the first len(data1) entries of permuted\_data as perm\_sample\_1 and the last len(data2) entries of permuted\_data as perm\_sample\_2. In practice, this can be achieved by using :len(data1) and len(data1): to slice permuted\_data.
* Return perm\_sample\_1 and perm\_sample\_2.
* def permutation\_sample(data1, data2):
* """Generate a permutation sample from two data sets."""
* # Concatenate the data sets: data
* data = \_\_\_\_
* # Permute the concatenated array: permuted\_data
* permuted\_data = \_\_\_\_
* # Split the permuted array into two: perm\_sample\_1, perm\_sample\_2
* perm\_sample\_1 = permuted\_data[\_\_\_\_]
* perm\_sample\_2 = permuted\_data[\_\_\_\_]
* return perm\_sample\_1, perm\_sample\_2

def permutation\_sample(data1, data2):

    """Generate a permutation sample from two data sets."""

    # Concatenate the data sets: data

    data = np.concatenate((data1, data2))

    # Permute the concatenated array: permuted\_data

    permuted\_data = np.random.permutation(data)

    # Split the permuted array into two: perm\_sample\_1, perm\_sample\_2

    perm\_sample\_1 = permuted\_data[:len(data1)]

    perm\_sample\_2 = permuted\_data[len(data1):]

    return perm\_sample\_1, perm\_sample\_2

def permutation\_sample(data1, data2): """Generate a permutation sample from two data sets.""" # Concatenate the data sets: data data = np.concatenate((data1, data2)) # Permute the concatenated array: permuted\_data permuted\_data = np.random.permutation(data) # Split the permuted array into two: perm\_sample\_1, perm\_sample\_2 perm\_sample\_1 = permuted\_data[:len(data1)] perm\_sample\_2 = permuted\_data[len(data1):] return perm\_sample\_1, perm\_sample\_2

# Visualizing permutation sampling

To help see how permutation sampling works, in this exercise you will generate permutation samples and look at them graphically.

We will use the Sheffield Weather Station data again, this time considering the monthly rainfall in June (a dry month) and November (a wet month). We expect these might be differently distributed, so we will take permutation samples to see how their ECDFs would look if they were identically distributed.

The data are stored in the NumPy arrays rain\_june and rain\_november.

As a reminder, permutation\_sample() has a function signature of permutation\_sample(data\_1, data\_2) with a return value of permuted\_data[:len(data\_1)], permuted\_data[len(data\_1):], where permuted\_data = np.random.permutation(np.concatenate((data\_1, data\_2))).

##### Instructions

* Write a for loop to generate 50 permutation samples, compute their ECDFs, and plot them.
  + Generate a permutation sample pair from rain\_june and rain\_november using your permutation\_sample() function.
  + Generate the x and y values for an ECDF for each of the two permutation samples for the ECDF using your ecdf() function.
  + Plot the ECDF of the first permutation sample (x\_1 and y\_1) as dots. Do the same for the second permutation sample (x\_2 and y\_2).
* Generate x and y values for ECDFs for the rain\_june and rain\_november data and plot the ECDFs using respectively the keyword arguments color='red' and color='blue'.
* Label your axes, set a 2% margin, and show your plot. This has been done for you, so just hit submit to view the plot!
* for \_ in \_\_\_\_:
* # Generate permutation samples
* perm\_sample\_1, perm\_sample\_2 = \_\_\_\_
* # Compute ECDFs
* x\_1, y\_1 = \_\_\_\_
* x\_2, y\_2 = \_\_\_\_
* # Plot ECDFs of permutation sample
* \_ = plt.plot(\_\_\_\_, \_\_\_\_, marker='.', linestyle='none',
* color='red', alpha=0.02)
* \_ = plt.plot(\_\_\_\_, \_\_\_\_, marker='.', linestyle='none',
* color='blue', alpha=0.02)
* # Create and plot ECDFs from original data
* x\_1, y\_1 = \_\_\_\_
* x\_2, y\_2 = \_\_\_\_
* \_ = plt.plot(x\_1, y\_1, marker='.', linestyle='none', color='red')
* \_ = plt.plot(x\_2, y\_2, marker='.', linestyle='none', color='blue')
* # Label axes, set margin, and show plot
* plt.margins(0.02)
* \_ = plt.xlabel('monthly rainfall (mm)')
* \_ = plt.ylabel('ECDF')
* plt.show()

for i in range(50):

    # Generate permutation samples

    perm\_sample\_1, perm\_sample\_2 = permutation\_sample(rain\_june, rain\_november)

    # Compute ECDFs

    x\_1, y\_1 = ecdf(perm\_sample\_1)

    x\_2, y\_2 = ecdf(perm\_sample\_2)

    # Plot ECDFs of permutation sample

    \_ = plt.plot(x\_1, y\_1, marker='.', linestyle='none',

                 color='red', alpha=0.02)

    \_ = plt.plot(x\_2, y\_2, marker='.', linestyle='none',

                 color='blue', alpha=0.02)

# Create and plot ECDFs from original data

x\_1, y\_1 = ecdf(rain\_june)

x\_2, y\_2 = ecdf(rain\_november)

\_ = plt.plot(x\_1, y\_1, marker='.', linestyle='none', color='red')

\_ = plt.plot(x\_2, y\_2, marker='.', linestyle='none', color='blue')

# Label axes, set margin, and show plot

plt.margins(0.02)

\_ = plt.xlabel('monthly rainfall (mm)')

\_ = plt.ylabel('ECDF')

plt.show()

for i in range(50): # Generate permutation samples perm\_sample\_1, perm\_sample\_2 = permutation\_sample(rain\_june, rain\_november) # Compute ECDFs x\_1, y\_1 = ecdf(perm\_sample\_1) x\_2, y\_2 = ecdf(perm\_sample\_2) # Plot ECDFs of permutation sample \_ = plt.plot(x\_1, y\_1, marker='.', linestyle='none', color='red', alpha=0.02) \_ = plt.plot(x\_2, y\_2, marker='.', linestyle='none', color='blue', alpha=0.02) # Create and plot ECDFs from original data x\_1, y\_1 = ecdf(rain\_june) x\_2, y\_2 = ecdf(rain\_november) \_ = plt.plot(x\_1, y\_1, marker='.', linestyle='none', color='red') \_ = plt.plot(x\_2, y\_2, marker='.', linestyle='none', color='blue') # Label axes, set margin, and show plot plt.margins(0.02) \_ = plt.xlabel('monthly rainfall (mm)') \_ = plt.ylabel('ECDF') plt.show()

Great work! Notice that the permutation samples ECDFs overlap and give a purple haze. None of the ECDFs from the permutation samples overlap with the observed data, suggesting that the hypothesis is not commensurate with the data. June and November rainfall are not identically distributed.

# Test statistics and p-values

**50 XP**

## 1. Test statistics and p-values

Now that we know how to simulate the null hypothesis using permutation, we can start to test it. We will continue our study of hypothesis testing with

## 2. Are OH and PA different?

the Ohio/Pennsylvania vote data. We are testing the null hypothesis that the county-level voting is identically distributed between the two states. Remember that

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. Hypothesis testing

testing a hypothesis is an assessment of how reasonable the observed data are assuming the hypothesis is true. But this is a bit vague. What about the data do we assess and how do we quantify the assessment? The answer to these questions hinges on the concept of

## 4. Test statistic

a test statistic. A test statistic is a single number that can be computed from observed data and also from data you simulate under the null hypothesis. It serves as a basis of comparison between what the hypothesis predicts and what we actually observed. Importantly, you should choose your test statistic to be something that is pertinent to the question you are trying to answer with your hypothesis test, in this case, are the two states different? If they are identical, they should have the same mean vote share for Obama. So the difference in mean vote share should be zero. We will therefore choose the difference in means as our test statistic.

## 5. Permutation replicate

From the permutation sample we generated in the last video, the value of the test statistic is 1-point-12%. The value of a test statistic computed from a permutation sample is called a permutation replicate, in this case, 1-point-12%. We already calculated that the difference in mean vote share from the actual election was 1-point-16%. So, for this permutation replicate, we did not quite get as big of a difference in means than what was observed in the original data. Now, we can "redo" the election 10,000 times under the null hypothesis by generating lots and lots of permutation replicates. (You will write for loops to do this in the exercises.)

## 6. Mean vote difference under null hypothesis

We can plot a histogram of all the permutation replicates. The difference of means from the elections simulated under the null hypothesis lies somewhere between -4 and 4%. The actual mean percent vote difference was 1-point-16%, shown by the red line. If we tally up the area of the histogram that is

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 7. Mean vote difference under null hypothesis

to the right of the read line, we get that about 23% of the simulated elections had at least a 1-point-16% difference or greater. This value, point-23, is called

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 8. p-value

the p-value. It is the probability of getting at least a 1-point-16% difference in the mean vote share assuming the states have identically distributed voting. So is it plausible that we would observe the vote share we got if Pennsylvania and Ohio had identically distributed county-level voting? Sure it is. It happened 23% of the time under the null hypothesis. Now, we have to be careful about the definition of the p-value. Again, the p-value is the probability of obtaining a value of your test statistic that is at least as extreme as what was observed, under the assumption the null hypothesis is true. The p-value is exactly that. It is not the probability that the null hypothesis is true. Further, the p-value is only meaningful if the null hypothesis is clearly stated, along with the test statistic used to evaluate it. When the p-value is small, it is often said that the data are

## 9. Statistical significance

statistically significantly different than what we would observe under the null hypothesis. For this reason, the hypothesis testing we're doing is sometimes called

## 10. Null hypothesis significance testing (NHST)

null hypothesis significance testing, or NHST. I encourage you not to just label something as statistically significant or not, but rather to consider the value of the p-value, as well as how much different the data are from what you would expect from the null hypothesis.

## 11. statistical significance ? practical significance

Remember: statistical significance (that is, low p-values) and practical significance, whether or not the difference of the data from the null hypothesis matters for practical considerations, are two different things.

## 12. Let's practice!

Ok, now let's perform some hypothesis tests!

# Test statistics

When performing hypothesis tests, your choice of test statistic should be:

##### Answer the question

* something well-known, like the mean or median.
* be a parameter that can be estimated.
* **be pertinent to the question you are seeking to answer in your hypothesis test.**
* **Yes! The most important thing to consider is: What are you asking?**

# What is a p-value?

The p-value is generally a measure of:

##### Answer the question

* the probability that the hypothesis you are testing is true.
* the probability of observing your data if the hypothesis you are testing is true.
* **the probability of observing a test statistic equally or more extreme than the one you observed, given that the null hypothesis is true.**

# Generating permutation replicates

As discussed in the video, a permutation replicate is a single value of a statistic computed from a permutation sample. As the draw\_bs\_reps() function [you wrote in chapter 2](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-2/bootstrap-confidence-intervals?ex=6) is useful for you to generate bootstrap replicates, it is useful to have a similar function, draw\_perm\_reps(), to generate permutation replicates. You will write this useful function in this exercise.

The function has call signature draw\_perm\_reps(data\_1, data\_2, func, size=1). Importantly, func must be a function that takes two arrays as arguments. In most circumstances, func will be a function you write yourself.

##### Instructions

* Define a function with this signature: draw\_perm\_reps(data\_1, data\_2, func, size=1).
  + Initialize an array to hold the permutation replicates using np.empty().
  + Write a for loop to:
    - Compute a permutation sample using your permutation\_sample() function
    - Pass the samples into func() to compute the replicate and store the result in your array of replicates.
  + Return the array of replicates.
* def draw\_perm\_reps(data\_1, data\_2, func, size=1):
* """Generate multiple permutation replicates."""
* # Initialize array of replicates: perm\_replicates
* perm\_replicates = \_\_\_\_
* for i in \_\_\_\_:
* # Generate permutation sample
* perm\_sample\_1, perm\_sample\_2 = \_\_\_\_
* # Compute the test statistic
* perm\_replicates[i] = \_\_\_\_
* return perm\_replicates

def draw\_perm\_reps(data\_1, data\_2, func, size=1):

    """Generate multiple permutation replicates."""

    # Initialize array of replicates: perm\_replicates

    perm\_replicates = np.empty(size)

    for i in range(size):

        # Generate permutation sample

        perm\_sample\_1, perm\_sample\_2 = permutation\_sample(data\_1,data\_2)

        # Compute the test statistic

        perm\_replicates[i] = func(perm\_sample\_1, perm\_sample\_2)

    return perm\_replicates

def draw\_perm\_reps(data\_1, data\_2, func, size=1):

    """Generate multiple permutation replicates."""

    # Initialize array of replicates: perm\_replicates

    perm\_replicates = np.empty(size)

    for i in range(size):

        # Generate permutation sample

        perm\_sample\_1, perm\_sample\_2 = permutation\_sample(data\_1,data\_2)

        # Compute the test statistic

        perm\_replicates[i] = func(perm\_sample\_1, perm\_sample\_2)

    return perm\_replicates

def draw\_perm\_reps(data\_1, data\_2, func, size=1): """Generate multiple permutation replicates.""" # Initialize array of replicates: perm\_replicates perm\_replicates = np.empty(size) for i in range(size): # Generate permutation sample perm\_sample\_1, perm\_sample\_2 = permutation\_sample(data\_1,data\_2) # Compute the test statistic perm\_replicates[i] = func(perm\_sample\_1, perm\_sample\_2) return perm\_replicates

# Look before you leap: EDA before hypothesis testing

Kleinteich and Gorb (Sci. Rep., **4**, 5225, 2014) performed an interesting experiment with South American horned frogs. They held a plate connected to a force transducer, along with a bait fly, in front of them. They then measured the impact force and adhesive force of the frog's tongue when it struck the target.

Frog A is an adult and Frog B is a juvenile. The researchers measured the impact force of 20 strikes for each frog. In the next exercise, we will test the hypothesis that the two frogs have the same distribution of impact forces. But, remember, it is important to do EDA first! Let's make a bee swarm plot for the data. They are stored in a pandas DataFrame, df, where column ID is the identity of the frog and column impact\_force is the impact force in Newtons (N).

##### Instructions

* Use sns.swarmplot() to make a bee swarm plot of the data by specifying the x, y, and data keyword arguments.
* Label your axes.
* Show the plot.
* # Make bee swarm plot
* \_ = \_\_\_\_
* # Label axes
* \_ = plt.\_\_\_\_('frog')
* \_ = plt.\_\_\_\_('impact force (N)')
* # Show the plot
* \_\_\_\_

# Make bee swarm plot

\_ = sns.swarmplot(x='ID',y='impact\_force',data=df)

# Label axes

\_ = plt.xlabel('frog')

\_ = plt.ylabel('impact force (N)')

# Show the plot

plt.show()

# Make bee swarm plot \_ = sns.swarmplot(x='ID',y='impact\_force',data=df) # Label axes \_ = plt.xlabel('frog') \_ = plt.ylabel('impact force (N)') # Show the plot plt.show()

Eyeballing it, it does not look like they come from the same distribution. Frog A, the adult, has three or four very hard strikes, and Frog B, the juvenile, has a couple weak ones. However, it is possible that with only 20 samples it might be too difficult to tell if they have difference distributions, so we should proceed with the hypothesis test.

# Permutation test on frog data

The average strike force of Frog A was 0.71 Newtons (N), and that of Frog B was 0.42 N for a difference of 0.29 N. It is possible the frogs strike with the same force and this observed difference was by chance. You will compute the probability of getting at least a 0.29 N difference in mean strike force under the hypothesis that the distributions of strike forces for the two frogs are identical. We use a permutation test with a test statistic of the difference of means to test this hypothesis.

For your convenience, the data has been stored in the arrays force\_a and force\_b.

##### Instructions

* Define a function with call signature diff\_of\_means(data\_1, data\_2) that returns the differences in means between two data sets, mean of data\_1 minus mean of data\_2.
* Use this function to compute the empirical difference of means that was observed in the frogs.
* Draw 10,000 permutation replicates of the difference of means.
* Compute the p-value.
* Print the p-value.
* def diff\_of\_means(data\_1, data\_2):
* """Difference in means of two arrays."""
* # The difference of means of data\_1, data\_2: diff
* diff = \_\_\_\_
* return diff
* # Compute difference of mean impact force from experiment: empirical\_diff\_means
* empirical\_diff\_means = \_\_\_\_
* # Draw 10,000 permutation replicates: perm\_replicates
* perm\_replicates = draw\_perm\_reps(\_\_\_\_, \_\_\_\_,
* \_\_\_\_, size=10000)
* # Compute p-value: p
* p = np.sum(\_\_\_\_ >= \_\_\_\_) / len(\_\_\_\_)
* # Print the result
* print('p-value =', p)

def diff\_of\_means(data\_1, data\_2):

    """Difference in means of two arrays."""

    # The difference of means of data\_1, data\_2: diff

    diff = np.mean(data\_1) - np.mean(data\_2)

    return diff

# Compute difference of mean impact force from experiment: empirical\_diff\_means

empirical\_diff\_means = diff\_of\_means(force\_a, force\_b)

# Draw 10,000 permutation replicates: perm\_replicates

perm\_replicates = draw\_perm\_reps(force\_a, force\_b,

                                 diff\_of\_means, size=10000)

# Compute p-value: p

p = np.sum(perm\_replicates >= empirical\_diff\_means) / len(perm\_replicates)

# Print the result

print('p-value =', p)

**def diff\_of\_means(data\_1, data\_2):**

**"""Difference in means of two arrays."""**

**# The difference of means of data\_1, data\_2: diff**

**diff = data\_1 - data\_2**

**return diff**

**# Compute difference of mean impact force from experiment: empirical\_diff\_means**

**empirical\_diff\_means = diff\_of\_means(force\_a, force\_b)**

**# Draw 10,000 permutation replicates: perm\_replicates**

**perm\_replicates = draw\_perm\_reps(force\_a, force\_b,**

**diff\_of\_means, size=10000)**

**# Compute p-value: p**

**p = np.sum(perm\_replicates >= empirical\_diff\_means) / len(perm\_replicates)**

**# Print the result**

**print('p-value =', p)**

**p-value = 6.9827**

**<script.py> output:**

**p-value = 6.9825**

**p = np.sum(perm\_replicates >= empirical\_diff\_means) / len(perm\_replicates)**

**# Print the result**

**print('p-value =', p)**

**p-value = 6.9827**

**<script.py> output:**

**p-value = 0.0063**

The p-value tells you that there is about a 0.6% chance that you would get the difference of means observed in the experiment if frogs were exactly the same. A p-value below 0.01 is typically said to be "statistically significant," but: warning! warning! warning! You have computed a p-value; it is a number. I encourage you not to distill it to a yes-or-no phrase. p = 0.006 and p = 0.000000006 are both said to be "statistically significant," but they are definitely not the same!

# Bootstrap hypothesis tests

**50 XP**

## 1. Bootstrap hypothesis tests

Let's go over the pipeline of hypothesis testing that we have been studying.

## 2. Pipeline for hypothesis testing

First, clearly state the null hypothesis. Stating the null hypothesis so that it is crystal clear is essential to be able to simulate it. Next, define your test statistic. Then generate many sets of simulated data assuming the null hypothesis is true. Compute the test statistic for each simulated data set. The p-value is then the fraction of your simulated data sets for which the test statistic is at least as extreme as for the real data. Now let's do another hypothesis test.

## 3. Michelson and Newcomb: speed of light pioneers

Consider again Michelson's measurements of the speed of light. Around the same time that Michelson did his experiment, his future collaborator Simon Newcomb also measured the speed of light.

1. 1 Michelson image: public domain, Smithsonian
2. 2 Newcomb image: US Library of Congress

## 4. Michelson and Newcomb: speed of light pioneers

Newcomb's measurements had a mean of 299,860 km/s, differing from Michelson's by about 8 km/s. We want to know if there is something fundamentally different about Newcomb's and Michelson's measurements. The thing is: we only have Newcomb's mean and none of his data points!

1. 1 Michelson image: public domain, Smithsonian
2. 2 Newcomb image: US Library of Congress

## 5. The data we have

The question is: could Michelson have gotten the data set he did if the true mean speed of light in his experiments was equal to Newcomb's? So, we formally

1. 1 Data: Michelson, 1880

## 6. Null hypothesis

state our hypothesis as this: the true mean speed of light in Michelson's experiments was actually Newcomb's reported mean, which we'll call the Newcomb value. When I say the true mean speed of light in Michelson's experiments, think the mean Michelson would have gotten had done his experiment lots and lots and lots of times. Because we are comparing a data set with a value, a permutation test is not applicable. We need to simulate the situation in which the true mean speed of light in Michelson's experiments is Newcomb's value.

## 7. Shifting the Michelson data

To achieve this, we shift Michelson's data such that its mean now matches Newcomb's value. See here the ECDF of the shifted data relative to the original data. We can then use bootstrapping on this shifted data to simulate data acquisition under the null hypothesis.

## 8. Calculating the test statistic

The test statistic is the the mean of the bootstrap sample minus Newcomb's value. We write a function diff\_from\_newcomb to compute it, and compute the observed test statistic.

## 9. Computing the p-value

We then use the draw\_bs\_reps function you have already written to generate the bootstrap replicates, which is the value of the test statistic computed from a bootstrap sample. Note that the data we pass into the function are the shifted Michelson measurements because those are the ones we use to simulate the null hypothesis. The p-value is computed exactly the same way as for the permutation test. We report the fraction of bootstrap replicates that are less than the observed test statistic. In this case, we use less than because the mean from Michelson's experiments was less than Newcomb's value. We get a p-value of 0-point-16. This suggests that it is quite possible the Newcomb and Michelson did not really have fundamental differences in their measurements. This is an example of

## 10. One sample test

a one-sample test, since we had one set of samples from Michelson and were comparing to a single number from Newcomb. Often in the field, you will do two-sample tests that require the bootstrap, which you will explore in the exercises. I know this video was a lot to take in. Explicitly simulating a null hypothesis like this, we we have to shift the mean, can be tricky. You'll get a chance to practice in the exercises, and you may want to go over the procedure again to make sure you understand it.

## 11. Let's practice!

Ok, enough talk. Let's do some bootstrap hypothesis tests!

# A one-sample bootstrap hypothesis test

Another juvenile frog was studied, Frog C, and you want to see if Frog B and Frog C have similar impact forces. Unfortunately, you do not have Frog C's impact forces available, but you know they have a mean of 0.55 N. Because you don't have the original data, you cannot do a permutation test, and you cannot assess the hypothesis that the forces from Frog B and Frog C come from the same distribution. You will therefore test another, less restrictive hypothesis: The mean strike force of Frog B is equal to that of Frog C.

To set up the bootstrap hypothesis test, you will take the mean as our test statistic. Remember, your goal is to calculate the probability of getting a mean impact force less than or equal to what was observed for Frog B if the hypothesis that the true mean of Frog B's impact forces is equal to that of Frog C is true. You first translate all of the data of Frog B such that the mean is 0.55 N. This involves adding the mean force of Frog C and subtracting the mean force of Frog B from each measurement of Frog B. This leaves other properties of Frog B's distribution, such as the variance, unchanged.

##### Instructions

**100 XP**

* Translate the impact forces of Frog B such that its mean is 0.55 N.
* Use your draw\_bs\_reps() function to take 10,000 bootstrap replicates of the mean of your translated forces.
* Compute the p-value by finding the fraction of your bootstrap replicates that are less than the observed mean impact force of Frog B. Note that the variable of interest here is force\_b.
* Print your p-value.
* # Make an array of translated impact forces: translated\_force\_b
* translated\_force\_b = \_\_\_\_
* # Take bootstrap replicates of Frog B's translated impact forces: bs\_replicates
* bs\_replicates = draw\_bs\_reps(\_\_\_\_, \_\_\_\_, 10000)
* # Compute fraction of replicates that are less than the observed Frog B force: p
* p = np.sum(\_\_\_\_ <= np.mean(\_\_\_\_)) / 10000
* # Print the p-value
* print('p = ', \_\_\_\_)

# Make an array of translated impact forces: translated\_force\_b

translated\_force\_b = force\_b +0.55 - np.mean(force\_b)

# Take bootstrap replicates of Frog B's translated impact forces: bs\_replicates

bs\_replicates = draw\_bs\_reps(translated\_force\_b, np.mean, 10000)

# Compute fraction of replicates that are less than the observed Frog B force: p

p = np.sum(bs\_replicates <= np.mean(force\_b)) / 10000

# Print the p-value

print('p = ', p)

**# Make an array of translated impact forces: translated\_force\_b**

**translated\_force\_b = force\_b +0.55 - np.mean(force\_b)**

**# Take bootstrap replicates of Frog B's translated impact forces: bs\_replicates**

**bs\_replicates = draw\_bs\_reps(translated\_force\_b, np.mean, 10000)**

**# Compute fraction of replicates that are less than the observed Frog B force: p**

**p = np.sum(bs\_replicates <= np.mean(force\_b)) / 10000**

**# Print the p-value**

**print('p = ', p)**

**p = 0.0059**

**<script.py> output:**

**p = 0.0046**

Great work! The low p-value suggests that the null hypothesis that Frog B and Frog C have the same mean impact force is false.

# A two-sample bootstrap hypothesis test for difference of means

We now want to test the hypothesis that Frog A and Frog B have the same mean impact force, but not necessarily the same distribution, which is also impossible with a permutation test.

To do the two-sample bootstrap test, we shift both arrays to have the same mean, since we are simulating the hypothesis that their means are, in fact, equal. We then draw bootstrap samples out of the shifted arrays and compute the difference in means. This constitutes a bootstrap replicate, and we generate many of them. The p-value is the fraction of replicates with a difference in means greater than or equal to what was observed.

The objects forces\_concat and empirical\_diff\_means are already in your namespace.

##### Instructions

* Compute the mean of all forces (from forces\_concat) using np.mean().
* Generate shifted data sets for both force\_a and force\_b such that the mean of each is the mean of the concatenated array of impact forces.
* Generate 10,000 bootstrap replicates of the mean each for the two shifted arrays.
* Compute the bootstrap replicates of the difference of means by subtracting the replicates of the shifted impact force of Frog B from those of Frog A.
* Compute and print the p-value from your bootstrap replicates.
* # Compute mean of all forces: mean\_force
* mean\_force = \_\_\_\_
* # Generate shifted arrays
* force\_a\_shifted = force\_a - np.mean(force\_a) + mean\_force
* force\_b\_shifted = \_\_\_\_
* # Compute 10,000 bootstrap replicates from shifted arrays
* bs\_replicates\_a = draw\_bs\_reps(\_\_\_\_, \_\_\_\_, \_\_\_\_)
* bs\_replicates\_b = draw\_bs\_reps(\_\_\_\_, \_\_\_\_, \_\_\_\_)
* # Get replicates of difference of means: bs\_replicates
* bs\_replicates = \_\_\_\_
* # Compute and print p-value: p
* p = \_\_\_\_ / \_\_\_\_
* print('p-value =', p)

# Compute mean of all forces: mean\_force

mean\_force = np.mean(forces\_concat)

# Generate shifted arrays

force\_a\_shifted = force\_a - np.mean(force\_a) + mean\_force

force\_b\_shifted = force\_b - np.mean(force\_b) + mean\_force

# Compute 10,000 bootstrap replicates from shifted arrays

bs\_replicates\_a = draw\_bs\_reps(force\_a\_shifted, np.mean, 10000)

bs\_replicates\_b = draw\_bs\_reps(force\_b\_shifted, np.mean, 10000)

# Get replicates of difference of means: bs\_replicates

bs\_replicates = bs\_replicates\_a - bs\_replicates\_b

# Compute and print p-value: p

p = np.sum(bs\_replicates >= empirical\_diff\_means) / 10000

print('p-value =', p)

**# Compute mean of all forces: mean\_force**

**mean\_force = np.mean(forces\_concat)**

**# Generate shifted arrays**

**force\_a\_shifted = force\_a - np.mean(force\_a) + mean\_force**

**force\_b\_shifted = force\_b - np.mean(force\_b) + mean\_force**

**# Compute 10,000 bootstrap replicates from shifted arrays**

**bs\_replicates\_a = draw\_bs\_reps(force\_a\_shifted, np.mean, 10000)**

**bs\_replicates\_b = draw\_bs\_reps(force\_b\_shifted, np.mean, 10000)**

**# Get replicates of difference of means: bs\_replicates**

**bs\_replicates = bs\_replicates\_a - bs\_replicates\_b**

**# Compute and print p-value: p**

**p = np.sum(bs\_replicates >= empirical\_diff\_means) / 10000**

**print('p-value =', p)**

**p-value = 0.0046**

**<script.py> output:**

**p-value = 0.0043**

Nice work! You got a similar result as when you did the permutation test. Nonetheless, remember that it is important to carefully think about what question you want to ask. Are you only interested in the mean impact force, or in the distribution of impact forces?

# A/B testing

**50 XP**

## 1. A/B testing

Imagine your company has a proposed redesign of the splash page of its website. They are interested in how many more users click through to the website for the redesign versus the original design. You devise a test.

## 2. Is your redesign effective?

Take a set of 1000 visitors to the site and direct 500 of them to the original splash page and 500 of them to the redesigned one. You determine whether or not each of them clicks through to the rest of the website. On the original page, which we'll call page A,

## 3. Is your redesign effective?

45 visitors clicked through, and on the redesigned page, page B, 67 visitors clicked through. This makes you happy because that is almost a 50% increase in the click-through rate. But maybe there really is no difference between the effect of two designs on click-through rate and the difference you saw is due the random chance. You want to check: what is the probability that you would observe at least the observed difference in number of clicks through if that were the case? This is asking exactly the question you can address with

## 4. Null hypothesis

a hypothesis test. A permutation test is a good choice here because you can simulate the result as if the redesign had no effect on the click-through rate. Let's code it up in Python:

## 5. Permutation test of clicks through

for each splash page design, we have a NumPy array which contains 1 or 0 values for whether or not a visitor clicked through. Next, we need to define a function diff\_frac for our test statistic. Ours is the fraction of visitors who click through. We can compute the fraction who click through by summing the entries in the arrays of ones and zeros and then dividing by the number of entries. Finally we compute the observed value of the test statistic using this function diff\_frac. Now everything is in place to generate our permutation replicates of the test statistic

## 6. Permutation test of clicks through

using the permutation\_replicate function you wrote in the exercises; we will generate 10,000. We compute the p-value as the number of replicates where the test statistic was at least as great as what we observed. We get a value of 0-point-016, which is relatively small, so we might reasonably think that the redesign is a real improvement. This is an example of an A/B test.

## 7. A/B test

A/B testing is often used by organizations to see if a change in strategy gives different, hopefully better, results. Generally,

## 8. Null hypothesis of an A/B test

the null hypothesis in an A/B test is that your test statistic is impervious to the change. A low p-value implies that the change in strategy lead to a change in performance. Once again, though, be warned that statistical significance does not mean practical significance. A difference in click-though rate may be statistically significant, but if it is only a couple people more per day, your marketing team may not consider the change worth the cost! A/B testing is just a special case of the hypothesis testing framework we have already been working with, a fun and informative one.

## 9. Let's practice!

Let's practice in with some exercises!

# The vote for the Civil Rights Act in 1964

The Civil Rights Act of 1964 was one of the most important pieces of legislation ever passed in the USA. Excluding "present" and "abstain" votes, 153 House Democrats and 136 Republicans voted yea. However, 91 Democrats and 35 Republicans voted nay. Did party affiliation make a difference in the vote?

To answer this question, you will evaluate the hypothesis that the party of a House member has no bearing on his or her vote. You will use the fraction of Democrats voting in favor as your test statistic and evaluate the probability of observing a fraction of Democrats voting in favor at least as small as the observed fraction of 153/244. (That's right, at least as small as. In 1964, it was the Democrats who were less progressive on civil rights issues.) To do this, permute the party labels of the House voters and then arbitrarily divide them into "Democrats" and "Republicans" and compute the fraction of Democrats voting yea.

##### Instructions

**100 XP**

* Construct Boolean arrays, dems and reps that contain the votes of the respective parties; e.g., dems has 153 True entries and 91 False entries.
* Write a function, frac\_yea\_dems(dems, reps) that returns the fraction of Democrats that voted yea. The first input is an array of Booleans, Two inputs are required to use your draw\_perm\_reps() function, but the second is not used.
* Use your draw\_perm\_reps() function to draw 10,000 permutation replicates of the fraction of Democrat yea votes.
* Compute and print the p-value.
* # Construct arrays of data: dems, reps
* dems = np.array([True] \* 153 + [False] \* 91)
* reps = \_\_\_\_
* def frac\_yea\_dems(dems, reps):
* """Compute fraction of Democrat yea votes."""
* frac = \_\_\_\_ / \_\_\_\_
* return frac
* # Acquire permutation samples: perm\_replicates
* perm\_replicates = \_\_\_\_(\_\_\_\_, \_\_\_\_, frac\_yea\_dems, \_\_\_\_)
* # Compute and print p-value: p
* p = np.sum(\_\_\_\_ <= 153/244) / len(\_\_\_\_)
* print('p-value =', p)
* print('p-value =', p)
* # Construct arrays of data: dems, reps
* dems = np.array([True] \* 153 + [False] \* 91)
* reps = np.array([True] \* 136 + [False] \* 35)
* def frac\_yea\_dems(dems, reps):
* """Compute fraction of Democrat yea votes."""
* frac = np.sum(dems) / 244
* return frac
* # Acquire permutation samples: perm\_replicates
* perm\_replicates = draw\_perm\_reps(dems, reps, frac\_yea\_dems, size=10000)
* # Compute and print p-value: p
* p = np.sum(perm\_replicates <= 153/244) / len(perm\_replicates)
* print('p-value =', p)

# Construct arrays of data: dems, reps

dems = np.array([True] \* 153 + [False] \* 91)

reps = np.array([True] \* 136 + [False] \* 35)

def frac\_yea\_dems(dems, reps):

    """Compute fraction of Democrat yea votes."""

    frac = np.sum(dems) / 244

    return frac

# Acquire permutation samples: perm\_replicates

perm\_replicates = draw\_perm\_reps(dems, reps, frac\_yea\_dems, size=10000)

# Compute and print p-value: p

p = np.sum(perm\_replicates <= 153/244) / len(perm\_replicates)

print('p-value =', p)

<script.py> output: p-value = 0.0002

Great work! This small p-value suggests that party identity had a lot to do with the voting. Importantly, the South had a higher fraction of Democrat representatives, and consequently also a more racist bias.

# What is equivalent?

You have experience matching stories to probability distributions. Similarly, you use the same procedure for two different A/B tests if their stories match. In the Civil Rights Act example you just did, you performed an A/B test on voting data, which has a Yes/No type of outcome for each subject (in that case, a voter). Which of the following situations involving testing by a web-based company has an equivalent set up for an A/B test as the one you just did with the Civil Rights Act of 1964?

##### Answer the question

#### Possible Answers

* You measure how much time each customer spends on your website before and after an advertising campaign.
* **You measure the number of people who click on an ad on your company's website before and after changing its color.**
* You measure how many clicks each person has on your company's website before and after changing the header layout.

Yup! The "Democrats" are those who view the ad before the color change, and the "Republicans" are those who view it after.

# A time-on-website analog

It turns out that you already did a hypothesis test analogous to an A/B test where you are interested in how much time is spent on the website before and after an ad campaign. The frog tongue force (a continuous quantity like time on the website) is an analog. "Before" = Frog A and "after" = Frog B. Let's practice this again with something that actually is a before/after scenario.

We return to the no-hitter data set. In 1920, Major League Baseball implemented important rule changes that ended the so-called dead ball era. Importantly, the pitcher was no longer allowed to spit on or scuff the ball, an activity that greatly favors pitchers. In this problem you will perform an A/B test to determine if these rule changes resulted in a slower rate of no-hitters (i.e., longer average time between no-hitters) using the difference in mean inter-no-hitter time as your test statistic. The inter-no-hitter times for the respective eras are stored in the arrays nht\_dead and nht\_live, where "nht" is meant to stand for "no-hitter time."

Since you will be using your draw\_perm\_reps() function in this exercise, it may be useful to remind yourself of its call signature: draw\_perm\_reps(d1, d2, func, size=1) or even [referring back](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-2/introduction-to-hypothesis-testing?ex=7) to the chapter 3 exercise in which you defined it.

##### Instructions

* Compute the observed difference in mean inter-nohitter time using diff\_of\_means().
* Generate 10,000 permutation replicates of the difference of means using draw\_perm\_reps().
* Compute and print the p-value.
* # Compute the observed difference in mean inter-no-hitter times: nht\_diff\_obs
* nht\_diff\_obs = \_\_\_\_
* # Acquire 10,000 permutation replicates of difference in mean no-hitter time: perm\_replicates
* perm\_replicates = \_\_\_\_
* # Compute and print the p-value: p
* p = \_\_\_\_
* print('p-val =', p)

# Compute the observed difference in mean inter-no-hitter times: nht\_diff\_obs

nht\_diff\_obs = diff\_of\_means(np.mean(nht\_dead), np.mean(nht\_live))

# Acquire 10,000 permutation replicates of difference in mean no-hitter time: perm\_replicates

perm\_replicates = draw\_perm\_reps(nht\_dead, nht\_live, diff\_of\_means, 10000)

# Compute and print the p-value: p

p = np.sum(perm\_replicates <= nht\_diff\_obs)/ len(perm\_replicates)

print('p-val =', p)

**# Compute the observed difference in mean inter-no-hitter times: nht\_diff\_obs**

**nht\_diff\_obs = diff\_of\_means(np.mean(nht\_dead), np.mean(nht\_live))**

**# Acquire 10,000 permutation replicates of difference in mean no-hitter time: perm\_replicates**

**perm\_replicates = draw\_perm\_reps(nht\_dead, nht\_live, diff\_of\_means, 10000)**

**# Compute and print the p-value: p**

**p = np.sum(perm\_replicates <= nht\_diff\_obs)/ len(perm\_replicates)**

**print('p-val =', p)**

**p-val = 0.0**

**<script.py> output:**

**p-val = 0.0001**

Your p-value is 0.0001, which means that only one out of your 10,000 replicates had a result as extreme as the actual difference between the dead ball and live ball eras. This suggests strong statistical significance. Watch out, though, you could very well have gotten zero replicates that were as extreme as the observed value. This just means that the p-value is quite small, almost certainly smaller than 0.001.

# What should you have done first?

That was a nice hypothesis test you just did to check out whether the rule changes in 1920 changed the rate of no-hitters. But what should you have done with the data first?

#### Possible Answers

* **Performed EDA, perhaps plotting the ECDFs of inter-no-hitter times in the dead ball and live ball eras.**
* Nothing. The hypothesis test was only a few lines of code.

Yes! Always a good idea to do first! I encourage you to go ahead and plot the ECDFs right now. You will see by eye that the null hypothesis that the distributions are the same is almost certainly not true.

# Test of correlation

## 1. Test of correlation

Consider again the swing state county-level voting data.

## 2. 2008 US swing state election results

In the prequel to this course, we computed the Pearson correlation coefficient between Obama's vote share and the total number of votes. Remember the Pearson correlation coefficient is a measure of how much of the variability in two variables is due to them being correlated. It ranges from -1 for totally negatively correlated to 1 for positively correlated. We got a value of about point-54. This value of the Pearson correlation indicates that the data are not perfectly correlated, but are correlated nonetheless. But how can we know for sure if this correlation is real, or if it could have happened just by chance?

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. Hypothesis test of correlation

We can do a hypothesis test! We posit a null hypothesis that there is no correlation between the two variables, in this case Obama's vote share and total votes. We then simulate the election assuming the null hypothesis is true (which you will figure out how to do in the exercises), and use the Pearson correlation coefficient as the test statistic. The p-value is then the fraction of replicates that have a Pearson correlation coefficient at least as large as what was observed. I did this procedure, and in all 10,000 of my replicates under the null hypothesis,

## 4. More populous counties voted for Obama

not one had a Pearson correlation coefficient as high as the observed value of point-54. I tried generating 100,000, and then a million replicates. In all cases, not one replicate had a Pearson correlation coefficient as high as point-54. This does not mean that the p-value is zero. It means that it is so low that we would have to generate an enormous number of replicates to have even one that has a test statistic sufficiently extreme. We conclude that the p-value

## 5. More populous counties voted for Obama

is very very small and there is essentially no doubt that counties with higher vote count tended to vote for Obama. After all, that is how he won the election.

## 6. Let's practice!

Now it is your turn to think about how to do a hypothesis test on correlation and execute it!

# Simulating a null hypothesis concerning correlation

The observed correlation between female illiteracy and fertility in the data set of 162 countries may just be by chance; the fertility of a given country may actually be totally independent of its illiteracy. You will test this null hypothesis in the next exercise.

To do the test, you need to simulate the data assuming the null hypothesis is true. Of the following choices, which is the best way to do it?

#### Possible Answers

* Choose 162 random numbers to represent the illiteracy rate and 162 random numbers to represent the corresponding fertility rate.
* Do a pairs bootstrap: Sample pairs of observed data with replacement to generate a new set of (illiteracy, fertility) data.
* Do a bootstrap sampling in which you sample 162 illiteracy values with replacement and then 162 fertility values with replacement.
* **Do a permutation test: Permute the illiteracy values but leave the fertility values fixed to generate a new set of (illiteracy, fertility) data.**
* Do a permutation test: Permute both the illiteracy and fertility values to generate a new set of (illiteracy, fertility data).

Yes, this exactly simulates the null hypothesis and does so more efficiently than the last option. It is exact because it uses all data and eliminates any correlation because which illiteracy value pairs to which fertility value is shuffled.

# Hypothesis test on Pearson correlation

The observed correlation between female illiteracy and fertility may just be by chance; the fertility of a given country may actually be totally independent of its illiteracy. You will test this hypothesis. To do so, permute the illiteracy values but leave the fertility values fixed. This simulates the hypothesis that they are totally independent of each other. For each permutation, compute the Pearson correlation coefficient and assess how many of your permutation replicates have a Pearson correlation coefficient greater than the observed one.

The function pearson\_r() that you [wrote in the prequel to this course](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-1/quantitative-exploratory-data-analysis?ex=15) for computing the Pearson correlation coefficient is already available for you.

##### Instructions

* Compute the observed Pearson correlation between illiteracy and fertility.
* Initialize an array to store your permutation replicates.
* Write a for loop to draw 10,000 replicates:
  + Permute the illiteracy measurements using np.random.permutation().
  + Compute the Pearson correlation between the permuted illiteracy array, illiteracy\_permuted, and fertility.
* Compute and print the p-value from the replicates.
* # Compute observed correlation: r\_obs
* r\_obs = \_\_\_\_
* # Initialize permutation replicates: perm\_replicates
* perm\_replicates = np.empty(10000)
* # Draw replicates
* for \_\_\_\_ in \_\_\_\_:
* # Permute illiteracy measurments: illiteracy\_permuted
* illiteracy\_permuted = \_\_\_\_
* # Compute Pearson correlation
* perm\_replicates[i] = \_\_\_\_
* # Compute p-value: p
* p = \_\_\_\_
* print('p-val =', p)

# Compute observed correlation: r\_obs

r\_obs = pearson\_r(illiteracy, fertility)

# Initialize permutation replicates: perm\_replicates

perm\_replicates = np.empty(10000)

# Draw replicates

for i  in range(10000):

    # Permute illiteracy measurments: illiteracy\_permuted

    illiteracy\_permuted = np.random.permutation(illiteracy)

    # Compute Pearson correlation

    perm\_replicates[i] = pearson\_r(illiteracy\_permuted, fertility)

# Compute p-value: p

p = np.sum(perm\_replicates>=r\_obs)/10000

print('p-val =', p)

**# Compute observed correlation: r\_obs**

**r\_obs = pearson\_r(illiteracy, fertility)**

**# Initialize permutation replicates: perm\_replicates**

**perm\_replicates = np.empty(10000)**

**# Draw replicates**

**for i in range(10000):**

**# Permute illiteracy measurments: illiteracy\_permuted**

**illiteracy\_permuted = np.random.permutation(illiteracy)**

**# Compute Pearson correlation**

**perm\_replicates[i] = pearson\_r(illiteracy\_permuted, fertility)**

**# Compute p-value: p**

**p = np.sum(perm\_replicates>=r\_obs)/10000**

**print('p-val =', p)**

**p-val = 0.0**

**<script.py> output:**

**p-val = 0.0**

**You got a p-value of zero. In hacker statistics, this means that your p-value is very low, since you never got a single replicate in the 10,000 you took that had a Pearson correlation greater than the observed one. You could try increasing the number of replicates you take to continue to move the upper bound on your p-value lower and lower.**

# Do neonicotinoid insecticides have unintended consequences?

As a final exercise in hypothesis testing before we put everything together in our case study in the next chapter, you will investigate the effects of neonicotinoid insecticides on bee reproduction. These insecticides are very widely used in the United States to combat aphids and other pests that damage plants.

In a recent study, Straub, et al. ([Proc. Roy. Soc. B, 2016](http://dx.doi.org/10.1098/rspb.2016.0506)) investigated the effects of neonicotinoids on the sperm of pollinating bees. In this and the next exercise, you will study how the pesticide treatment affected the count of live sperm per half milliliter of semen.

First, we will do EDA, as usual. Plot ECDFs of the alive sperm count for untreated bees (stored in the NumPy array control) and bees treated with pesticide (stored in the NumPy array treated).

##### Instructions

* Use your ecdf() function to generate x,y values from the control and treated arrays for plotting the ECDFs.
* Plot the ECDFs on the same plot.
* The margins have been set for you, along with the legend and axis labels. Hit submit to see the result!
* # Compute x,y values for ECDFs
* x\_control, y\_control = \_\_\_\_
* x\_treated, y\_treated = \_\_\_\_
* # Plot the ECDFs
* plt.plot(\_\_\_\_, \_\_\_\_, marker='.', linestyle='none')
* plt.plot(\_\_\_\_, \_\_\_\_, marker='.', linestyle='none')
* # Set the margins
* plt.margins(0.02)
* # Add a legend
* plt.legend(('control', 'treated'), loc='lower right')
* # Label axes and show plot
* plt.xlabel('millions of alive sperm per mL')
* plt.ylabel('ECDF')
* plt.show()

# Compute x,y values for ECDFs

x\_control, y\_control = ecdf(control)

x\_treated, y\_treated = ecdf(treated)

# Plot the ECDFs

plt.plot(x\_control, y\_control, marker='.', linestyle='none')

plt.plot(x\_treated, y\_treated, marker='.', linestyle='none')

# Set the margins

plt.margins(0.02)

# Add a legend

plt.legend(('control', 'treated'), loc='lower right')

# Label axes and show plot

plt.xlabel('millions of alive sperm per mL')

plt.ylabel('ECDF')

plt.show()

**# Compute x,y values for ECDFs**

**x\_control, y\_control = ecdf(control)**

**x\_treated, y\_treated = ecdf(treated)**

**# Plot the ECDFs**

**plt.plot(x\_control, y\_control, marker='.', linestyle='none')**

**plt.plot(x\_treated, y\_treated, marker='.', linestyle='none')**

**# Set the margins**

**plt.margins(0.02)**

**# Add a legend**

**plt.legend(('control', 'treated'), loc='lower right')**

**# Label axes and show plot**

**plt.xlabel('millions of alive sperm per mL')**

**plt.ylabel('ECDF')**

**plt.show()**

Nice plot! The ECDFs show a pretty clear difference between the treatment and control; treated bees have fewer alive sperm. Let's now do a hypothesis test in the next exercise.

# Bootstrap hypothesis test on bee sperm counts

Now, you will test the following hypothesis: On average, male bees treated with neonicotinoid insecticide have the same number of active sperm per milliliter of semen than do untreated male bees. You will use the difference of means as your test statistic.

For your reference, the call signature for the draw\_bs\_reps() function [you wrote in chapter 2](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-2/bootstrap-confidence-intervals?ex=6) is draw\_bs\_reps(data, func, size=1).

##### Instructions

* Compute the mean alive sperm count of control minus that of treated.
* Compute the mean of all alive sperm counts. To do this, first concatenate control and treated and take the mean of the concatenated array.
* Generate shifted data sets for both control and treated such that the shifted data sets have the same mean. This has already been done for you.
* Generate 10,000 bootstrap replicates of the mean each for the two shifted arrays. Use your draw\_bs\_reps() function.
* Compute the bootstrap replicates of the difference of means.
* The code to compute and print the p-value has been written for you. Hit submit to see the result!
* # Compute the difference in mean sperm count: diff\_means
* diff\_means = \_\_\_\_
* # Compute mean of pooled data: mean\_count
* mean\_count = \_\_\_\_
* # Generate shifted data sets
* control\_shifted = control - np.mean(control) + mean\_count
* treated\_shifted = treated - np.mean(treated) + mean\_count
* # Generate bootstrap replicates
* bs\_reps\_control = \_\_\_\_(\_\_\_\_,
* np.mean, size=10000)
* bs\_reps\_treated = \_\_\_\_(\_\_\_\_,
* np.mean, size=10000)
* # Get replicates of difference of means: bs\_replicates
* bs\_replicates = \_\_\_\_
* # Compute and print p-value: p
* p = np.sum(bs\_replicates >= np.mean(control) - np.mean(treated)) \
* / len(bs\_replicates)
* print('p-value =', p)

# Compute the difference in mean sperm count: diff\_means

diff\_means = np.mean(control) - np.mean(treated)

# Compute mean of pooled data: mean\_count

mean\_count = np.mean(np.concatenate((control, treated)))

# Generate shifted data sets

control\_shifted = control - np.mean(control) + mean\_count

treated\_shifted = treated - np.mean(treated) + mean\_count

# Generate bootstrap replicates

bs\_reps\_control = draw\_bs\_reps(control\_shifted,

                       np.mean, size=10000)

bs\_reps\_treated = draw\_bs\_reps(treated\_shifted,

                       np.mean, size=10000)

# Get replicates of difference of means: bs\_replicates

bs\_replicates = bs\_reps\_control -bs\_reps\_treated

# Compute and print p-value: p

p = np.sum(bs\_replicates >= np.mean(control) - np.mean(treated)) \

            / len(bs\_replicates)

print('p-value =', p)

**# Compute the difference in mean sperm count: diff\_means**

**diff\_means = np.mean(control) - np.mean(treated)**

**# Compute mean of pooled data: mean\_count**

**mean\_count = np.mean(np.concatenate((control, treated)))**

**# Generate shifted data sets**

**control\_shifted = control - np.mean(control) + mean\_count**

**treated\_shifted = treated - np.mean(treated) + mean\_count**

**# Generate bootstrap replicates**

**bs\_reps\_control = draw\_bs\_reps(control\_shifted,**

**np.mean, size=10000)**

**bs\_reps\_treated = draw\_bs\_reps(treated\_shifted,**

**np.mean, size=10000)**

**# Get replicates of difference of means: bs\_replicates**

**bs\_replicates = bs\_reps\_control -bs\_reps\_treated**

**# Compute and print p-value: p**

**p = np.sum(bs\_replicates >= np.mean(control) - np.mean(treated)) \**

**/ len(bs\_replicates)**

**print('p-value =', p)**

**p-value = 0.0**

**<script.py> output:**

**p-value = 0.0**

Nice work! The p-value is small, most likely less than 0.0001, since you never saw a bootstrap replicated with a difference of means at least as extreme as what was observed. In fact, when I did the calculation with 10 million replicates, I got a p-value of 2e-05.

**Daily XP750**

# Finch beaks and the need for statistics

**50 XP**

## 1. Finch beaks and the need for statistics

You are now carrying a well-stocked toolbox.

## 2. Your well-equipped toolbox

You know how to take a data set and explore it. You know how to think probabilistically and infer values and confidence intervals for parameters. And you know how to formulate a hypothesis and use your data, and Python, to test it. Let's put those tools to use! In this chapter, you will immerse yourself in what is in my opinion one of the most fascinating data sets out there. You may know that many of the important observations that

## 3. Image: Public domain, US

led Charles Darwin to develop the theory of evolution were made

1. 1 Image: Public domain, US

## 4. Image: NASA

in the Galápagos archipelago, particularly in the study of the small birds, called finches, that inhabit them. The islands are ideal for studying evolution because they are isolated so they do not have complicated effects from interactions with other species including humans. Furthermore, some of them are small, so entire populations can be monitored on a given island. Every year since 1973, Peter and Rosemary Grant of Princeton University have been spending several months of the year on

1. 1 Image: NASA

## 5. The island of Daphne Major

the tiny volcanic cinder cone island of Daphne Major in the Galápagos.

1. 1 Image: Grant and Grant, 2014

## 6. The finches of Daphne Major

This island has two dominant ground finch species, Geospiza fortis and Geospiza scandens. The Grants have monitored them every year, tagging them, making physiological measurements, taking samples for genetic sequencing, and more. In 2014,

1. 1 Source: John Gould, public domain

## 7. Our data source

they published a book entitled "40 Years of Evolution: Darwin's Finches on Daphne Major Island". They generously placed all of their data on the Dryad data repository making it free for anyone to use. The data set is impressive and a great set for using data science and statistical inference to learn about evolution. You will work with the Grants' measurements of

## 8. The dimensions of the finch beak

the beak length and

1. 1 Source: John Gould, public domain

## 9. The dimensions of the finch beak

beak depth. You will consider different aspects of the beak geometry, including how it varies over time, from species to species, and from parents to offspring. I will come back with short discussions as you are working through the data sets, but you are mostly on your own. You can do it! Ok, for your first analysis,

1. 1 Source: John Gould, public domain

## 10. Investigation of G. scandens beak depth

you will investigate how the beak depth of Geospiza scandens has changed over time. You will start with some exploratory analysis of the measurements of these species from 1975 and 2012. You will then perform a parameter estimation, with confidence intervals, of mean beak depth for those respective years. Finally, you will do a hypothesis test investigating if the mean beak depth has changed from 1975 to 2012.

## 11. Let's practice!

Have at it!

# EDA of beak depths of Darwin's finches

For your first foray into the Darwin finch data, you will study how the beak depth (the distance, top to bottom, of a closed beak) of the finch species Geospiza scandens has changed over time. The Grants have noticed some changes of beak geometry depending on the types of seeds available on the island, and they also noticed that there was some interbreeding with another major species on Daphne Major, Geospiza fortis. These effects can lead to changes in the species over time.

In the next few problems, you will look at the beak depth of G. scandens on Daphne Major in 1975 and in 2012. To start with, let's plot all of the beak depth measurements in 1975 and 2012 in a bee swarm plot.

The data are stored in a pandas DataFrame called df with columns 'year' and 'beak\_depth'. The units of beak depth are millimeters (mm).

##### Instructions

**100 XP**

* Create the beeswarm plot.
* Label the axes.
* Show the plot.
* # Create bee swarm plot
* \_ = \_\_\_\_
* # Label the axes
* \_ = plt.\_\_\_\_('year')
* \_ = plt.\_\_\_\_('beak depth (mm)')
* # Show the plot
* \_\_\_\_

# Create bee swarm plot

\_ = sns.swarmplot(x='year', y='beak\_depth', data=df)

# Label the axes

\_ = plt.xlabel('year')

\_ = plt.ylabel('beak depth (mm)')

# Show the plot

plt.show()

# Create bee swarm plot \_ = sns.swarmplot(x='year', y='beak\_depth', data=df) # Label the axes \_ = plt.xlabel('year') \_ = plt.ylabel('beak depth (mm)') # Show the plot plt.show()

It is kind of hard to see if there is a clear difference between the 1975 and 2012 data set. Eyeballing it, it appears as though the mean of the 2012 data set might be slightly higher, and it might have a bigger variance.

# ECDFs of beak depths

While bee swarm plots are useful, we found that ECDFs are often even better when doing EDA. Plot the ECDFs for the 1975 and 2012 beak depth measurements on the same plot.

For your convenience, the beak depths for the respective years has been stored in the NumPy arrays bd\_1975 and bd\_2012.

##### Instructions

* Compute the ECDF for the 1975 and 2012 data.
* Plot the two ECDFs.
* Set a 2% margin and add axis labels and a legend to the plot.
* Hit submit to view the plot!
* # Compute ECDFs
* x\_1975, y\_1975 = \_\_\_\_
* x\_2012, y\_2012 = \_\_\_\_
* # Plot the ECDFs
* \_ = plt.\_\_\_\_(\_\_\_\_, \_\_\_\_, marker='.', linestyle='none')
* \_ = plt.\_\_\_\_(\_\_\_\_, \_\_\_\_, marker='.', linestyle='none')
* # Set margins
* \_\_\_\_
* # Add axis labels and legend
* \_ = plt.\_\_\_\_('beak depth (mm)')
* \_ = plt.\_\_\_\_('ECDF')
* \_ = plt.\_\_\_\_(('1975', '2012'), loc='lower right')
* # Show the plot
* plt.show()
* # Compute ECDFs
* x\_1975, y\_1975 = ecdf(bd\_1975)
* x\_2012, y\_2012 = ecdf(bd\_2012)
* # Plot the ECDFs
* \_ = plt.plot(x\_1975, y\_1975, marker='.', linestyle='none')
* \_ = plt.plot(x\_2012, y\_2012, marker='.', linestyle='none')
* # Set margins
* plt.margins(0.02)
* # Add axis labels and legend
* \_ = plt.xlabel('beak depth (mm)')
* \_ = plt.ylabel('ECDF')
* \_ = plt.legend(('1975', '2012'), loc='lower right')
* # Show the plot
* plt.show()

# Compute ECDFs x\_1975, y\_1975 = ecdf(bd\_1975) x\_2012, y\_2012 = ecdf(bd\_2012) # Plot the ECDFs \_ = plt.plot(x\_1975, y\_1975, marker='.', linestyle='none') \_ = plt.plot(x\_2012, y\_2012, marker='.', linestyle='none') # Set margins plt.margins(0.02) # Add axis labels and legend \_ = plt.xlabel('beak depth (mm)') \_ = plt.ylabel('ECDF') \_ = plt.legend(('1975', '2012'), loc='lower right') # Show the plot plt.show()

The differences are much clearer in the ECDF. The mean is larger in the 2012 data, and the variance does appear larger as well.

# Parameter estimates of beak depths

Estimate the difference of the mean beak depth of the G. scandens samples from 1975 and 2012 and report a 95% confidence interval.

Since in this exercise you will use the draw\_bs\_reps() function you wrote in chapter 2, it may be helpful to [refer back to it](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-2/bootstrap-confidence-intervals?ex=6).

##### Instructions

* Compute the difference of the sample means.
* Take 10,000 bootstrap replicates of the mean for the 1975 beak depths using your draw\_bs\_reps() function. Also get 10,000 bootstrap replicates of the mean for the 2012 beak depths.
* Subtract the 1975 replicates from the 2012 replicates to get bootstrap replicates of the difference of means.
* Use the replicates to compute the 95% confidence interval.
* Hit submit to view the results!
* # Compute the difference of the sample means: mean\_diff
* mean\_diff = \_\_\_\_
* # Get bootstrap replicates of means
* bs\_replicates\_1975 = \_\_\_\_
* bs\_replicates\_2012 = \_\_\_\_
* # Compute samples of difference of means: bs\_diff\_replicates
* bs\_diff\_replicates = \_\_\_\_
* # Compute 95% confidence interval: conf\_int
* conf\_int = \_\_\_\_
* # Print the results
* print('difference of means =', mean\_diff, 'mm')
* print('95% confidence interval =', conf\_int, 'mm')

# Compute the difference of the sample means: mean\_diff

mean\_diff = np.mean(bd\_2012) - np.mean(bd\_1975)

# Get bootstrap replicates of means

bs\_replicates\_1975 = draw\_bs\_reps(bd\_1975, np.mean, 10000)

bs\_replicates\_2012 = draw\_bs\_reps(bd\_2012, np.mean, 10000)

# Compute samples of difference of means: bs\_diff\_replicates

bs\_diff\_replicates = bs\_replicates\_2012 - bs\_replicates\_1975

# Compute 95% confidence interval: conf\_int

conf\_int = np.percentile(bs\_diff\_replicates, [2.5, 97.5])

# Print the results

print('difference of means =', mean\_diff, 'mm')

print('95% confidence interval =', conf\_int, 'mm')

Great work!

**# Compute the difference of the sample means: mean\_diff**

**mean\_diff = np.mean(bd\_2012) - np.mean(bd\_1975)**

**# Get bootstrap replicates of means**

**bs\_replicates\_1975 = draw\_bs\_reps(bd\_1975, np.mean, 10000)**

**bs\_replicates\_2012 = draw\_bs\_reps(bd\_2012, np.mean, 10000)**

**# Compute samples of difference of means: bs\_diff\_replicates**

**bs\_diff\_replicates = bs\_replicates\_2012 - bs\_replicates\_1975**

**# Compute 95% confidence interval: conf\_int**

**conf\_int = np.percentile(bs\_diff\_replicates, [2.5, 97.5])**

**# Print the results**

**print('difference of means =', mean\_diff, 'mm')**

**print('95% confidence interval =', conf\_int, 'mm')**

**difference of means = 0.22622047244094645 mm**

**95% confidence interval = [0.05940465 0.39410616] mm**

**<script.py> output:**

**difference of means = 0.22622047244094645 mm**

**95% confidence interval = [0.05633521 0.39190544] mm**

# Hypothesis test: Are beaks deeper in 2012?

Your plot of the ECDF and determination of the confidence interval make it pretty clear that the beaks of G. scandens on Daphne Major have gotten deeper. But is it possible that this effect is just due to random chance? In other words, what is the probability that we would get the observed difference in mean beak depth if the means were the same?

Be careful! The hypothesis we are testing is not that the beak depths come from the same distribution. For that we could use a permutation test. The hypothesis is that the means are equal. To perform this hypothesis test, we need to shift the two data sets so that they have the same mean and then use bootstrap sampling to compute the difference of means.

##### Instructions

* Make a concatenated array of the 1975 and 2012 beak depths and compute and store its mean.
* Shift bd\_1975 and bd\_2012 such that their means are equal to the one you just computed for the combined data set.
* Take 10,000 bootstrap replicates of the mean each for the 1975 and 2012 beak depths.
* Subtract the 1975 replicates from the 2012 replicates to get bootstrap replicates of the difference.
* Compute and print the p-value. The observed difference in means you computed in the last exercise is still in your namespace as mean\_diff.
* # Compute mean of combined data set: combined\_mean
* combined\_mean = \_\_\_\_(\_\_\_\_((bd\_1975, bd\_2012)))
* # Shift the samples
* bd\_1975\_shifted = \_\_\_\_
* bd\_2012\_shifted = \_\_\_\_
* # Get bootstrap replicates of shifted data sets
* bs\_replicates\_1975 = \_\_\_\_
* bs\_replicates\_2012 = \_\_\_\_
* # Compute replicates of difference of means: bs\_diff\_replicates
* bs\_diff\_replicates = \_\_\_\_
* # Compute the p-value
* p = np.sum(\_\_\_\_ >= \_\_\_\_) / len(\_\_\_\_)
* # Print p-value
* print('p =', p)

# Compute mean of combined data set: combined\_mean

combined\_mean = np.mean(np.concatenate((bd\_1975, bd\_2012)))

# Shift the samples

bd\_1975\_shifted = bd\_1975 - np.mean(bd\_1975) + combined\_mean

bd\_2012\_shifted = bd\_2012 - np.mean(bd\_2012) + combined\_mean

# Get bootstrap replicates of shifted data sets

bs\_replicates\_1975 = draw\_bs\_reps(bd\_1975\_shifted, np.mean, size=10000)

bs\_replicates\_2012 = draw\_bs\_reps(bd\_2012\_shifted, np.mean, size=10000)

# Compute replicates of difference of means: bs\_diff\_replicates

bs\_diff\_replicates = bs\_replicates\_2012 - bs\_replicates\_1975

# Compute the p-value

p = np.sum(bs\_diff\_replicates >= mean\_diff) / len(bs\_diff\_replicates)

# Print p-value

print('p =', p)

**# Get bootstrap replicates of shifted data sets**

**bs\_replicates\_1975 = draw\_bs\_reps(bd\_1975\_shifted, np.mean, size=10000)**

**bs\_replicates\_2012 = draw\_bs\_reps(bd\_2012\_shifted, np.mean, size=10000)**

**# Compute replicates of difference of means: bs\_diff\_replicates**

**bs\_diff\_replicates = bs\_replicates\_2012 - bs\_replicates\_1975**

**# Compute the p-value**

**p = np.sum(bs\_diff\_replicates >= mean\_diff) / len(bs\_diff\_replicates)**

**# Print p-value**

**print('p =', p)**

**p = 0.003**

**<script.py> output:**

**p = 0.0034**

We get a p-value of 0.0034, which suggests that there is a statistically significant difference. But remember: it is very important to know how different they are! In the previous exercise, you got a difference of 0.2 mm between the means. You should combine this with the statistical significance. Changing by 0.2 mm in 37 years is substantial by evolutionary standards. If it kept changing at that rate, the beak depth would double in only 400 years.

# Variation in beak shapes

## 1. Variation in beak shapes

You just determined that the beak depth of Geospiza scandens changed over the course of 37 years. There are a few hypotheses as to why this is the case. One reason may be

## 2. The drought of winter 1976/1977

a drought in 1976 and 1977 that resulted in the death of the plants that produce small seeds on the island. The larger seeds required deeper beaks to crack them, so large-beaked birds survived and then reproduced. If this is the case, it stands to reason that the length of the beak might also change over time. Importantly, if the length and depth change at the same rate,

1. 1 Source: Grant and Grant, 2014

## 3. Beak geometry

the beak has the same shape; it just gets bigger. But if the beak length and beak depth change differently, the shape of the beak changes. In the next few exercises, you will investigate how beak length and depth change together. That means it's time for some linear regression! As a hint,

1. 1 Source: John Gould, public domain

## 4. Hint

the draw\_bs\_pairs\_linreg function you wrote will be helpful in computing confidence intervals this of your linear regression parameters.

## 5. Let's practice!

Now, off you go, back to your analysis! -

# EDA of beak length and depth

The beak length data are stored as bl\_1975 and bl\_2012, again with units of millimeters (mm). You still have the beak depth data stored in bd\_1975 and bd\_2012. Make scatter plots of beak depth (y-axis) versus beak length (x-axis) for the 1975 and 2012 specimens.

##### Instructions

* Make a scatter plot of the 1975 data. Use the color='blue' keyword argument. Also use an alpha=0.5 keyword argument to have transparency in case data points overlap.
* Do the same for the 2012 data, but use the color='red' keyword argument.
* Add a legend and label the axes.
* Show your plot.
* # Make scatter plot of 1975 data
* \_ = \_\_\_\_(\_\_\_\_, \_\_\_\_, marker='.',
* linestyle='None', \_\_\_\_, \_\_\_\_)
* # Make scatter plot of 2012 data
* \_ = \_\_\_\_(\_\_\_\_, \_\_\_\_, marker='.',
* linestyle='None', \_\_\_\_, \_\_\_\_)
* # Label axes and make legend
* \_ = plt.\_\_\_\_('beak length (mm)')
* \_ = plt.\_\_\_\_('beak depth (mm)')
* \_ = plt.\_\_\_\_(('1975', '2012'), loc='upper left')
* # Show the plot
* \_\_\_\_

# Make scatter plot of 1975 data

\_ = plt.plot(bl\_1975, bd\_1975, marker='.',

             linestyle='None', color='blue', alpha=0.5)

# Make scatter plot of 2012 data

\_ = plt.plot(bl\_2012, bd\_2012, marker='.',

            linestyle='None', color='red', alpha=0.5)

# Label axes and make legend

\_ = plt.xlabel('beak length (mm)')

\_ = plt.ylabel('beak depth (mm)')

\_ = plt.legend(('1975', '2012'), loc='upper left')

# Show the plot

plt.show()

# Make scatter plot of 1975 data \_ = plt.plot(bl\_1975, bd\_1975, marker='.', linestyle='None', color='blue', alpha=0.5) # Make scatter plot of 2012 data \_ = plt.plot(bl\_2012, bd\_2012, marker='.', linestyle='None', color='red', alpha=0.5) # Label axes and make legend \_ = plt.xlabel('beak length (mm)') \_ = plt.ylabel('beak depth (mm)') \_ = plt.legend(('1975', '2012'), loc='upper left') # Show the plot plt.show()

Great work! In looking at the plot, we see that beaks got deeper (the red points are higher up in the y-direction), but not really longer. If anything, they got a bit shorter, since the red dots are to the left of the blue dots. So, it does not look like the beaks kept the same shape; they became shorter and deeper.

# Linear regressions

Perform a linear regression for both the 1975 and 2012 data. Then, perform pairs bootstrap estimates for the regression parameters. Report 95% confidence intervals on the slope and intercept of the regression line.

You will use the draw\_bs\_pairs\_linreg() function you [wrote back in chapter 2](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-2/bootstrap-confidence-intervals?ex=12).

As a reminder, its call signature is draw\_bs\_pairs\_linreg(x, y, size=1), and it returns bs\_slope\_reps and bs\_intercept\_reps. The beak length data are stored as bl\_1975 and bl\_2012, and the beak depth data is stored in bd\_1975 and bd\_2012.

##### Instructions

* Compute the slope and intercept for both the 1975 and 2012 data sets.
* Obtain 1000 pairs bootstrap samples for the linear regressions using your draw\_bs\_pairs\_linreg() function.
* Compute 95% confidence intervals for the slopes and the intercepts.
* # Compute the linear regressions
* slope\_1975, intercept\_1975 = \_\_\_\_
* slope\_2012, intercept\_2012 = \_\_\_\_
* # Perform pairs bootstrap for the linear regressions
* bs\_slope\_reps\_1975, bs\_intercept\_reps\_1975 = \
* \_\_\_\_
* bs\_slope\_reps\_2012, bs\_intercept\_reps\_2012 = \
* \_\_\_\_
* # Compute confidence intervals of slopes
* slope\_conf\_int\_1975 = \_\_\_\_
* slope\_conf\_int\_2012 = \_\_\_\_
* intercept\_conf\_int\_1975 = \_\_\_\_
* intercept\_conf\_int\_2012 = \_\_\_\_
* # Print the results
* print('1975: slope =', slope\_1975,
* 'conf int =', slope\_conf\_int\_1975)
* print('1975: intercept =', intercept\_1975,
* 'conf int =', intercept\_conf\_int\_1975)
* print('2012: slope =', slope\_2012,
* 'conf int =', slope\_conf\_int\_2012)
* print('2012: intercept =', intercept\_2012,
* 'conf int =', intercept\_conf\_int\_2012)

# Compute the linear regressions

slope\_1975, intercept\_1975 = np.polyfit(bl\_1975,bd\_1975,1)

slope\_2012, intercept\_2012 = np.polyfit(bl\_2012, bd\_2012, 1)

# Perform pairs bootstrap for the linear regressions

bs\_slope\_reps\_1975, bs\_intercept\_reps\_1975 = draw\_bs\_pairs\_linreg(bl\_1975, bd\_1975, size=1000)

bs\_slope\_reps\_2012, bs\_intercept\_reps\_2012 = draw\_bs\_pairs\_linreg(bl\_2012, bd\_2012, size=1000)

# Compute confidence intervals of slopes

slope\_conf\_int\_1975 = np.percentile(bs\_slope\_reps\_1975, [2.5, 97.5])

slope\_conf\_int\_2012 = np.percentile(bs\_slope\_reps\_2012, [2.5, 97.5])

intercept\_conf\_int\_1975 = np.percentile(bs\_intercept\_reps\_1975, [2.5, 97.5])

intercept\_conf\_int\_2012 = np.percentile(bs\_intercept\_reps\_2012, [2.5, 97.5])

# Print the results

print('1975: slope =', slope\_1975,

      'conf int =', slope\_conf\_int\_1975)

print('1975: intercept =', intercept\_1975,

      'conf int =', intercept\_conf\_int\_1975)

print('2012: slope =', slope\_2012,

      'conf int =', slope\_conf\_int\_2012)

print('2012: intercept =', intercept\_2012,

      'conf int =', intercept\_conf\_int\_2012)

# Compute the linear regressions

slope\_1975, intercept\_1975 = np.polyfit(bl\_1975,bd\_1975,1)

slope\_2012, intercept\_2012 = np.polyfit(bl\_2012, bd\_2012, 1)

# Perform pairs bootstrap for the linear regressions

bs\_slope\_reps\_1975, bs\_intercept\_reps\_1975 = draw\_bs\_pairs\_linreg(bl\_1975, bd\_1975, size=1000)

bs\_slope\_reps\_2012, bs\_intercept\_reps\_2012 = draw\_bs\_pairs\_linreg(bl\_2012, bd\_2012, size=1000)

# Compute confidence intervals of slopes

slope\_conf\_int\_1975 = np.percentile(bs\_slope\_reps\_1975, [2.5, 97.5])

slope\_conf\_int\_2012 = np.percentile(bs\_slope\_reps\_2012, [2.5, 97.5])

intercept\_conf\_int\_1975 = np.percentile(bs\_intercept\_reps\_1975, [2.5, 97.5])

intercept\_conf\_int\_2012 = np.percentile(bs\_intercept\_reps\_2012, [2.5, 97.5])

# Print the results

print('1975: slope =', slope\_1975,

'conf int =', slope\_conf\_int\_1975)

print('1975: intercept =', intercept\_1975,

'conf int =', intercept\_conf\_int\_1975)

print('2012: slope =', slope\_2012,

'conf int =', slope\_conf\_int\_2012)

print('2012: intercept =', intercept\_2012,

'conf int =', intercept\_conf\_int\_2012)

1975: slope = 0.4652051691605937 conf int = [0.33851226 0.59306491]

1975: intercept = 2.3908752365842254 conf int = [0.64892945 4.18037063]

2012: slope = 0.4626303588353128 conf int = [0.33137479 0.60695527]

2012: intercept = 2.97724749823602 conf int = [1.06792753 4.70599387]

<script.py> output:

**1975: slope = 0.4652051691605937 conf int = [0.33851226 0.59306491]**

**1975: intercept = 2.3908752365842254 conf int = [0.64892945 4.18037063]**

**2012: slope = 0.4626303588353128 conf int = [0.33137479 0.60695527]**

**2012: intercept = 2.97724749823602 conf int = [1.06792753 4.70599387]**

**Nicely done! It looks like they have the same slope, but different intercepts.**

# Displaying the linear regression results

Now, you will display your linear regression results on the scatter plot, the code for which is already pre-written for you from your previous exercise. To do this, take the first 100 bootstrap samples (stored in bs\_slope\_reps\_1975, bs\_intercept\_reps\_1975, bs\_slope\_reps\_2012, and bs\_intercept\_reps\_2012) and plot the lines with alpha=0.2 and linewidth=0.5 keyword arguments to plt.plot().

##### Instructions

* Generate the x-values for the bootstrap lines using np.array(). They should consist of 10 mm and 17 mm.
* Write a for loop to plot 100 of the bootstrap lines for the 1975 and 2012 data sets. The lines for the 1975 data set should be 'blue' and those for the 2012 data set should be 'red'.
* Hit submit to view the plot!

# Make scatter plot of 1975 data

\_ = plt.plot(bl\_1975, bd\_1975, marker='.',

             linestyle='none', color='blue', alpha=0.5)

# Make scatter plot of 2012 data

\_ = plt.plot(bl\_2012, bd\_2012, marker='.',

             linestyle='none', color='red', alpha=0.5)

# Label axes and make legend

\_ = plt.xlabel('beak length (mm)')

\_ = plt.ylabel('beak depth (mm)')

\_ = plt.legend(('1975', '2012'), loc='upper left')

# Generate x-values for bootstrap lines: x

x = np.array([\_\_\_\_, \_\_\_\_])

# Plot the bootstrap lines

for i in range(100):

    plt.plot(\_\_\_\_, \_\_\_\_,

             linewidth=0.5, alpha=0.2, color=\_\_\_\_)

    plt.plot(\_\_\_\_, \_\_\_\_,

             linewidth=0.5, alpha=0.2, color=\_\_\_\_)

# Draw the plot again

plt.show()

**# Make scatter plot of 1975 data**

**\_ = plt.plot(bl\_1975, bd\_1975, marker='.',**

**linestyle='none', color='blue', alpha=0.5)**

**# Make scatter plot of 2012 data**

**\_ = plt.plot(bl\_2012, bd\_2012, marker='.',**

**linestyle='none', color='red', alpha=0.5)**

**# Label axes and make legend**

**\_ = plt.xlabel('beak length (mm)')**

**\_ = plt.ylabel('beak depth (mm)')**

**\_ = plt.legend(('1975', '2012'), loc='upper left')**

**# Generate x-values for bootstrap lines: x**

**x = np.array([10, 17])**

**# Plot the bootstrap lines**

**for i in range(100):**

**plt.plot(x,bs\_slope\_reps\_1975[i] \* x + bs\_intercept\_reps\_1975[i],**

**linewidth=0.5, alpha=0.2, color='blue')**

**plt.plot(x,bs\_slope\_reps\_2012[i] \* x + bs\_intercept\_reps\_2012[i],**

**linewidth=0.5, alpha=0.2, color='red')**

**# Draw the plot again**

**plt.show()**

# Make scatter plot of 1975 data \_ = plt.plot(bl\_1975, bd\_1975, marker='.', linestyle='none', color='blue', alpha=0.5) # Make scatter plot of 2012 data \_ = plt.plot(bl\_2012, bd\_2012, marker='.', linestyle='none', color='red', alpha=0.5) # Label axes and make legend \_ = plt.xlabel('beak length (mm)') \_ = plt.ylabel('beak depth (mm)') \_ = plt.legend(('1975', '2012'), loc='upper left') # Generate x-values for bootstrap lines: x x = np.array([10, 17]) # Plot the bootstrap lines for i in range(100): plt.plot(x,bs\_slope\_reps\_1975[i] \* x + bs\_intercept\_reps\_1975[i], linewidth=0.5, alpha=0.2, color='blue') plt.plot(x,bs\_slope\_reps\_2012[i] \* x + bs\_intercept\_reps\_2012[i], linewidth=0.5, alpha=0.2, color='red') # Draw the plot again plt.show()

Great work!

# Beak length to depth ratio

The linear regressions showed interesting information about the beak geometry. The slope was the same in 1975 and 2012, suggesting that for every millimeter gained in beak length, the birds gained about half a millimeter in depth in both years. However, if we are interested in the shape of the beak, we want to compare the ratio of beak length to beak depth. Let's make that comparison.

Remember, the data are stored in bd\_1975, bd\_2012, bl\_1975, and bl\_2012.

##### Instructions

* Make arrays of the beak length to depth ratio of each bird for 1975 and for 2012.
* Compute the mean of the length to depth ratio for 1975 and for 2012.
* Generate 10,000 bootstrap replicates each for the mean ratio for 1975 and 2012 using your draw\_bs\_reps() function.
* Get a **99%** bootstrap confidence interval for the length to depth ratio for 1975 and 2012.
* Print the results.
* # Compute length-to-depth ratios
* ratio\_1975 = \_\_\_\_
* ratio\_2012 = \_\_\_\_
* # Compute means
* mean\_ratio\_1975 = \_\_\_\_
* mean\_ratio\_2012 = \_\_\_\_
* # Generate bootstrap replicates of the means
* bs\_replicates\_1975 = \_\_\_\_
* bs\_replicates\_2012 = \_\_\_\_
* # Compute the 99% confidence intervals
* conf\_int\_1975 = \_\_\_\_
* conf\_int\_2012 = \_\_\_\_
* # Print the results
* print('1975: mean ratio =', mean\_ratio\_1975,
* 'conf int =', conf\_int\_1975)
* print('2012: mean ratio =', mean\_ratio\_2012,
* 'conf int =', conf\_int\_2012)

**# Compute length-to-depth ratios**

**ratio\_1975 = bl\_1975 / bd\_1975**

**ratio\_2012 = bl\_2012 / bd\_2012**

**# Compute means**

**mean\_ratio\_1975 = np.mean(ratio\_1975)**

**mean\_ratio\_2012 = np.mean(ratio\_2012)**

**# Generate bootstrap replicates of the means**

**bs\_replicates\_1975 = draw\_bs\_reps(ratio\_1975, np.mean, size=10000)**

**bs\_replicates\_2012 = draw\_bs\_reps(ratio\_2012, np.mean, size=10000)**

**# Compute the 99% confidence intervals**

**conf\_int\_1975 = np.percentile(bs\_replicates\_1975, [0.5,99.5])**

**conf\_int\_2012 = np.percentile(bs\_replicates\_2012, [0.5,99.5])**

**# Print the results**

**print('1975: mean ratio =', mean\_ratio\_1975,**

**'conf int =', conf\_int\_1975)**

**print('2012: mean ratio =', mean\_ratio\_2012,**

**'conf int =', conf\_int\_2012)**

**<script.py> output: 1975: mean ratio = 1.5788823771858533 conf int = [1.55668803 1.60073509]**

**2012: mean ratio = 1.4658342276847767 conf int = [1.44363932 1.48729149]**

# Compute length-to-depth ratios ratio\_1975 = bl\_1975 / bd\_1975 ratio\_2012 = bl\_2012 / bd\_2012 # Compute means mean\_ratio\_1975 = np.mean(ratio\_1975) mean\_ratio\_2012 = np.mean(ratio\_2012) # Generate bootstrap replicates of the means bs\_replicates\_1975 = draw\_bs\_reps(ratio\_1975, np.mean, size=10000) bs\_replicates\_2012 = draw\_bs\_reps(ratio\_2012, np.mean, size=10000) # Compute the 99% confidence intervals conf\_int\_1975 = np.percentile(bs\_replicates\_1975, [0.5,99.5]) conf\_int\_2012 = np.percentile(bs\_replicates\_2012, [0.5,99.5]) # Print the results print('1975: mean ratio =', mean\_ratio\_1975, 'conf int =', conf\_int\_1975) print('2012: mean ratio =', mean\_ratio\_2012, 'conf int =', conf\_int\_2012)

**How different is the ratio?**

In the previous exercise, you computed the mean beak length to depth ratio with 99% confidence intervals for 1975 and for 2012. The results of that calculation are shown graphically in the plot accompanying this problem. In addition to these results, what would you say about the ratio of beak length to depth?

**Possible Answers**

* **The mean beak length-to-depth ratio decreased by about 0.1, or 7%, from 1975 to 2012. The 99% confidence intervals are not even close to overlapping, so this is a real change. The beak shape changed.**
* It is impossible to say if this is a real effect or just due to noise without computing a p-value. Let me compute the p-value and get back to you.

**Daily XP1700**

# Calculation of heritability

**50 XP**

## 1. Calculation of heritability

What is causing the beaks of Geospiza scandens to get bigger over time? I mentioned the selective pressure brought on by the drought. But why do some birds have such large beaks to begin with? A prevailing explanation is that scandens birds are mating the other major finch species on Daphne Major, Geopsiza fortis.

## 2. The finches of Daphne Major

These hybrid birds then mate with pure scandens, in a process called introgressive hybridization, which can bring fortis characteristics into the scandens species. This is similar to what likely happened to humans when they encountered neanderthals. In order assess the viability of this explanation,

1. 1 Source: John Gould, public domain

## 3. Heredity

we need to know how strongly parental traits are passed on to offspring. In this last set of exercises, we will investigate the extent to which parental beak depth is inherited by offspring in both scandens and fortis. You are applying your new statistical skills to a real, fundamental scientific problem.

## 4. Let's practice!

I find this exhilarating, and I hope you do, too!

# EDA of heritability

The array bd\_parent\_scandens contains the average beak depth (in mm) of two parents of the species G. scandens. The array bd\_offspring\_scandens contains the average beak depth of the offspring of the respective parents. The arrays bd\_parent\_fortis and bd\_offspring\_fortis contain the same information about measurements from G. fortis birds.

Make a scatter plot of the average offspring beak depth (y-axis) versus average parental beak depth (x-axis) for both species. Use the alpha=0.5 keyword argument to help you see overlapping points.

##### Instructions

* Generate scatter plots for both species. Display the data for G. fortis in blue and G. scandens in red.
* Set the axis labels, make a legend, and show the plot.
* # Make scatter plots
* \_ = \_\_\_\_(\_\_\_\_, \_\_\_\_,
* marker=\_\_\_\_, linestyle=\_\_\_\_, color=\_\_\_\_, alpha=\_\_\_\_)
* \_ = plt.plot(bd\_parent\_scandens, \_\_\_\_,
* marker=\_\_\_\_, linestyle=\_\_\_\_, color=\_\_\_\_, alpha=\_\_\_\_)
* # Label axes
* \_ = plt.\_\_\_\_('parental beak depth (mm)')
* \_ = plt.\_\_\_\_('offspring beak depth (mm)')
* # Add legend
* \_ = plt.\_\_\_\_(('G. fortis', 'G. scandens'), loc='lower right')
* **# Show plot**
* **plt.show()**

**# Make scatter plots**

**\_ = plt.plot(bd\_parent\_fortis, bd\_offspring\_fortis,**

**marker='.', linestyle='none', color='blue', alpha=0.5)**

**\_ = plt.plot(bd\_parent\_scandens, bd\_offspring\_scandens,**

**marker='.', linestyle='none', color='red', alpha=0.5)**

**# Label axes**

**\_ = plt.xlabel('parental beak depth (mm)')**

**\_ = plt.ylabel('offspring beak depth (mm)')**

**# Add legend**

**\_ = plt.legend(('G. fortis', 'G. scandens'), loc='lower right')**

**# Show plot**

**plt.show()**

# Make scatter plots \_ = plt.plot(bd\_parent\_fortis, bd\_offspring\_fortis, marker='.', linestyle='none', color='blue', alpha=0.5) \_ = plt.plot(bd\_parent\_scandens, bd\_offspring\_scandens, marker='.', linestyle='none', color='red', alpha=0.5) # Label axes \_ = plt.xlabel('parental beak depth (mm)') \_ = plt.ylabel('offspring beak depth (mm)') # Add legend \_ = plt.legend(('G. fortis', 'G. scandens'), loc='lower right') # Show plot plt.show()

# Correlation of offspring and parental data

In an effort to quantify the correlation between offspring and parent beak depths, we would like to compute statistics, such as the Pearson correlation coefficient, between parents and offspring. To get confidence intervals on this, we need to do a pairs bootstrap.

You have [already written](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-2/bootstrap-confidence-intervals?ex=12) a function to do pairs bootstrap to get estimates for parameters derived from linear regression. Your task in this exercise is to make a new function with call signature draw\_bs\_pairs(x, y, func, size=1) that performs pairs bootstrap and computes a single statistic on pairs samples defined. The statistic of interest is computed by calling func(bs\_x, bs\_y). In the next exercise, you will use pearson\_r for func.

##### Instructions

* Set up an array of indices to sample from. (Remember, when doing pairs bootstrap, we randomly choose indices and use those to get the pairs.)
* Initialize the array of bootstrap replicates. This should be a one-dimensional array of length size.
* Write a for loop to draw the samples.
* Randomly choose indices from the array of indices you previously set up.
* Extract x values and y values from the input array using the indices you just chose to generate a bootstrap sample.
* Use func to compute the statistic of interest from the bootstrap samples of x and y and store it in your array of bootstrap replicates.
* Return the array of bootstrap replicates.
* def draw\_bs\_pairs(x, y, func, size=1):
* """Perform pairs bootstrap for a single statistic."""
* # Set up array of indices to sample from: inds
* inds = \_\_\_\_
* # Initialize replicates: bs\_replicates
* bs\_replicates = \_\_\_\_
* # Generate replicates
* for i in range(size):
* bs\_inds = \_\_\_\_
* bs\_x, bs\_y = \_\_\_\_
* bs\_replicates[i] = \_\_\_\_
* return bs\_replicates

def draw\_bs\_pairs(x, y, func, size=1):

    """Perform pairs bootstrap for a single statistic."""

    # Set up array of indices to sample from: inds

    inds = np.arange(len(x))

    # Initialize replicates: bs\_replicates

    bs\_replicates = np.empty(size)

    # Generate replicates

    for i in range(size):

        bs\_inds = np.random.choice(inds, size=len(inds))

        bs\_x, bs\_y = func(bs\_x,bs\_y)

        bs\_replicates[i] = np.polyfit(x,y)

    return bs\_replicates

def draw\_bs\_pairs(x, y, func, size=1): """Perform pairs bootstrap for a single statistic.""" # Set up array of indices to sample from: inds inds = np.arange(len(x)) # Initialize replicates: bs\_replicates bs\_replicates = np.empty(size) # Generate replicates for i in range(size): bs\_inds = np.random.choice(inds, size=len(inds)) bs\_x, bs\_y = func(bs\_x,bs\_y) bs\_replicates[i] = np.polyfit(x,y) return bs\_replicates

# Pearson correlation of offspring and parental data

The Pearson correlation coefficient seems like a useful measure of how strongly the beak depth of parents are inherited by their offspring. Compute the Pearson correlation coefficient between parental and offspring beak depths for G. scandens. Do the same for G. fortis. Then, use the function you wrote in the last exercise to compute a 95% confidence interval using pairs bootstrap.

Remember, the data are stored in bd\_parent\_scandens, bd\_offspring\_scandens, bd\_parent\_fortis, and bd\_offspring\_fortis.

##### Instructions

* Use the pearson\_r() function [you wrote in the prequel to this course](https://campus.datacamp.com/courses/statistical-thinking-in-python-part-1/quantitative-exploratory-data-analysis?ex=15) to compute the Pearson correlation coefficient for G. scandens and G. fortis.
* Acquire 1000 pairs bootstrap replicates of the Pearson correlation coefficient using the draw\_bs\_pairs() function you wrote in the previous exercise for G. scandens and G. fortis.
* Compute the 95% confidence interval for both using your bootstrap replicates.
* # Compute the Pearson correlation coefficients
* r\_scandens = \_\_\_\_
* r\_fortis = \_\_\_\_
* # Acquire 1000 bootstrap replicates of Pearson r
* bs\_replicates\_scandens = \_\_\_\_
* bs\_replicates\_fortis = \_\_\_\_
* # Compute 95% confidence intervals
* conf\_int\_scandens = \_\_\_\_
* conf\_int\_fortis = \_\_\_\_
* # Print results
* print('G. scandens:', r\_scandens, conf\_int\_scandens)
* print('G. fortis:', r\_fortis, conf\_int\_fortis)

# Compute the Pearson correlation coefficients

r\_scandens = pearson\_r(bd\_parent\_scandens, bd\_offspring\_scandens)

r\_fortis = pearson\_r(bd\_parent\_fortis, bd\_offspring\_fortis)

# Acquire 1000 bootstrap replicates of Pearson r

bs\_replicates\_scandens = draw\_bs\_pairs(bd\_parent\_scandens,bd\_offspring\_scandens,pearson\_r, 1000)

bs\_replicates\_fortis = draw\_bs\_pairs(bd\_parent\_fortis, bd\_offspring\_fortis,pearson\_r, 1000)

# Compute 95% confidence intervals

conf\_int\_scandens = np.percentile(bs\_replicates\_scandens,[2.5, 97.5])

conf\_int\_fortis = np.percentile(bs\_replicates\_fortis,[2.5, 97.5])

# Print results

print('G. scandens:', r\_scandens, conf\_int\_scandens)

print('G. fortis:', r\_fortis, conf\_int\_fortis)

**# Compute the Pearson correlation coefficients**

**r\_scandens = pearson\_r(bd\_parent\_scandens, bd\_offspring\_scandens)**

**r\_fortis = pearson\_r(bd\_parent\_fortis, bd\_offspring\_fortis)**

**# Acquire 1000 bootstrap replicates of Pearson r**

**bs\_replicates\_scandens = draw\_bs\_pairs(bd\_parent\_scandens,bd\_offspring\_scandens,pearson\_r, 1000)**

**bs\_replicates\_fortis = draw\_bs\_pairs(bd\_parent\_fortis, bd\_offspring\_fortis,pearson\_r, 1000)**

**# Compute 95% confidence intervals**

**conf\_int\_scandens = np.percentile(bs\_replicates\_scandens,[2.5, 97.5])**

**conf\_int\_fortis = np.percentile(bs\_replicates\_fortis,[2.5, 97.5])**

**# Print results**

**print('G. scandens:', r\_scandens, conf\_int\_scandens)**

**print('G. fortis:', r\_fortis, conf\_int\_fortis)**

**G. scandens: 0.41170636294012586 [0.26564228 0.54388972]**

**G. fortis: 0.7283412395518484 [0.66985843 0.77795471]**

**<script.py> output:**

**G. scandens: 0.41170636294012586 [0.26564228 0.54388972]**

**G. fortis: 0.7283412395518484 [0.6694112 0.77840616]**

**It is clear from the confidence intervals that beak depth of the offspring of G. fortis parents is more strongly correlated with their offspring than their G. scandens counterparts.**

# Measuring heritability

Remember that the Pearson correlation coefficient is the ratio of the covariance to the geometric mean of the variances of the two data sets. This is a measure of the correlation between parents and offspring, but might not be the best estimate of heritability. If we stop and think, it makes more sense to define heritability as the ratio of the covariance between parent and offspring to the variance of the parents alone. In this exercise, you will estimate the heritability and perform a pairs bootstrap calculation to get the 95% confidence interval.

This exercise highlights a very important point. Statistical inference (and data analysis in general) is not a plug-n-chug enterprise. You need to think carefully about the questions you are seeking to answer with your data and analyze them appropriately. If you are interested in how heritable traits are, the quantity we defined as the heritability is more apt than the off-the-shelf statistic, the Pearson correlation coefficient.

Remember, the data are stored in bd\_parent\_scandens, bd\_offspring\_scandens, bd\_parent\_fortis, and bd\_offspring\_fortis.

##### Instructions

* Write a function heritability(parents, offspring) that computes heritability defined as the ratio of the covariance of the trait in parents and offspring divided by the variance of the trait in the parents. Hint: Remind yourself of the np.cov() function we covered in the prequel to this course.
* Use this function to compute the heritability for G. scandens and G. fortis.
* Acquire 1000 bootstrap replicates of the heritability using pairs bootstrap for G. scandens and G. fortis.
* Compute the 95% confidence interval for both using your bootstrap replicates.
* Print the results.
* def heritability(parents, offspring):
* """Compute the heritability from parent and offspring samples."""
* covariance\_matrix = np.cov(parents, offspring)
* return \_\_\_\_ / \_\_\_\_
* # Compute the heritability
* heritability\_scandens = \_\_\_\_
* heritability\_fortis = \_\_\_\_
* # Acquire 1000 bootstrap replicates of heritability
* replicates\_scandens = draw\_bs\_pairs(
* \_\_\_\_, \_\_\_\_, \_\_\_\_, size=\_\_\_\_)
* replicates\_fortis = draw\_bs\_pairs(
* \_\_\_\_, \_\_\_\_, \_\_\_\_, size=\_\_\_\_)
* # Compute 95% confidence intervals
* conf\_int\_scandens = \_\_\_\_
* conf\_int\_fortis = \_\_\_\_
* # Print results
* print('G. scandens:', heritability\_scandens, conf\_int\_scandens)
* print('G. fortis:', heritability\_fortis, conf\_int\_fortis)

def heritability(parents, offspring):

    """Compute the heritability from parent and offspring samples."""

    covariance\_matrix = np.cov(parents, offspring)

    return covariance\_matrix[0,1] / covariance\_matrix[0,0]

# Compute the heritability

heritability\_scandens = heritability(bd\_parent\_scandens, bd\_offspring\_scandens)

heritability\_fortis = heritability(bd\_parent\_fortis, bd\_offspring\_fortis)

# Acquire 1000 bootstrap replicates of heritability

replicates\_scandens = draw\_bs\_pairs(bd\_parent\_scandens, bd\_offspring\_scandens, heritability, size=1000)

replicates\_fortis =  draw\_bs\_pairs(bd\_parent\_fortis,bd\_offspring\_fortis, heritability, size=1000)

# Compute 95% confidence intervals

conf\_int\_scandens = np.percentile(replicates\_scandens, [2.5, 97.5])

conf\_int\_fortis = np.percentile(replicates\_fortis, [2.5, 97.5])

# Print results

print('G. scandens:', heritability\_scandens, conf\_int\_scandens)

print('G. fortis:', heritability\_fortis, conf\_int\_fortis)

def heritability(parents, offspring):

"""Compute the heritability from parent and offspring samples."""

covariance\_matrix = np.cov(parents, offspring)

return covariance\_matrix[0,1] / covariance\_matrix[0,0]

# Compute the heritability

heritability\_scandens = heritability(bd\_parent\_scandens, bd\_offspring\_scandens)

heritability\_fortis = heritability(bd\_parent\_fortis, bd\_offspring\_fortis)

# Acquire 1000 bootstrap replicates of heritability

replicates\_scandens = draw\_bs\_pairs(bd\_parent\_scandens, bd\_offspring\_scandens, heritability, size=1000)

replicates\_fortis = draw\_bs\_pairs(bd\_parent\_fortis,bd\_offspring\_fortis, heritability, size=1000)

# Compute 95% confidence intervals

conf\_int\_scandens = np.percentile(replicates\_scandens, [2.5, 97.5])

conf\_int\_fortis = np.percentile(replicates\_fortis, [2.5, 97.5])

# Print results

print('G. scandens:', heritability\_scandens, conf\_int\_scandens)

print('G. fortis:', heritability\_fortis, conf\_int\_fortis)

G. scandens: 0.5485340868685982 [0.35738761 0.73844253]

G. fortis: 0.7229051911438156 [0.64669164 0.79504662]

<script.py> output:

G. scandens: 0.5485340868685982 [0.34395487 0.75638267]

G. fortis: 0.7229051911438156 [0.64655013 0.79688342]

**Here again, we see that G. fortis has stronger heritability than G. scandens. This suggests that the traits of G. fortis may be strongly incorporated into G. scandens by introgressive hybridization.**

# Is beak depth heritable at all in G. scandens?

The heritability of beak depth in G. scandens seems low. It could be that this observed heritability was just achieved by chance and beak depth is actually not really heritable in the species. You will test that hypothesis here. To do this, you will do a pairs permutation test.

##### Instructions

* Initialize your array of replicates of heritability. We will take 10,000 pairs permutation replicates.
* Write a for loop to generate your replicates.
  + Permute the bd\_parent\_scandens array using np.random.permutation().
  + Compute the heritability between the permuted array and the bd\_offspring\_scandens array using the heritability() function you wrote in the last exercise. Store the result in the replicates array.
* Compute the p-value as the number of replicates that are greater than the observed heritability\_scandens you computed in the last exercise.
* # Initialize array of replicates: perm\_replicates
* perm\_replicates = \_\_\_\_
* # Draw replicates
* for i in range(10000):
* # Permute parent beak depths
* bd\_parent\_permuted = \_\_\_\_
* perm\_replicates[i] = \_\_\_\_
* # Compute p-value: p
* p = np.sum(\_\_\_\_ >= \_\_\_\_) / len(\_\_\_\_)
* # Print the p-value
* print('p-val =', p)

# Initialize array of replicates: perm\_replicates

perm\_replicates = np.empty(10000)

# Draw replicates

for i in range(10000):

    # Permute parent beak depths

    bd\_parent\_permuted = np.random.permutation(bd\_parent\_scandens)

    perm\_replicates[i] = heritability(bd\_parent\_permuted, bd\_offspring\_scandens)

# Compute p-value: p

p = np.sum(perm\_replicates >= heritability\_scandens) / len(perm\_replicates)

# Print the p-value

print('p-val =', p)

**# Draw replicates**

**for i in range(10000):**

**# Permute parent beak depths**

**bd\_parent\_permuted = np.random.permutation(bd\_parent\_scandens)**

**perm\_replicates[i] = heritability(bd\_parent\_permuted, bd\_offspring\_scandens)**

**# Compute p-value: p**

**p = np.sum(perm\_replicates >= heritability\_scandens) / len(perm\_replicates)**

**# Print the p-value**

**print('p-val =', p)**

**p-val = 0.0**

**<script.py> output:**

**p-val = 0.0**

**You get a p-value of zero, which means that none of the 10,000 permutation pairs replicates you drew had a heritability high enough to match that which was observed. This strongly suggests that beak depth is heritable in G. scandens, just not as much as in G. fortis. If you like, you can plot a histogram of the heritability replicates to get a feel for how extreme of a value of heritability you might expect by chance.**

# Final thoughts

**50 XP**

## 1. Final thoughts

Wow, that finch beak analysis was intense. It makes me happy to know that you got to dive into one of my favorite data sets of all time. I hope going through that case study showed you how powerful you now are with your new skills in

## 2. Your statistical thinking skills

statistical inference, along with being able to implement them in Python. You can now do effective exploratory data analysis, plotting ECDFs and computing summary statistics. You can obtain parameters by optimization, including by linear regression. You can think probabilistically about those parameter values and give them confidence intervals. To do this, you can use the exceptionally powerful and versatile bootstrap method. Finally, you can formulate and test statistical hypotheses. There is a world of data out there for you to make inferences about using your hard-earnt Python skills! You have the knowledge and skills to do so. You are now a more powerful data scientist.

## 3. Let's practice!

Bon voyage!