Reference

DataCamp course

Defined functions

In [1]:
```python
def ecdf(data):
    """Compute ECDF for a one-dimensional array of measurements."""

    n = len(data)
    x = np.sort(data)
    y = np.arange(1, n+1) / n
    return x, y

def pearson_r(x, y):
    """Compute Pearson correlation coefficient between two arrays."""
    corr_mat = np.corrcoef(x,y)

    return corr_mat[0,1]

# Returns a replicate, a single value, but not an array of data.
def bootstrap_replicate_1d(data, func):

    return func(np.random.choice(data, size=len(data)))
    #options for .choice():
    #replacement = true/false; size choice; probability for choosing each element in data; data can be string arr

def draw_bs_reps(data, func, size=1):
    """Draw bootstrap replicates."""

    bs_replicates = np.empty(size)

    for i in range(size):
        bs_replicates[i] = bootstrap_replicate_1d(data,func)

    return bs_replicates

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(len(x))
    #Note it is not range() at all
    #from 0 to len(x)-1, because the boundary is not included in the right.

    # 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

# The key to draw bootstrap pairs is to create a random array of indices.
def draw_bs_pairs(x, y,func,size=1):
    """Perform pairs bootstrap for correlation calculation."""

    inds = np.arange(len(x))

    bs_replicates = np.empty(size)

    for i in range(size):
        bs_inds = np.random.choice(inds, len(inds))
        bs_x, bs_y = x[bs_inds], y[bs_inds]
        bs_replicates[i] = func(bs_x, bs_y)
        #For example, the func here can be the pearson_r(x, y), which returns a correlation coefficient of two ve

    return bs_replicates

def permutation_sample(data1, data2):
    """Generate a permutation sample from two data sets."""

    data = np.concatenate((data1,data2))
    #IMPORTANT. the argument of concatenate is a tuple.
    #np.concatenate(data1,data2) is wrong

    permuted_data = np.random.permutation(data)

    perm_sample_1 = permuted_data[:len(data1)]
    perm_sample_2 = permuted_data[len(data1):]

    return perm_sample_1, perm_sample_2

def draw_perm_reps(data_1, data_2, func, size=1):
    """Generate multiple permutation replicates."""

    perm_replicates = np.empty(size)
    for i in range(size):
        perm_sample_1, perm_sample_2 = permutation_sample(data_1, data_2)

        perm_replicates[i] = func(perm_sample_1, perm_sample_2)

    return perm_replicates
```

Parameter estimation by optimization

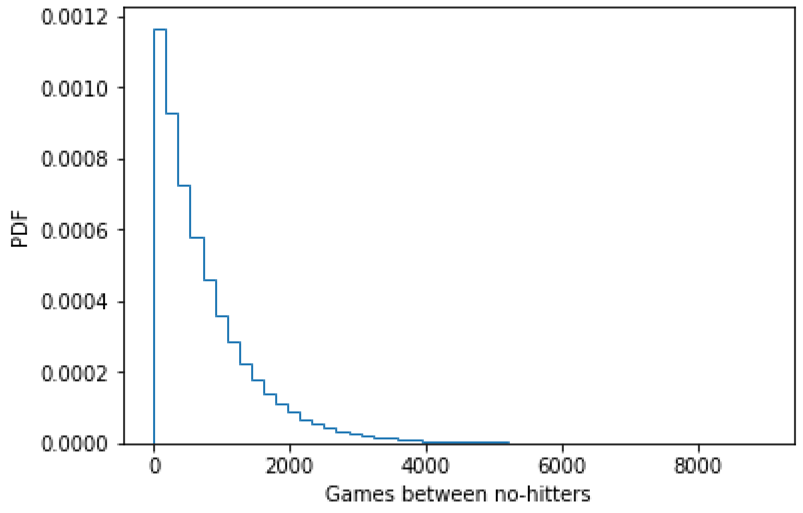
The whole point of most machine learning is parameter estimation. See notes on machine learning. The following is a simple introduction.

How often do we get no-hitters?

If we assume that no-hitters are described as a Poisson process, then the time between no-hitters is Exponentially distributed. The value of the parameter τ 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 calculated τ and plot the histogram as an approximation to the PDF.

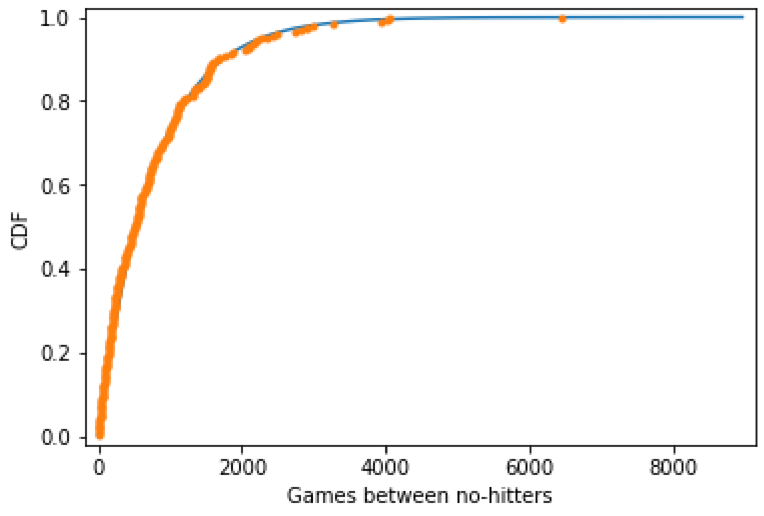
In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
nohitter_times = np.array([ 843, 1613, 1101,  215,  684,  814,  278,  324,  161,  219,  545,
        715,  966,  624,   29,  450,  107,   20,   91, 1325,  124, 1468,
        104, 1309,  429,   62, 1878, 1104,  123,  251,   93,  188,  983,
        166,   96,  702,   23,  524,   26,  299,   59,   39,   12,    2,
        308, 1114,  813,  887,  645, 2088,   42, 2090,   11,  886, 1665,
       1084, 2900, 2432,  750, 4021, 1070, 1765, 1322,   26,  548, 1525,
         77, 2181, 2752,  127, 2147,  211,   41, 1575,  151,  479,  697,
        557, 2267,  542,  392,   73,  603,  233,  255,  528,  397, 1529,
       1023, 1194,  462,  583,   37,  943,  996,  480, 1497,  717,  224,
        219, 1531,  498,   44,  288,  267,  600,   52,  269, 1086,  386,
        176, 2199,  216,   54,  675, 1243,  463,  650,  171,  327,  110,
        774,  509,    8,  197,  136,   12, 1124,   64,  380,  811,  232,
        192,  731,  715,  226,  605,  539, 1491,  323,  240,  179,  702,
        156,   82, 1397,  354,  778,  603, 1001,  385,  986,  203,  149,
        576,  445,  180, 1403,  252,  675, 1351, 2983, 1568,   45,  899,
       3260, 1025,   31,  100, 2055, 4043,   79,  238, 3931, 2351,  595,
        110,  215,    0,  563,  206,  660,  242,  577,  179,  157,  192,
        192, 1848,  792, 1693,   55,  388,  225, 1134, 1172, 1555,   31,
       1582, 1044,  378, 1687, 2915,  280,  765, 2819,  511, 1521,  745,
       2491,  580, 2072, 6450,  578,  745, 1075, 1103, 1549, 1520,  138,
       1202,  296,  277,  351,  391,  950,  459,   62, 1056, 1128,  139,
        420,   87,   71,  814,  603, 1349,  162, 1027,  783,  326,  101,
        876,  381,  905,  156,  419,  239,  119,  129,  467])
# Seed random number generator

np.random.seed(42)

tau = np.mean(nohitter_times)

# Draw out of an exponential distribution with parameter tau: inter_nohitter_time
inter_nohitter_time = np.random.exponential(tau, 100000)

# Plot the PDF and label axes
_ = plt.hist(inter_nohitter_time,
             density = 'normed', histtype = 'step', bins = 50)
_ = plt.xlabel('Games between no-hitters')
_ = plt.ylabel('PDF')

plt.show()
```



Do the data follow our story?

Create an ECDF of the real data. Overlay the theoretical CDF with the ECDF from the data. This helps verify that the Exponential distribution describes the observed data. How about directly using using PDF to compare?

```
In [76]: x, y = ecdf(nohitter_times)

         # Create a CDF from theoretical samples: x_theor, y_theor
         #tau = np.mean(nohitter_times)
         x_theor, y_theor = ecdf(inter_nohitter_time)

         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')

         plt.show()
```



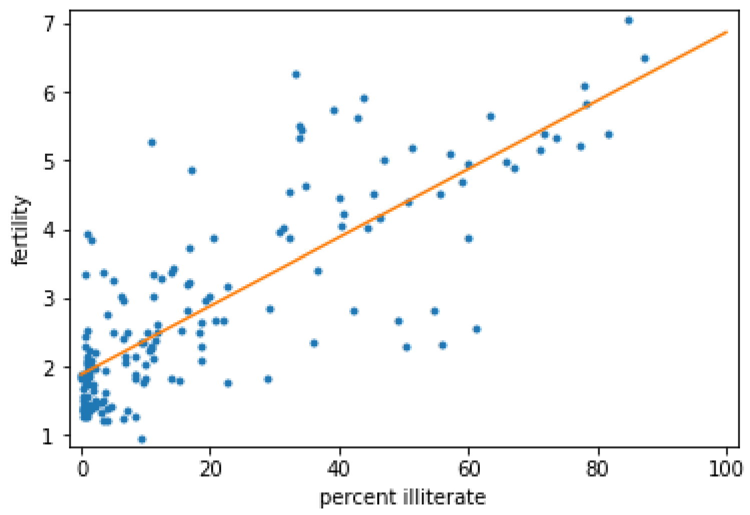
How is this parameter optimal?

When using different τ CDF really deviats a lot.

```
In [16]: illiteracy = np.array([ 9.5, 49.2,  1. , 11.2,  9.8, 60. , 50.2, 51.2,  0.6,  1. ,  8.5,
                 6.1,  9.8,  1. , 42.2, 77.2, 18.7, 22.8,  8.5, 43.9,  1. ,  1. ,
                 1.5, 10.8, 11.9,  3.4,  0.4,  3.1,  6.6, 33.7, 40.4,  2.3, 17.2,
                 0.7, 36.1,  1. , 33.2, 55.9, 30.8, 87.4, 15.4, 54.6,  5.1,  1.1,
                10.2, 19.8,  0. , 40.7, 57.2, 59.9,  3.1, 55.7, 22.8, 10.9, 34.7,
                32.2, 43. ,  1.3,  1. ,  0.5, 78.4, 34.2, 84.9, 29.1, 31.3, 18.3,
                81.8, 39. , 11.2, 67. ,  4.1,  0.2, 78.1,  1. ,  7.1,  1. , 29. ,
                 1.1, 11.7, 73.6, 33.9, 14. ,  0.3,  1. ,  0.8, 71.9, 40.1,  1. ,
                 2.1,  3.8, 16.5,  4.1,  0.5, 44.4, 46.3, 18.7,  6.5, 36.8, 18.6,
                11.1, 22.1, 71.1,  1. ,  0. ,  0.9,  0.7, 45.5,  8.4,  0. ,  3.8,
                 8.5,  2. ,  1. , 58.9,  0.3,  1. , 14. , 47. ,  4.1,  2.2,  7.2,
                 0.3,  1.5, 50.5,  1.3,  0.6, 19.1,  6.9,  9.2,  2.2,  0.2, 12.3,
                 4.9,  4.6,  0.3, 16.5, 65.7, 63.5, 16.8,  0.2,  1.8,  9.6, 15.2,
                14.4,  3.3, 10.6, 61.3, 10.9, 32.2,  9.3, 11.6, 20.7,  6.5,  6.7,
                 3.5,  1. ,  1.6, 20.5,  1.5, 16.7,  2. ,  0.9])
         fertility = np.array([1.769, 2.682, 2.077, 2.132, 1.827, 3.872, 2.288, 5.173, 1.393,
                1.262, 2.156, 3.026, 2.033, 1.324, 2.816, 5.211, 2.1  , 1.781,
                1.822, 5.908, 1.881, 1.852, 1.39 , 2.281, 2.505, 1.224, 1.361,
                1.468, 2.404, 5.52 , 4.058, 2.223, 4.859, 1.267, 2.342, 1.579,
                6.254, 2.334, 3.961, 6.505, 2.53 , 2.823, 2.498, 2.248, 2.508,
                3.04 , 1.854, 4.22 , 5.1  , 4.967, 1.325, 4.514, 3.173, 2.308,
                4.62 , 4.541, 5.637, 1.926, 1.747, 2.294, 5.841, 5.455, 7.069,
                2.859, 4.018, 2.513, 5.405, 5.737, 3.363, 4.89 , 1.385, 1.505,
                6.081, 1.784, 1.378, 1.45 , 1.841, 1.37 , 2.612, 5.329, 5.33 ,
                3.371, 1.281, 1.871, 2.153, 5.378, 4.45 , 1.46 , 1.436, 1.612,
                3.19 , 2.752, 3.35 , 4.01 , 4.166, 2.642, 2.977, 3.415, 2.295,
                3.019, 2.683, 5.165, 1.849, 1.836, 2.518, 2.43 , 4.528, 1.263,
                1.885, 1.943, 1.899, 1.442, 1.953, 4.697, 1.582, 2.025, 1.841,
                5.011, 1.212, 1.502, 2.516, 1.367, 2.089, 4.388, 1.854, 1.748,
                2.978, 2.152, 2.362, 1.988, 1.426, 3.29 , 3.264, 1.436, 1.393,
                2.822, 4.969, 5.659, 3.24 , 1.693, 1.647, 2.36 , 1.792, 3.45 ,
                1.516, 2.233, 2.563, 5.283, 3.885, 0.966, 2.373, 2.663, 1.251,
                2.052, 3.371, 2.093, 2.   , 3.883, 3.852, 3.718, 1.732, 3.928])
         # Linear regression
         # Plot the illiteracy rate versus fertility
         _ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')
         plt.margins(0.02)
         _ = plt.xlabel('percent illiterate')
         _ = plt.ylabel('fertility')

         a, b = np.polyfit(illiteracy,fertility,1) #Do not forget the degree 1 for linear
```

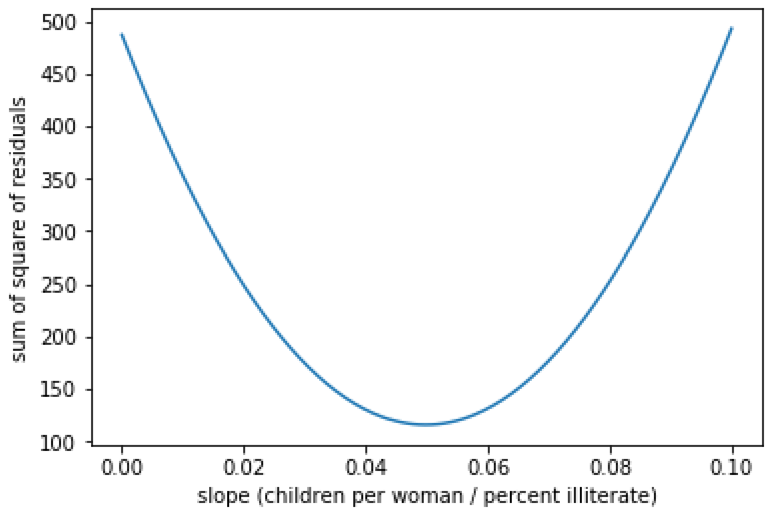
```
print('slope =', a, 'children per woman / percent illiterate')
print('intercept =', b, 'children per woman')

x = np.array([0,100])
y = a * x + b

_ = plt.plot(x, y)

plt.show()
```

```
slope = 0.04979854809063423 children per woman / percent illiterate
intercept = 1.888050610636557 children per woman
```



How is it optimal?

In [17]:
```
# 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)
#np.empty vs np.empty_like
#The empty_like() function returns a new array with the same shape and type
#as a given array (in this case, 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()
```



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, Compute the slope and intercept for each set.

In [18]:
```
x1 = [10.,  8., 13.,  9., 11., 14.,  6.,  4., 12.,  7.,  5.]
x2 = [10.,  8., 13.,  9., 11., 14.,  6.,  4., 12.,  7.,  5.]
x3 = [10.,  8., 13.,  9., 11., 14.,  6.,  4., 12.,  7.,  5.]
x4 = [ 8.,  8.,  8.,  8.,  8.,  8.,  8., 19.,  8.,  8.,  8.]
anscombe_x = [x1,x2,x3,x4]

y1 = [8.04,  6.95,  7.58,  8.81,  8.33,  9.96,  7.24,  4.26, 10.84, 4.82,  5.68]
y2 = [9.14, 8.14, 8.74, 8.77, 9.26, 8.1 , 6.13, 3.1 , 9.13, 7.26, 4.74]
y3 = [ 7.46,  6.77, 12.74,  7.11,  7.81,  8.84,  6.08,  5.39,  8.15, 6.42,  5.73]
y4 = [ 6.58,  5.76,  7.71,  8.84,  8.47,  7.04,  5.25, 12.5 ,  5.56, 7.91,  6.89]
anscombe_y = [y1,y2, y3, y4]

# 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: 0.5000909090909095 intercept: 3.0000909090909076
slope: 0.5000000000000004 intercept: 3.000909090909089
slope: 0.4997272727272731 intercept: 3.0024545454545453
slope: 0.49990909090909064 intercept: 3.0017272727272735
```

Do it another way.

In [19]:
```
for i in range(len(anscombe_x)):
    a,b = np.polyfit(anscombe_x[i],anscombe_y[i],1)
    # Print the result
    print('slope:', a, 'intercept:', b)
```

```
slope: 0.5000909090909095 intercept: 3.0000909090909076
slope: 0.5000000000000004 intercept: 3.000909090909089
slope: 0.4997272727272731 intercept: 3.0024545454545453
slope: 0.49990909090909064 intercept: 3.0017272727272735
```

Bootstrap confidence intervals

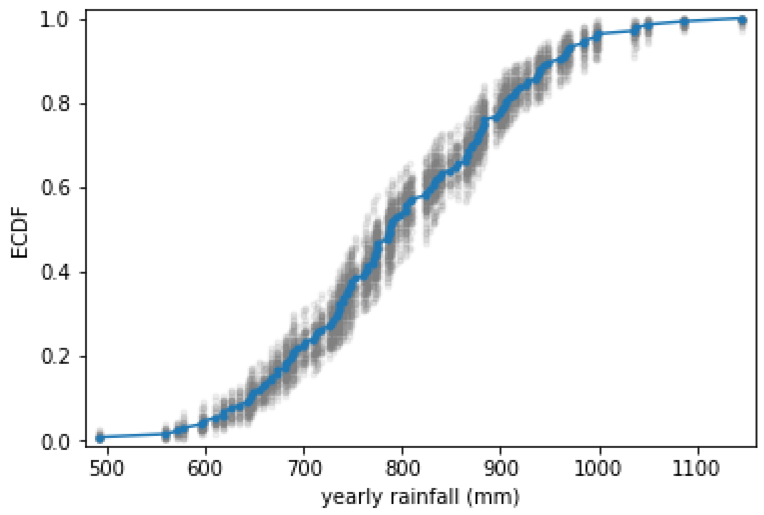
- Different terminology for bootstrap samples and replicates are often used. Here is what we means.
- If we have a data set with n repeated measurements, a bootstrap sample is an array of length n that was drawn from the original data with replacement.
- What is a bootstrap replicate? Answer: A single value of a statistic computed from a bootstrap sample.
- Bootstrapping is simulate to do many many experiments that we cannot do in practice

Visualizing bootstrap samples

- Generate bootstrap samples from a set of annual rainfall data.
- Displaying the bootstrap samples with an ECDF.

In [20]:

```
rainfall = np.array([ 875.5,  648.2,  788.1,  940.3,  491.1,  743.5,  730.1,  686.5,
        878.8,  865.6,  654.9,  831.5,  798.1,  681.8,  743.8,  689.1,
        752.1,  837.2,  710.6,  749.2,  967.1,  701.2,  619. ,  747.6,
        803.4,  645.6,  804.1,  787.4,  646.8,  997.1,  774. ,  734.5,
        835. ,  840.7,  659.6,  828.3,  909.7,  856.9,  578.3,  904.2,
        883.9,  740.1,  773.9,  741.4,  866.8,  871.1,  712.5,  919.2,
        927.9,  809.4,  633.8,  626.8,  871.3,  774.3,  898.8,  789.6,
        936.3,  765.4,  882.1,  681.1,  661.3,  847.9,  683.9,  985.7,
        771.1,  736.6,  713.2,  774.5,  937.7,  694.5,  598.2,  983.8,
        700.2,  901.3,  733.5,  964.4,  609.3, 1035.2,  718. ,  688.6,
        736.8,  643.3, 1038.5,  969. ,  802.7,  876.6,  944.7,  786.6,
        770.4,  808.6,  761.3,  774.2,  559.3,  674.2,  883.6,  823.9,
        960.4,  877.8,  940.6,  831.8,  906.2,  866.5,  674.1,  998.1,
        789.3,  915. ,  737.1,  763. ,  666.7,  824.5,  913.8,  905.1,
        667.8,  747.4,  784.7,  925.4,  880.2, 1086.9,  764.4, 1050.1,
        595.2,  855.2,  726.9,  785.2,  948.8,  970.6,  896. ,  618.4,
        572.4, 1146.4,  728.2,  864.2,  793. ])

%matplotlib inline
import matplotlib.pyplot as plt

for i in range(50):
    # Generate bootstrap sample: bs_sample
    bs_sample = np.random.choice(rainfall, size=len(rainfall))
    #Note the sample length is the same as the original
    #replace = True is the default.

    # Compute and plot ECDF from bootstrap sample
    x, y = ecdf(bs_sample)
    _ = plt.plot(x, y, marker='.', linestyle='none',
                 color='gray', alpha=0.1) #alpha = 0.1 for semi-transparent.

# Compute and plot ECDF from original data
x, y = ecdf(rainfall)
_ = plt.plot(x, y, marker='.')

plt.margins(0.02)
_ = plt.xlabel('yearly rainfall (mm)')
_ = plt.ylabel('ECDF')

plt.show()
```

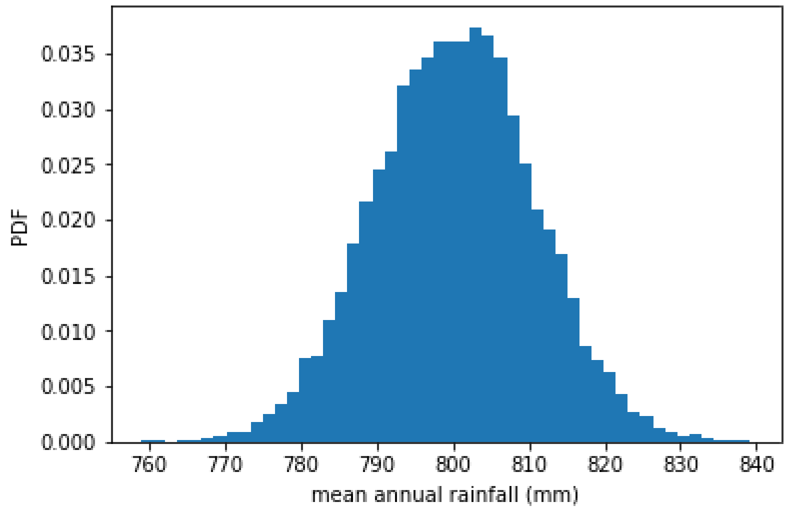


Generating many bootstrap replicates

- Bootstrap replicates of the mean and the SEM.
- 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.
- **As show below, theoretical SEM is very similar to the value obtained by the STD of sample mean**.
- Using hacker statistics, we get this same result without the need to derive it, but you will verify this result from your bootstrap replicates.

```
In [21]: # Take 10,000 bootstrap replicates of the mean: bs_replicates
bs_replicates = draw_bs_reps(rainfall, np.mean, 10000)
#np.mean is the 'func' parameter.

# 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)



_ = plt.hist(bs_replicates, bins=50, density = 'normed')
_ = plt.xlabel('mean annual rainfall (mm)')
_ = plt.ylabel('PDF')

plt.show()
```

```
10.510549150506188
10.466406332674087
```



Confidence intervals of rainfall data

- 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 just take percentiles of your bootstrap replicates to get your confidence interval. Conveniently, you can use the np.percentile() function.
- Compute the 95% confidence interval. That is, give the 2.5th and 97.5th percentile of your bootstrap replicates stored as bs_replicates.

In [22]:
```
np.percentile(bs_replicates, [2.5,97.5])
```

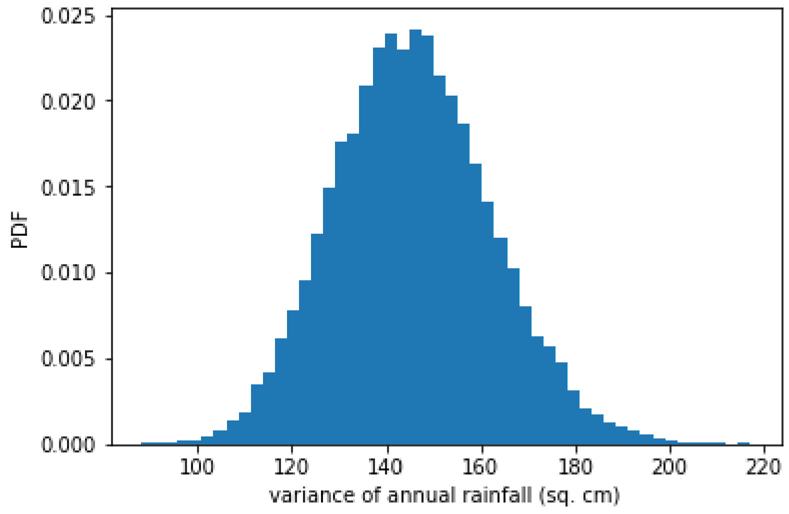
Out[22]:
```
array([779.95407895, 820.87830827])
```

Bootstrap replicates of other statistics

- Sample mean is Normally distributed. This does not necessarily hold for other statistics, but no worry: we can always take bootstrap replicates!
- Generate bootstrap replicates for the variance of the annual rainfall at the Sheffield Weather Station and plot the histogram of the replicates.

```
In [23]: # Generate 10,000 bootstrap replicates of the variance: bs_replicates
         bs_replicates = draw_bs_reps(rainfall, np.var, 10000)

         # Put the variance in units of square centimeters
         bs_replicates = bs_replicates/100

         # Make a histogram of the results
         _ = plt.hist(bs_replicates, normed = True, bins = 50)
         _ = plt.xlabel('variance of annual rainfall (sq. cm)')
         _ = plt.ylabel('PDF')

         # Show the plot
         plt.show()
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py:6462: UserWarning: The 'normed' kwarg is de
precated, and has been replaced by the 'density' kwarg.
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



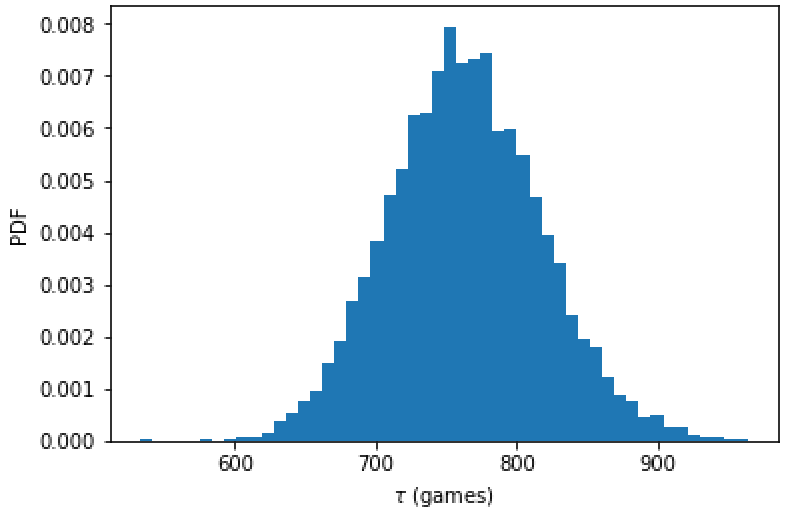
This is not normal distribution as it has longer right tail.

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

```
In [24]: # Draw bootstrap replicates of the mean no-hitter time (equal to tau): bs_replicates
bs_replicates = draw_bs_reps(nohitter_times,np.mean,10000)

conf_int = np.percentile(bs_replicates,[2.5,97.5])

print('95% confidence interval =', conf_int, 'games')

_ = plt.hist(bs_replicates, bins=50, density = 'normed')
_ = plt.xlabel(r'$\tau$ (games)')
_ = plt.ylabel('PDF')

plt.show()
```

```
95% confidence interval = [661.87320717 873.8686255 ] games
```

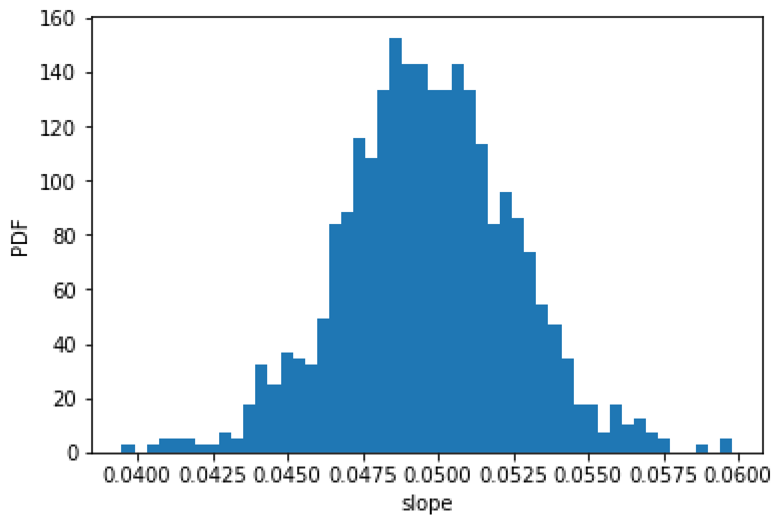


A function to do pairs bootstrap

- Pairs bootstrap involves resampling pairs of data. Each collection of pairs fit with a line, in this case using np.polyfit().

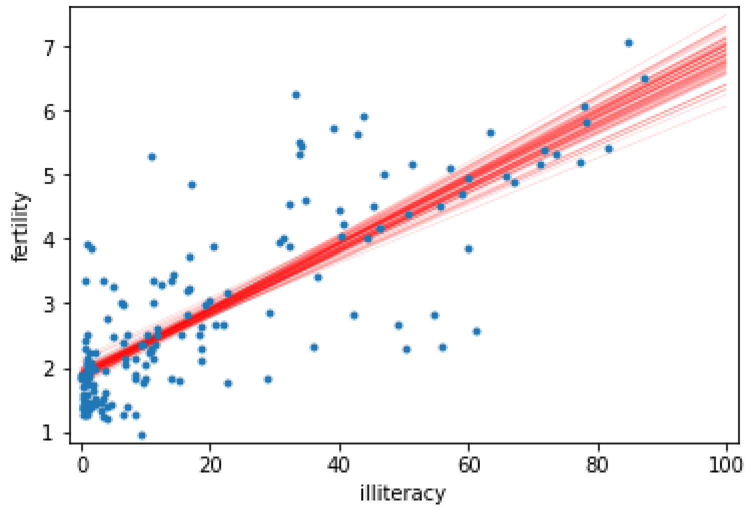
In [27]:
```
# Generate replicates of slope and intercept using pairs bootstrap
bs_slope_reps, bs_intercept_reps = draw_bs_pairs_linreg(illiteracy,fertility,1000)

# Compute and print 95% CI for slope
print(np.percentile(bs_slope_reps, [2.5,97.5]))

_ = plt.hist(bs_slope_reps, bins=50, density = 'normed')
_ = plt.xlabel('slope')
_ = plt.ylabel('PDF')
plt.show()
```

```
[0.04423499 0.0555734 ]
```



Plotting bootstrap regressions

In [39]:
```
# Generate array of x-values for bootstrap lines: x
x = np.array([0,100])
#This gives the scope of x values of the plot below. Note x is two element array but not a range.

# 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')
    #Here in the plot() for each i, two points are determined and thus a line is fixed.

# Plot the data (scattered)
_ = 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()
```



Introduction to hypothesis testing

Here we mainly focus on the hypothesis testing with numerical simulation. Specific problems are solved with python. A general introduction to hypothesis testing and its steps are on the notes for probability and statistics.

Generating a permutation sample

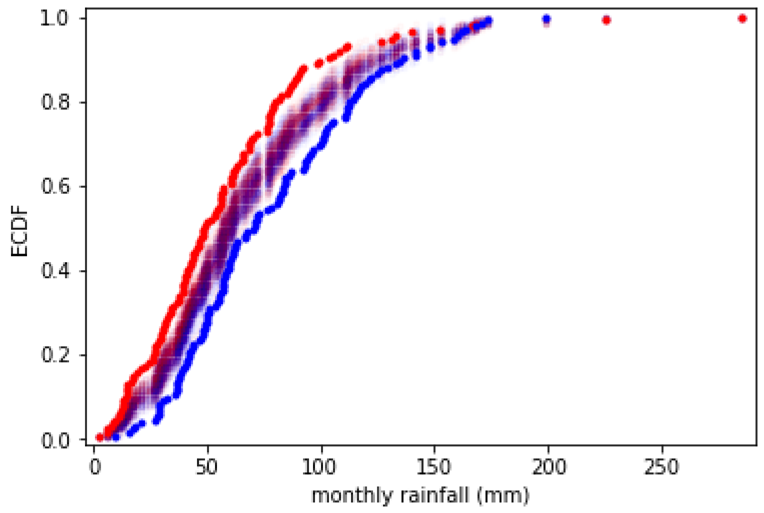
- Permutation sampling is a great way to simulate the hypothesis that two variables have identical probability distributions.
- A permutation sample of two arrays having respectively n_1 and n_2 entries is constructed by concatenating the arrays together, scrambling the contents of the concatenated array, and then taking the first n_1 entries as the permutation sample of the first array and the last n_1 entries as the permutation sample of the second array.

Visualizing permutation sampling

- Generate permutation samples and look at them graphically.
- 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. July and November rainfall are not identically distributed.

In [43]:
```
rain_july = np.array([ 66.2,  39.7,  76.4,  26.5,  11.2,  61.8,   6.1,  48.4,  89.2,
       104. ,  34. ,  60.6,  57.1,  79.1,  90.9,  32.3,  63.8,  78.2,
        27.5,  43.4,  30.1,  17.3,  77.5,  44.9,  92.2,  39.6,  79.4,
        66.1,  53.5,  98.5,  20.8,  55.5,  39.6,  56. ,  65.1,  14.8,
        13.2,  88.1,   8.4,  32.1,  19.6,  40.4,   2.2,  77.5, 105.4,
        77.2,  38. ,  27.1, 111.8,  17.2,  26.7,  23.3,  77.2,  87.2,
        27.7,  50.6,  60.3,  15.1,   6. ,  29.4,  39.3,  56.3,  80.4,
        85.3,  68.4,  72.5,  13.3,  28.4,  14.7,  37.4,  49.5,  57.2,
        85.9,  82.1,  31.8, 126.6,  30.7,  41.4,  33.9,  13.5,  99.1,
        70.2,  91.8,  61.3,  13.7,  54.9,  62.5,  24.2,  69.4,  83.1,
        44. ,  48.5,  11.9,  16.6,  66.4,  90. ,  34.9, 132.8,  33.4,
       225. ,   7.6,  40.9,  76.5,  48. , 140. ,  55.9,  54.1,  46.4,
        68.6,  52.2, 108.3,  14.6,  11.3,  29.8, 130.9, 152.4,  61. ,
        46.6,  43.9,  30.9, 111.1,  68.5,  42.2,   9.8, 285.6,  56.7,
       168.2,  41.2,  47.8, 166.6,  37.8,  45.4,  43.2])
rain_november = np.array([ 83.6,  30.9,  62.2,  37. ,  41. , 160.2,  18.2, 122.4,  71.3,
        44.2,  49.1,  37.6, 114.5,  28.8,  82.5,  71.9,  50.7,  67.7,
       112. ,  63.6,  42.8,  57.2,  99.1,  86.4,  84.4,  38.1,  17.7,
       102.2, 101.3,  58. ,  82. , 101.4,  81.4, 100.1,  54.6,  39.6,
        57.5,  29.2,  48.8,  37.3, 115.4,  55.6,  62. ,  95. ,  84.2,
       118.1, 153.2,  83.4, 104.7,  59. ,  46.4,  50. , 147.6,  76.8,
        59.9, 101.8, 136.6, 173. ,  92.5,  37. ,  59.8, 142.1,   9.9,
       158.2,  72.6,  28. , 112.9, 119.3, 199.2,  50.7,  44. , 170.7,
        67.2,  21.4,  61.3,  15.6, 106. , 116.2,  42.3,  38.5, 132.5,
        40.8, 147.5,  93.9,  71.4,  87.3, 163.7, 141.4,  62.6,  84.9,
        28.8, 121.1,  28.6,  32.4, 112. ,  50. ,  96.9,  81.8,  70.4,
       117.5,  41.2, 124.9,  78.2,  93. ,  53.5,  50.5,  42.6,  47.9,
        73.1, 129.1,  56.9, 103.3,  60.5, 134.3,  93.1,  49.5,  48.2,
       167.9,  27. , 111.1,  55.4,  36.2,  57.4,  66.8,  58.3,  60. ,
       161.6, 112.7,  37.4, 110.6,  56.6,  95.8, 126.8])
for i in range(50):
    perm_sample_1, perm_sample_2 = permutation_sample(rain_july, rain_november)

    x_1, y_1 = ecdf(perm_sample_1)
    x_2, y_2 = ecdf(perm_sample_2)

    _ = 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_july)
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()
```



Test statistics

When performing hypothesis tests, the choice of test statistic should be pertinent to the question we are seeking to answer in your hypothesis test

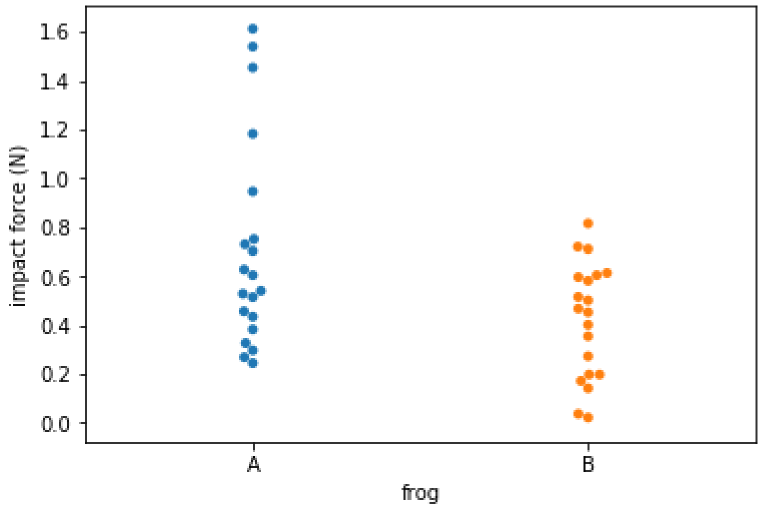
Generating permutation replicates

- A permutation replicate is a **single value** of a statistic computed from a permutation sample.
- The function draw_bs_reps() function generates bootstrap replicates, while draw_perm_reps() generates permutation 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 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.
- Do EDA first and make a bee swarm plot for the data.

In [3]:
```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

filePath = "C:/Users/ljyan/Desktop/courseNotes/dataScience/machineLearning/data/"
filename = "hello.txt"
file = filePath+filename
df = pd.read_csv(file,sep = '\s+', index_col=0)  #alternative way

# Make bee swarm plot
_ = sns.swarmplot(x='ID',y='impact_force', data = df) # All keywords argument

# Label axes
_ = plt.xlabel('frog')
_ = plt.ylabel('impact force (N)')

# Show the plot
plt.show()
```



- 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 different 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.
- 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.

```
In [13]: import numpy as np
force_a = np.array([1.612, 0.605, 0.327, 0.946, 0.541, 1.539, 0.529, 0.628, 1.453,
       0.297, 0.703, 0.269, 0.751, 0.245, 1.182, 0.515, 0.435, 0.383,
       0.457, 0.73 ])
force_b = np.array([0.172, 0.142, 0.037, 0.453, 0.355, 0.022, 0.502, 0.273, 0.72 ,
       0.582, 0.198, 0.198, 0.597, 0.516, 0.815, 0.402, 0.605, 0.711,
       0.614, 0.468])

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

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)

p = np.sum(perm_replicates >= empirical_diff_means) / len(perm_replicates)
# Understand why we use >= in the above sentence. This is related how the difference is calculated.

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

```
p-value = 0.0042
```

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: we cannot take it as a yes or no answer.

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.

- Thus we 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.
- The key here is we assume that two frogs have the same mean force. So we need first shift the force_b to have a mean of 0.55 N, as shown below.

In [15]:
```
force_b = np.array([0.172, 0.142, 0.037, 0.453, 0.355, 0.022, 0.502, 0.273, 0.72 ,
       0.582, 0.198, 0.198, 0.597, 0.516, 0.815, 0.402, 0.605, 0.711,
       0.614, 0.468])
translated_force_b = force_b + 0.55 - np.mean(force_b)

bs_replicates = draw_bs_reps(translated_force_b, np.mean, 10000)

p = np.sum(bs_replicates <= np.mean(force_b)) / 10000

print('p = ', p)
```

```
p =  0.0063
```

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

A bootstrap test for identical distributions

- Earlier we did the hypothesis test on identical distributions with permutation replicates. Now we will do the same pothesis test with bootstrap sampling.

In [49]:
```
empirical_diff_means = diff_of_means(force_a, force_b)

forces_concat = np.concatenate((force_a, force_b))
#IMPORTANT. a tuple is required above.
#I have made the same mistake twice.

bs_replicates = np.empty(10000)

for i in range(10000):
    bs_sample = np.random.choice(forces_concat, size=len(forces_concat))

    bs_replicates[i] = diff_of_means(bs_sample[:len(force_a)],
                                     bs_sample[len(force_a):])

p = np.sum(bs_replicates >= empirical_diff_means) / len(bs_replicates)
print('p-value =', p)
```

```
p-value = 0.0489
```

This is similar to what we got from the permutation test. These are very close, and indeed the tests are testing the same thing. **The bootstrap hypothesis test, while approximate, is more versatile.**

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

- One-sample bootstrap hypothesis test is impossible to do with permutation.
- Testing the hypothesis that two samples have the same distribution may be done with a bootstrap test, but a permutation test is preferred because it is more accurate (exact, in fact). But therein lies the limit of a permutation test; it is not very versatile. 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. This, too, is 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.

In [50]:
```
empirical_diff_means = np.mean(force_a) - np.mean(force_b)
mean_force = np.mean(np.concatenate((force_a,force_b)))

force_a_shifted = force_a - np.mean(force_a) + mean_force
force_b_shifted = force_b - np.mean(force_b) + mean_force

bs_replicates_a = draw_bs_reps(force_a_shifted, np.mean, 10000)
bs_replicates_b = draw_bs_reps(force_b_shifted, np.mean, 10000)

bs_replicates = bs_replicates_a - bs_replicates_b

p = np.sum(bs_replicates >= empirical_diff_means) / len(bs_replicates)
print('p-value =', p)
```

```
p-value = 0.0057
```

Not surprisingly, the more forgiving hypothesis, only that the means are equal as opposed to having identical distributions, gives a higher p-value.

Hypothesis test examples

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 yay. However, 91 Democrats and 35 Republicans voted nay. Did party affiliation make a difference in the vote?
- 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.
- Permute the party labels of the House voters and then arbitrarily divide them into "Democrats" and "Republicans" and compute the fraction of Democrats voting yay.

In [13]:
```
dems = np.array([True] * 153 + [False] * 91)
reps = np.array([True] * 136 + [False] * 35)

def frac_yay_dems(dems, reps):
    """Compute fraction of Democrat yay votes."""
    frac = np.sum(dems) / len(dems)
    return frac

perm_replicates = draw_perm_reps(dems, reps, frac_yay_dems, 10000)
p = np.sum(perm_replicates <= 153/244) / len(perm_replicates)
print('p-value =', p)
```

```
p-value = 0.0002
```

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?

Pretend you are working for a company that does all kinds of A/B tests on the user experience of its website. The following examples are essentially equivalent to the A/B test as the voting example Civil Rights Act of 1964?

- Measure the number of people who click on an ad on your company's website before and after changing its color.
- 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.
- An example below.

A time-on-website analog

- 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.
- 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."

```
In [52]: nht_dead = np.array([  -1,  894,   10,  130,    1,  934,   29,    6,  485,  254,  372,
                 81,  191,  355,  180,  286,   47,  269,  361,  173,  246,  492,
                462, 1319,   58,  297,   31, 2970,  640,  237,  434,  570,   77,
                271,  563, 3365,   89,    0,  379,  221,  479,  367,  628,  843,
               1613, 1101,  215,  684,  814,  278,  324,  161,  219,  545,  715,
                966,  624,   29,  450,  107,   20,   91, 1325,  124, 1468,  104,
               1309,  429,   62, 1878, 1104,  123,  251,   93,  188,  983,  166,
                 96,  702,   23,  524,   26,  299,   59,   39,   12,    2,  308,
               1114,  813,  887])
         nht_live = np.array([ 645, 2088,   42, 2090,   11,  886, 1665, 1084, 2900, 2432,  750,
               4021, 1070, 1765, 1322,   26,  548, 1525,   77, 2181, 2752,  127,
               2147,  211,   41, 1575,  151,  479,  697,  557, 2267,  542,  392,
                 73,  603,  233,  255,  528,  397, 1529, 1023, 1194,  462,  583,
                 37,  943,  996,  480, 1497,  717,  224,  219, 1531,  498,   44,
                288,  267,  600,   52,  269, 1086,  386,  176, 2199,  216,   54,
                675, 1243,  463,  650,  171,  327,  110,  774,  509,    8,  197,
                136,   12, 1124,   64,  380,  811,  232,  192,  731,  715,  226,
                605,  539, 1491,  323,  240,  179,  702,  156,   82, 1397,  354,
                778,  603, 1001,  385,  986,  203,  149,  576,  445,  180, 1403,
                252,  675, 1351, 2983, 1568,   45,  899, 3260, 1025,   31,  100,
               2055, 4043,   79,  238, 3931, 2351,  595,  110,  215,    0,  563,
                206,  660,  242,  577,  179,  157,  192,  192, 1848,  792, 1693,
                 55,  388,  225, 1134, 1172, 1555,   31, 1582, 1044,  378, 1687,
               2915,  280,  765, 2819,  511, 1521,  745, 2491,  580, 2072, 6450,
                578,  745, 1075, 1103, 1549, 1520,  138, 1202,  296,  277,  351,
                391,  950,  459,   62, 1056, 1128,  139,  420,   87,   71,  814,
                603, 1349,  162, 1027,  783,  326,  101,  876,  381,  905,  156,
                419,  239,  119,  129,  467])
```

```
In [53]: nht_dead = np.array([  -1,  894,   10,  130,    1,  934,   29,    6,  485,  254,  372,
                  81,  191,  355,  180,  286,   47,  269,  361,  173,  246,  492,
                 462, 1319,   58,  297,   31, 2970,  640,  237,  434,  570,   77,
                 271,  563, 3365,   89,    0,  379,  221,  479,  367,  628,  843,
                1613, 1101,  215,  684,  814,  278,  324,  161,  219,  545,  715,
                 966,  624,   29,  450,  107,   20,   91, 1325,  124, 1468,  104,
                1309,  429,   62, 1878, 1104,  123,  251,   93,  188,  983,  166,
                  96,  702,   23,  524,   26,  299,   59,   39,   12,    2,  308,
                1114,  813,  887])
         nht_live = np.array([ 645, 2088,   42, 2090,   11,  886, 1665, 1084, 2900, 2432,  750,
                4021, 1070, 1765, 1322,   26,  548, 1525,   77, 2181, 2752,  127,
                2147,  211,   41, 1575,  151,  479,  697,  557, 2267,  542,  392,
                  73,  603,  233,  255,  528,  397, 1529, 1023, 1194,  462,  583,
                  37,  943,  996,  480, 1497,  717,  224,  219, 1531,  498,   44,
                 288,  267,  600,   52,  269, 1086,  386,  176, 2199,  216,   54,
                 675, 1243,  463,  650,  171,  327,  110,  774,  509,    8,  197,
                 136,   12, 1124,   64,  380,  811,  232,  192,731,  715,  226,
                 605,  539, 1491])

         # Compute the observed difference in mean inter-no-hitter times: nht_diff_obs
         #nht_diff_obs = diff_of_means(nht_dead, nht_live)
         nht_diff_obs = diff_of_means(nht_live,nht_dead)

         perm_replicates = draw_perm_reps(nht_dead, nht_live,
                                          diff_of_means, 10000)

         p = np.sum(perm_replicates >= nht_diff_obs) / len(perm_replicates)
         print('p-val =',p)

         print(len(nht_dead),len(nht_live))
```

```
p-val = 0.0015
91 91
```

What should you have done first?

- Although A/B test can handle many types of problems. However, other types test can also do a better job.
- It is useful to do histograms, or ECDF, first, and then do A/B tests.

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. Is this a correct hypothesis?
- To do the test, simulate the data assuming the null hypothesis is true. Of the following choices, which is the best way to to do it?
    - Do a permutation test: Permute both the illiteracy and fertility values to generate a new set of (illiteracy, fertility data). This works perfectly, and is exact because it uses all data and eliminates any correlation because which illiteracy value pairs to which fertility value is shuffled. However, it is computationally inefficient not necessary to permute both illiteracy and fertility.
    - Do a permutation test: Permute the illiteracy values but leave the fertility values fixed to generate a new set of (illiteracy, fertility) data (because they are independent). 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

Now we choose the statistics as Pearson correlation to test the hypothesis.

In [55]:
```
r_obs = pearson_r(illiteracy, fertility)

perm_replicates = np.empty(10000)

for i in range(10000):
    illiteracy_permuted = np.random.permutation(illiteracy)


    perm_replicates[i] = pearson_r(illiteracy_permuted, fertility)

p = np.sum(perm_replicates >= r_obs) / len(perm_replicates)
print('p-val =', p)
```

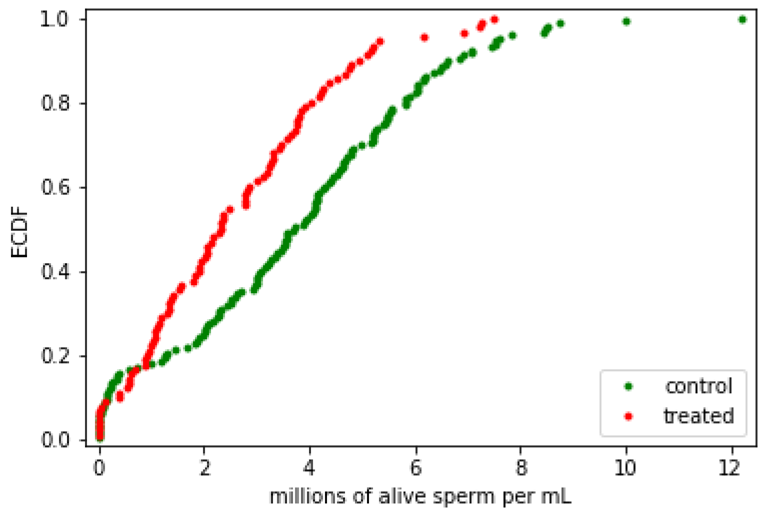
```
p-val = 0.0
```

Do neonicotinoid insecticides have unintended consequences?

- 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.
- Study how the pesticide treatment affected the count of live sperm per half milliliter of semen.

- First do EDA. 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).

```
In [4]: import numpy as np
        import matplotlib.pyplot as plt
        #Note the following two arrays are very different in size.
        control = np.array([ 4.159234,  4.408002,  0.172812,  3.498278,  3.104912,  5.164174,
                6.615262,  4.633066,  0.170408,  2.65    ,  0.0875  ,  1.997148,
                6.92668 ,  4.574932,  3.896466,  5.209814,  3.70625 ,  0.      ,
                4.62545 ,  3.01444 ,  0.732652,  0.4     ,  6.518382,  5.225   ,
                6.218742,  6.840358,  1.211308,  0.368252,  3.59937 ,  4.212158,
                6.052364,  2.115532,  6.60413 ,  5.26074 ,  6.05695 ,  6.481172,
                3.171522,  3.057228,  0.218808,  5.215112,  4.465168,  2.28909 ,
                3.732572,  2.17087 ,  1.834326,  6.074862,  5.841978,  8.524892,
                4.698492,  2.965624,  2.324206,  3.409412,  4.830726,  0.1     ,
                0.      ,  4.101432,  3.478162,  1.009688,  4.999296,  4.32196 ,
                0.299592,  3.606032,  7.54026 ,  4.284024,  0.057494,  6.036668,
                2.924084,  4.150144,  1.256926,  4.666502,  4.806594,  2.52478 ,
                2.027654,  2.52283 ,  4.735598,  2.033236,  0.      ,  6.177294,
                2.601834,  3.544408,  3.6045  ,  5.520346,  4.80698 ,  3.002478,
                3.559816,  7.075844, 10.      ,  0.139772,  6.17171 ,  3.201232,
                8.459546,  0.17857 ,  7.088276,  5.496662,  5.415086,  1.932282,
                3.02838 ,  7.47996 ,  1.86259 ,  7.838498,  2.242718,  3.292958,
                6.363644,  4.386898,  8.47533 ,  4.156304,  1.463956,  4.533628,
                5.573922,  1.29454 ,  7.547504,  3.92466 ,  5.820258,  4.118522,
                4.125   ,  2.286698,  0.591882,  1.273124,  0.      ,  0.      ,
                0.      , 12.22502 ,  7.601604,  5.56798 ,  1.679914,  8.77096 ,
                5.823942,  0.258374,  0.      ,  5.899236,  5.486354,  2.053148,
                3.25541 ,  2.72564 ,  3.364066,  2.43427 ,  5.282548,  3.963666,
                0.24851 ,  0.347916,  4.046862,  5.461436,  4.066104,  0.      ,
                0.065   ])
        treated = np.array([1.342686, 1.058476, 3.793784, 0.40428 , 4.528388, 2.142966,
               3.937742, 0.1375  , 6.919164, 0.      , 3.597812, 5.196538,
               2.78955 , 2.3229  , 1.090636, 5.323916, 1.021618, 0.931836,
               2.78    , 0.412202, 1.180934, 2.8674  , 0.      , 0.064354,
               3.008348, 0.876634, 0.      , 4.971712, 7.280658, 4.79732 ,
               2.084956, 3.251514, 1.9405  , 1.566192, 0.58894 , 5.219658,
               0.977976, 3.124584, 1.297564, 1.433328, 4.24337 , 0.880964,
               2.376566, 3.763658, 1.918426, 3.74    , 3.841726, 4.69964 ,
               4.386876, 0.      , 1.127432, 1.845452, 0.690314, 4.185602,
               2.284732, 7.237594, 2.185148, 2.799124, 3.43218 , 0.63354 ,
               1.142496, 0.586   , 2.372858, 1.80032 , 3.329306, 4.028804,
               3.474156, 7.508752, 2.032824, 1.336556, 1.906496, 1.396046,
               2.488104, 4.759114, 1.07853 , 3.19927 , 3.814252, 4.275962,
               2.817056, 0.552198, 3.27194 , 5.11525 , 2.064628, 0.      ,
       3.34101 , 6.177322, 0.      , 3.66415 , 2.352582, 1.531696])

x_control, y_control = ecdf(control)
x_treated, y_treated = ecdf(treated)


plt.plot(x_control, y_control, marker='.', linestyle='none', color = 'green')
plt.plot(x_treated, y_treated, marker='.', linestyle='none', color = 'red')


plt.margins(0.02)

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

plt.xlabel('millions of alive sperm per mL')
plt.ylabel('ECDF')
plt.show()
```



Bootstrap hypothesis test on bee sperm counts

- 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.
- Here we assume the same means but not the same distributions. So it is incorrect to use permutation.

```
In [67]: diff_means = np.mean(control) - np.mean(treated)

         mean_count = np.mean(np.concatenate((control,treated)))
         #I must add a (). Otherwise a mistake for third times

         control_shifted = control - np.mean(control) + mean_count
         treated_shifted = treated - np.mean(treated) + mean_count

         bs_reps_control = draw_bs_reps(control_shifted,
                                np.mean, size=10000)
         bs_reps_treated = draw_bs_reps(treated_shifted,
                                np.mean, size=10000)

         bs_replicates = bs_reps_control - bs_reps_treated

         p = np.sum(bs_replicates >= np.mean(control) - np.mean(treated)) \
                     / len(bs_replicates)
         print('p-value =', p)
```

```
p-value = 0.0
```

The p-value is small, 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.

Putting it all together: a case study

- This chapter covers almost everything learned in the two statistical courses. Just a fews not covered such as boxchart (Here data is not so much and thus beeswarm plot is already enough).
- Study evolution through data.

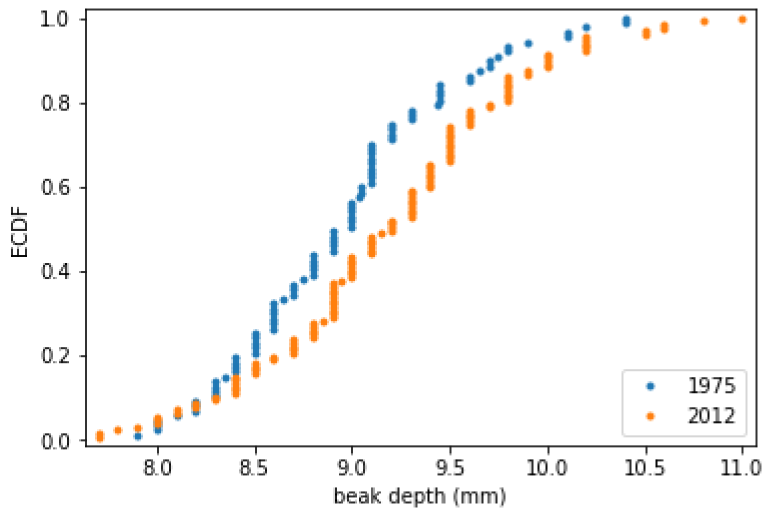
EDA of beak depths of Darwin's finches

- 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.
- Plot all of the beak depth measurements in 1975 and 2012 in a bee swarm plot. (not shown). The data is available in /data folder.

ECDFs of beak depths

- While bee swarm plots are useful, we found that ECDFs are often even better when doing EDA.

```
In [7]: bd_1975 = np.array([ 8.4 ,  8.8 ,  8.4 ,  8.  ,  7.9 ,  8.9 ,  8.6 ,  8.5 ,  8.9 ,
                9.1 ,  8.6 ,  9.8 ,  8.2 ,  9.  ,  9.7 ,  8.6 ,  8.2 ,  9.  ,
                8.4 ,  8.6 ,  8.9 ,  9.1 ,  8.3 ,  8.7 ,  9.6 ,  8.5 ,  9.1 ,
                9.  ,  9.2 ,  9.9 ,  8.6 ,  9.2 ,  8.4 ,  8.9 ,  8.5 , 10.4 ,
                9.6 ,  9.1 ,  9.3 ,  9.3 ,  8.8 ,  8.3 ,  8.8 ,  9.1 , 10.1 ,
                8.9 ,  9.2 ,  8.5 , 10.2 , 10.1 ,  9.2 ,  9.7 ,  9.1 ,  8.5 ,
                8.2 ,  9.  ,  9.3 ,  8.  ,  9.1 ,  8.1 ,  8.3 ,  8.7 ,  8.8 ,
                8.6 ,  8.7 ,  8.  ,  8.8 ,  9.  ,  9.1 ,  9.74,  9.1 ,  9.8 ,
               10.4 ,  8.3 ,  9.44,  9.04,  9.  ,  9.05,  9.65,  9.45,  8.65,
                9.45,  9.45,  9.05,  8.75,  9.45,  8.35])
        bd_2012 = np.array([ 9.4 ,  8.9 ,  9.5 , 11.  ,  8.7 ,  8.4 ,  9.1 ,  8.7 , 10.2 ,
                9.6 ,  8.85,  8.8 ,  9.5 ,  9.2 ,  9.  ,  9.8 ,  9.3 ,  9.  ,
               10.2 ,  7.7 ,  9.  ,  9.5 ,  9.4 ,  8.  ,  8.9 ,  9.4 ,  9.5 ,
                8.  , 10.  ,  8.95,  8.2 ,  8.8 ,  9.2 ,  9.4 ,  9.5 ,  8.1 ,
                9.5 ,  8.4 ,  9.3 ,  9.3 ,  9.6 ,  9.2 , 10.  ,  8.9 , 10.5 ,
                8.9 ,  8.6 ,  8.8 ,  9.15,  9.5 ,  9.1 , 10.2 ,  8.4 , 10.  ,
               10.2 ,  9.3 , 10.8 ,  8.3 ,  7.8 ,  9.8 ,  7.9 ,  8.9 ,  7.7 ,
                8.9 ,  9.4 ,  9.4 ,  8.5 ,  8.5 ,  9.6 , 10.2 ,  8.8 ,  9.5 ,
                9.3 ,  9.  ,  9.2 ,  8.7 ,  9.  ,  9.1 ,  8.7 ,  9.4 ,  9.8 ,
                8.6 , 10.6 ,  9.  ,  9.5 ,  8.1 ,  9.3 ,  9.6 ,  8.5 ,  8.2 ,
                8.  ,  9.5 ,  9.7 ,  9.9 ,  9.1 ,  9.5 ,  9.8 ,  8.4 ,  8.3 ,
                9.6 ,  9.4 , 10.  ,  8.9 ,  9.1 ,  9.8 ,  9.3 ,  9.9 ,  8.9 ,
                8.5 , 10.6 ,  9.3 ,  8.9 ,  8.9 ,  9.7 ,  9.8 , 10.5 ,  8.4 ,
               10.  ,  9.  ,  8.7 ,  8.8 ,  8.4 ,  9.3 ,  9.8 ,  8.9 ,  9.8 ,
                9.1 ])
        x_1975, y_1975 = ecdf(bd_1975)
        x_2012, y_2012 = ecdf(bd_2012)

        _ = plt.plot(x_1975, y_1975, marker='.', linestyle='none')
        _ = plt.plot(x_2012, y_2012, marker='.', linestyle='none')

        plt.margins(0.02)

        _ = plt.xlabel('beak depth (mm)')
        _ = plt.ylabel('ECDF')
        _ = plt.legend(('1975', '2012'), loc='lower right')

        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.

In [16]:
```
mean_diff = np.mean(bd_2012) - np.mean(bd_1975)

bs_replicates_1975 = draw_bs_reps(bd_1975,np.mean,10000)
bs_replicates_2012 = draw_bs_reps(bd_2012,np.mean,10000)

bs_diff_replicates = bs_replicates_2012 - bs_replicates_1975
# print(bs_replicates_2012[0:5],bs_replicates_1975[0:5],bs_diff_replicates[0:5])

conf_int = np.percentile(bs_diff_replicates,[2.5,97.5])

print('difference of means =', mean_diff, 'mm')
print('95% confidence interval =', conf_int, 'mm')
```

```
difference of means = 0.22622047244094645 mm
95% confidence interval = [0.05957532 0.39269409] mm
```

Hypothesis test: Are beaks deeper in 2012?

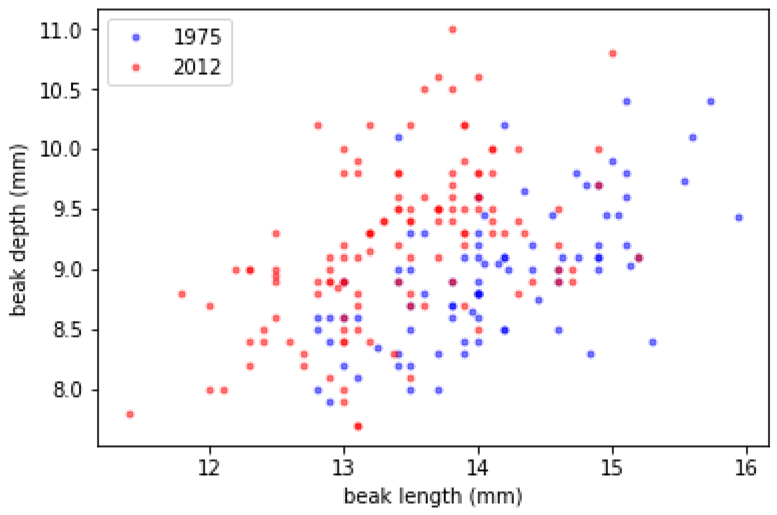
- Is it the possible that the beak depths means in two period are same while the observed difference of mean is from measuring volatility?
- The hypothesis we are testing is not that the beak depths come from the same distribution. For that we could use a permutation test. The hypothesis is that the 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.

In [17]:
```
combined_mean = np.mean(np.concatenate((bd_1975, bd_2012)))

bd_1975_shifted = bd_1975 - np.mean(bd_1975) + combined_mean
bd_2012_shifted = bd_2012 - np.mean(bd_2012) + combined_mean

bs_replicates_1975 = draw_bs_reps(bd_1975_shifted,np.mean,10000)
bs_replicates_2012 = draw_bs_reps(bd_2012_shifted,np.mean,10000)

bs_diff_replicates = bs_replicates_2012 - bs_replicates_1975

p = np.sum(bs_diff_replicates >= mean_diff) / len(bs_diff_replicates)
```

EDA of beak length and depth

- The beak length data are stored as bl_1975 and bl_2012, again with units of millimeters (mm).
- Make scatter plots of beak depth (y-axis) versus beak length (x-axis) for the 1975 and 2012 specimens.

```
In [18]: bl_1975 = np.array([13.9 , 14.  , 12.9 , 13.5 , 12.9 , 14.6 , 13.  , 14.2 , 14.  ,
                14.2 , 13.1 , 15.1 , 13.5 , 14.4 , 14.9 , 12.9 , 13.  , 14.9 ,
                14.  , 13.8 , 13.  , 14.75, 13.7 , 13.8 , 14.  , 14.6 , 15.2 ,
                13.5 , 15.1 , 15.  , 12.8 , 14.9 , 15.3 , 13.4 , 14.2 , 15.1 ,
                15.1 , 14.  , 13.6 , 14.  , 14.  , 13.9 , 14.  , 14.9 , 15.6 ,
                13.8 , 14.4 , 12.8 , 14.2 , 13.4 , 14.  , 14.8 , 14.2 , 13.5 ,
                13.4 , 14.6 , 13.5 , 13.7 , 13.9 , 13.1 , 13.4 , 13.8 , 13.6 ,
                14.  , 13.5 , 12.8 , 14.  , 13.4 , 14.9 , 15.54, 14.63, 14.73,
                15.73, 14.83, 15.94, 15.14, 14.23, 14.15, 14.35, 14.95, 13.95,
                14.05, 14.55, 14.05, 14.45, 15.05, 13.25])
         bl_2012 = np.array([14.3 , 12.5 , 13.7 , 13.8 , 12.  , 13.  , 13.  , 13.6 , 12.8 ,
                13.6 , 12.95, 13.1 , 13.4 , 13.9 , 12.3 , 14.  , 12.5 , 12.3 ,
                13.9 , 13.1 , 12.5 , 13.9 , 13.7 , 12.  , 14.4 , 13.5 , 13.8 ,
                13.  , 14.9 , 12.5 , 12.3 , 12.8 , 13.4 , 13.8 , 13.5 , 13.5 ,
                13.4 , 12.3 , 14.35, 13.2 , 13.8 , 14.6 , 14.3 , 13.8 , 13.6 ,
                12.9 , 13.  , 13.5 , 13.2 , 13.7 , 13.1 , 13.2 , 12.6 , 13.  ,
                13.9 , 13.2 , 15.  , 13.37, 11.4 , 13.8 , 13.  , 13.  , 13.1 ,
                12.8 , 13.3 , 13.5 , 12.4 , 13.1 , 14.  , 13.5 , 11.8 , 13.7 ,
                13.2 , 12.2 , 13.  , 13.1 , 14.7 , 13.7 , 13.5 , 13.3 , 14.1 ,
                12.5 , 13.7 , 14.6 , 14.1 , 12.9 , 13.9 , 13.4 , 13.  , 12.7 ,
                12.1 , 14.  , 14.9 , 13.9 , 12.9 , 14.6 , 14.  , 13.  , 12.7 ,
                14.  , 14.1 , 14.1 , 13.  , 13.5 , 13.4 , 13.9 , 13.1 , 12.9 ,
                14.  , 14.  , 14.1 , 14.7 , 13.4 , 13.8 , 13.4 , 13.8 , 12.4 ,
                14.1 , 12.9 , 13.9 , 14.3 , 13.2 , 14.2 , 13.  , 14.6 , 13.1 ,
                15.2 ])
         # 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()
```



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.

In [19]:
```
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_1975,1000)

bs_slope_reps_2012, bs_intercept_reps_2012 = draw_bs_pairs_linreg(bl_2012,bd_2012,1000)

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('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 = [0.32668441 0.59110117]
1975: intercept = 2.3908752365842263 conf int = [0.66212214 4.37871857]
2012: slope = 0.462630358835313 conf int = [0.33693208 0.59905005]
2012: intercept = 2.9772474982360198 conf int = [1.16315768 4.63992523]
```

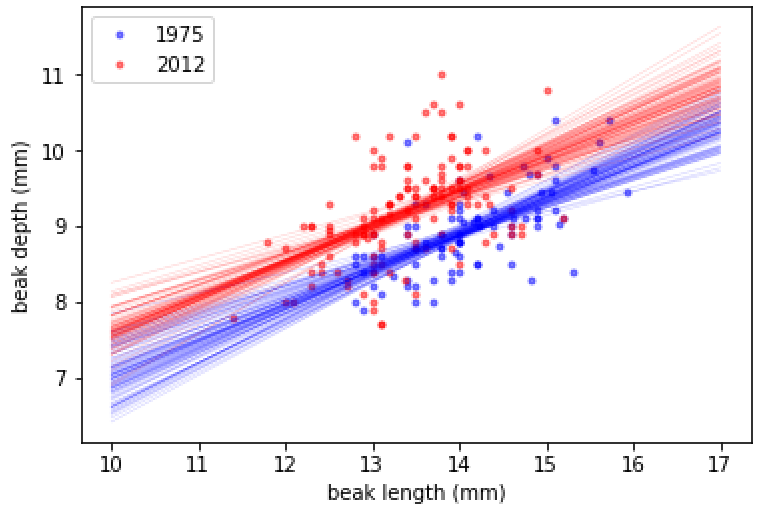
- It looks like they have the same slope, but different intercepts.
- It seems the different intercepts indicate the same conclusion drawn earlier: Not keep the same shape but : depth bigger, and length shorter.

Displaying the linear regression results

In [20]:
```
bs_slope_reps_1975 = np.array(
       [0.47134866, 0.53048191, 0.4389587 , 0.31389207, 0.55800189,
       0.44951317, 0.49750473, 0.36316442, 0.483553  , 0.44903622,
       0.45506655, 0.49499988, 0.47054508, 0.45802781, 0.51262947,
       0.52379531, 0.6079272 , 0.5200017 , 0.3741997 , 0.35555126,
       0.52907091, 0.46266681, 0.402726  , 0.47552382, 0.44677648,
       0.59315367, 0.58056774, 0.5408831 , 0.53729157, 0.53845703,
       0.49356109, 0.34710726, 0.44613105, 0.40804082, 0.49789028,
       0.49535537, 0.50901668, 0.35621315, 0.5282537 , 0.61466243,
       0.34215929, 0.46304499, 0.42956401, 0.31804759, 0.49038087,
       0.45269251, 0.42868738, 0.36716274, 0.3915092 , 0.56978366,
       0.48794121, 0.59219115, 0.38397442, 0.38523807, 0.45388343,
       0.51484475, 0.58443076, 0.52347349, 0.50205208, 0.59366747,
       0.56282699, 0.41896699, 0.57086934, 0.45024555, 0.53269717,
       0.541468  , 0.51973478, 0.37645289, 0.51446991, 0.46611665,
       0.50024092, 0.58482257, 0.37336028, 0.45394289, 0.47232794,
       0.43354103, 0.46267904, 0.41434392, 0.49152938, 0.26870374,
       0.46670887, 0.5078806 , 0.49180947, 0.58019869, 0.49615003,
       0.50005141, 0.47041797, 0.46017647, 0.57777062, 0.46235055,
       0.43419793, 0.45422523, 0.37563296, 0.4460656 , 0.3496609 ,
       0.45492812, 0.3677423 , 0.37041837, 0.39930878, 0.56496914])
bs_intercept_reps_1975 = np.array(
       [2.27942564, 1.41883129, 2.75361382, 4.4347406 , 1.02404388,
       2.54747202, 1.88603092, 3.79378417, 2.150503  , 2.60527654,
       2.51160229, 1.92062698, 2.37717268, 2.45680823, 1.68728308,
       1.60842131, 0.48104519, 1.61486809, 3.66095549, 3.90975449,
       1.49312804, 2.38236732, 3.23070157, 2.27225828, 2.66995479,
       0.57841674, 0.89629164, 1.32361369, 1.45107579, 1.27715441,
       1.97325432, 4.08170672, 2.65842355, 3.23547074, 1.94791522,
       1.98747637, 1.85922347, 3.94388361, 1.47104915, 0.26986774,
       4.18896106, 2.41940281, 2.92382233, 4.42533672, 1.95106054,
       2.67217129, 2.87606824, 3.73292168, 3.51873407, 0.91768485,
       2.06147536, 0.68197779, 3.50507497, 3.46467743, 2.59946526,
       1.73126325, 0.76473825, 1.62903971, 1.87115481, 0.53478046,
       0.94733624, 3.0005251 , 0.91738531, 2.62192868, 1.45958883,
       1.24250681, 1.66836242, 3.6399714 , 1.66970744, 2.31445337,
       1.97925983, 0.68931269, 3.69216823, 2.51553422, 2.27407697,
       2.79379784, 2.39471282, 3.09834188, 2.00243803, 5.16149402,
       2.30865372, 1.82665693, 2.0758009 , 0.81072918, 2.02063049,
       1.88209439, 2.3370979 , 2.4345369 , 0.75806804, 2.51661948,
       2.81849569, 2.50235235, 3.61478022, 2.61796687, 4.01807833,
       2.55267295, 3.80065854, 3.76046251, 3.26878997, 0.97814343])
```

```
bs_slope_reps_2012 = np.array(
       [0.59628281, 0.48439024, 0.42371069, 0.49629564, 0.51491878,
       0.48186363, 0.52810501, 0.53940898, 0.45992958, 0.45028497,
       0.41336967, 0.466558  , 0.54142368, 0.41716643, 0.50815707,
       0.43456584, 0.46348246, 0.34560176, 0.33699337, 0.51028172,
       0.47719388, 0.45820699, 0.54241929, 0.4717624 , 0.51253946,
       0.36148288, 0.4285887 , 0.45619521, 0.45965693, 0.48566772,
       0.65982296, 0.54846441, 0.67705445, 0.38769006, 0.46181636,
       0.57253065, 0.42364594, 0.48227146, 0.34770522, 0.44078756,
       0.54893477, 0.50356273, 0.42924231, 0.50838059, 0.31667064,
       0.48754205, 0.40774797, 0.42877991, 0.46231406, 0.48631945,
       0.48003865, 0.58455173, 0.48295332, 0.4642754 , 0.41711567,
       0.51022938, 0.46971345, 0.45997399, 0.46418409, 0.41240009,
       0.44767395, 0.37561945, 0.52867512, 0.40794457, 0.57751335,
       0.53380107, 0.54677372, 0.40433688, 0.49249857, 0.49314773,
       0.3012513 , 0.42908407, 0.43423558, 0.45175542, 0.3158103 ,
       0.34822848, 0.53845159, 0.61086607, 0.37055931, 0.46002918,
       0.4328848 , 0.48121256, 0.36676387, 0.47050143, 0.4528893 ,
       0.49624521, 0.4847631 , 0.39260194, 0.50522063, 0.34072385,
       0.47166831, 0.4820171 , 0.40489527, 0.46523298, 0.39137761,
       0.4039586 , 0.51029826, 0.41311951, 0.51270776, 0.43708734])
bs_intercept_reps_2012 = np.array(
       [1.13404465, 2.74184954, 3.39507267, 2.59482656, 2.35589454,
       2.73239097, 2.09436913, 1.90496321, 2.97656553, 3.16210863,
       3.58857463, 2.90946556, 1.98260442, 3.54463394, 2.39699885,
       3.38381154, 2.90490611, 4.68250739, 4.69050666, 2.33153459,
       2.82586946, 3.01032569, 1.88004065, 2.78119182, 2.37620524,
       4.31785985, 3.39593223, 3.08399697, 2.94636855, 2.62204972,
       0.32982159, 1.81886411, 0.12593015, 3.9550174 , 2.99803061,
       1.58581866, 3.45619131, 2.70785562, 4.46109103, 3.2236347 ,
       1.83990819, 2.39480652, 3.37826199, 2.46200175, 4.90345327,
       2.5600655 , 3.75100902, 3.41962052, 3.01020854, 2.67402403,
       2.73160625, 1.37324272, 2.71665479, 2.9659572 , 3.51190224,
       2.40861217, 2.90557792, 3.01693735, 2.95343924, 3.68235777,
       3.17128478, 4.18024784, 2.09121837, 3.74368494, 1.55267828,
       2.10235154, 1.90571839, 3.62901495, 2.61042102, 2.66203436,
       5.08656166, 3.34649795, 3.39840946, 3.09323066, 5.09647858,
       4.46437695, 2.01421553, 1.03568658, 4.22270941, 2.98028548,
       3.44170145, 2.71825519, 4.24532595, 2.91738888, 3.13609802,
       2.59227191, 2.70760862, 3.94823532, 2.46344915, 4.68918213,
       2.89052629, 2.76343499, 3.69298434, 2.95164292, 3.91149135,
       3.79092379, 2.36436701, 3.68804179, 2.37990098, 3.29606352])
```

```
_ = plt.plot(bl_1975, bd_1975, marker='.',
             linestyle='none', color='blue', alpha=0.5)

_ = plt.plot(bl_2012, bd_2012, marker='.',
             linestyle='none', color='red', alpha=0.5)

_ = plt.xlabel('beak length (mm)')
_ = plt.ylabel('beak depth (mm)')
_ = plt.legend(('1975', '2012'), loc='upper left')

x = np.array([10,17])

#Note here we only need two points of x.
# 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')
plt.show()
```



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.
- Now compare the ratio of beak length to beak depth. Let's make that comparison.

In [21]:

```
ratio_1975 = bl_1975/bd_1975
ratio_2012 = bl_2012/bd_2012

mean_ratio_1975 = np.mean(ratio_1975)
mean_ratio_2012 = np.mean(ratio_2012)

bs_replicates_1975 = draw_bs_reps(ratio_1975,np.mean,10000)
bs_replicates_2012 = draw_bs_reps(ratio_2012,np.mean,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)
```

```
1975: mean ratio = 1.5788823771858533 conf int = [1.55709126 1.60059958]
2012: mean ratio = 1.4658342276847767 conf int = [1.44410569 1.48759261]
```

How different is the ratio?

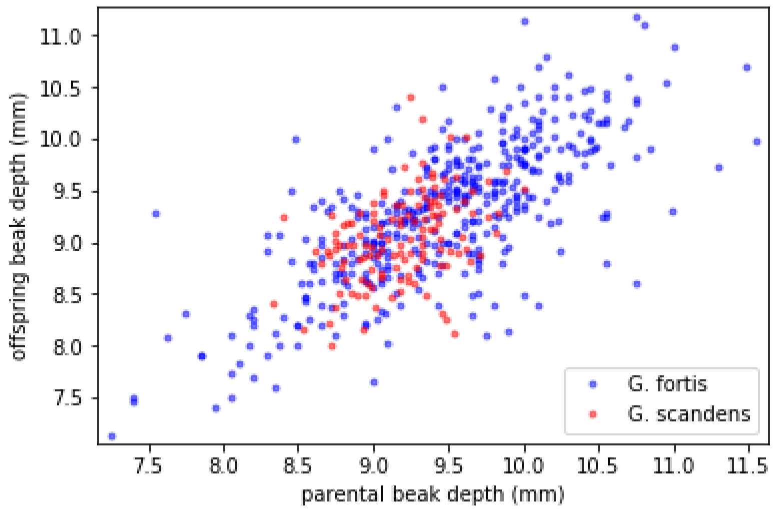
In the last exercise, you showed that the mean beak length to depth ratio was 1.58 in 1975 and 1.47 in 2012. The low end of the 1975 99% confidence interval was 1.56 mm and the high end of the 99% confidence interval in 2012 was 1.49 mm. In addition to these results, what would you say about the ratio of beak length to depth?

Answer: 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.

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.
- Scatter plotting 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.

In [24]:
```
bd_parent_scandens = np.array(
        [ 8.3318,  8.4035,  8.5317,  8.7202,  8.7089,  8.7541,  8.773 ,
        8.8107,  8.7919,  8.8069,  8.6523,  8.6146,  8.6938,  8.7127,
        8.7466,  8.7504,  8.7805,  8.7428,  8.7164,  8.8032,  8.8258,
        8.856 ,  8.9012,  8.9125,  8.8635,  8.8258,  8.8522,  8.8974,
        8.9427,  8.9879,  8.9615,  8.9238,  8.9351,  9.0143,  9.0558,
        9.0596,  8.9917,  8.905 ,  8.9314,  8.9465,  8.9879,  8.9804,
        9.0219,  9.052 ,  9.0407,  9.0407,  8.9955,  8.9992,  8.9992,
        9.0747,  9.0747,  9.5385,  9.4781,  9.4517,  9.3537,  9.2707,
        9.1199,  9.1689,  9.1425,  9.135 ,  9.1011,  9.1727,  9.2217,
        9.2255,  9.2821,  9.3235,  9.3198,  9.3198,  9.3198,  9.3273,
        9.3725,  9.3989,  9.4253,  9.4593,  9.4442,  9.4291,  9.2632,
        9.2293,  9.1878,  9.1425,  9.1275,  9.1802,  9.1765,  9.2481,
        9.2481,  9.1991,  9.1689,  9.1765,  9.2406,  9.3198,  9.3235,
        9.1991,  9.2971,  9.2443,  9.316 ,  9.2934,  9.3914,  9.3989,
        9.5121,  9.6176,  9.5535,  9.4668,  9.3725,  9.3348,  9.3763,
        9.3839,  9.4216,  9.4065,  9.3348,  9.4442,  9.4367,  9.5083,
        9.448 ,  9.4781,  9.595 ,  9.6101,  9.5686,  9.6365,  9.7119,
        9.8213,  9.825 ,  9.7609,  9.6516,  9.5988,  9.546 ,  9.6516,
        9.7572,  9.8854, 10.0023,  9.3914])
bd_offspring_scandens = np.array(
        [ 8.419 ,  9.2468,  8.1532,  8.0089,  8.2215,  8.3734,  8.5025,
        8.6392,  8.7684,  8.8139,  8.7911,  8.9051,  8.9203,  8.8747,
        8.943 ,  9.0038,  8.981 ,  9.0949,  9.2696,  9.1633,  9.1785,
        9.1937,  9.2772,  9.0722,  8.9658,  8.9658,  8.5025,  8.4949,
        8.4949,  8.5633,  8.6013,  8.6468,  8.1532,  8.3734,  8.662 ,
        8.6924,  8.7456,  8.8367,  8.8595,  8.9658,  8.9582,  8.8671,
        8.8671,  8.943 ,  9.0646,  9.1405,  9.2089,  9.2848,  9.3759,
        9.4899,  9.4519,  8.1228,  8.2595,  8.3127,  8.4949,  8.6013,
        8.4646,  8.5329,  8.7532,  8.8823,  9.0342,  8.6392,  8.6772,
        8.6316,  8.7532,  8.8291,  8.8975,  8.9734,  9.0494,  9.1253,
        9.1253,  9.1253,  9.1785,  9.2848,  9.4595,  9.3608,  9.2089,
        9.2544,  9.3684,  9.3684,  9.2316,  9.1709,  9.2316,  9.0342,
        8.8899,  8.8291,  8.981 ,  8.8975, 10.4089, 10.1886,  9.7633,
        9.7329,  9.6114,  9.5051,  9.5127,  9.3684,  9.6266,  9.5354,
       10.0215, 10.0215,  9.6266,  9.6038,  9.4063,  9.2316,  9.338 ,
        9.262 ,  9.262 ,  9.4063,  9.4367,  9.0342,  8.943 ,  8.9203,
        8.7835,  8.7835,  9.057 ,  8.9354,  8.8975,  8.8139,  8.8671,
        9.0873,  9.2848,  9.2392,  9.2924,  9.4063,  9.3152,  9.4899,
        9.5962,  9.6873,  9.5203,  9.6646])
bd_parent_fortis = np.array(
        [10.1  ,  9.55 ,  9.4  , 10.25 , 10.125,  9.7  ,  9.05 ,  7.4  ,
 9.   ,  8.65 ,  9.625,  9.9  ,  9.55 ,  9.05 ,  8.35 , 10.1  ,
10.1  ,  9.9  , 10.225, 10.   , 10.55 , 10.45 ,  9.2  , 10.2  ,
 8.95 , 10.05 , 10.2  ,  9.5  ,  9.925,  9.95 , 10.05 ,  8.75 ,
 9.2  , 10.15 ,  9.8  , 10.7  , 10.5  ,  9.55 , 10.55 , 10.475,
 8.65 , 10.7  ,  9.1  ,  9.4  , 10.3  ,  9.65 ,  9.5  ,  9.7  ,
10.525,  9.95 , 10.1  ,  9.75 , 10.05 ,  9.9  , 10.   ,  9.1  ,
 9.45 ,  9.25 ,  9.5  , 10.   , 10.525,  9.9  , 10.4  ,  8.95 ,
 9.4  , 10.95 , 10.75 , 10.1  ,  8.05 ,  9.1  ,  9.55 ,  9.05 ,
10.2  , 10.   , 10.55 , 10.75 ,  8.175,  9.7  ,  8.8  , 10.75 ,
 9.3  ,  9.7  ,  9.6  ,  9.75 ,  9.6  , 10.45 , 11.   , 10.85 ,
10.15 , 10.35 , 10.4  ,  9.95 ,  9.1  , 10.1  ,  9.85 ,  9.625,
 9.475,  9.   ,  9.25 ,  9.1  ,  9.25 ,  9.2  ,  9.95 ,  8.65 ,
 9.8  ,  9.4  ,  9.   ,  8.55 ,  8.75 ,  9.65 ,  8.95 ,  9.15 ,
 9.85 , 10.225,  9.825, 10.   ,  9.425, 10.4  ,  9.875,  8.95 ,
 8.9  ,  9.35 , 10.425, 10.   , 10.175,  9.875,  9.875,  9.15 ,
 9.45 ,  9.025,  9.7  ,  9.7  , 10.05 , 10.3  ,  9.6  , 10.   ,
 9.8  , 10.05 ,  8.75 , 10.55 ,  9.7  , 10.   ,  9.85 ,  9.8  ,
 9.175,  9.65 ,  9.55 ,  9.9  , 11.55 , 11.3  , 10.4  , 10.8  ,
 9.8  , 10.45 , 10.   , 10.75 ,  9.35 , 10.75 ,  9.175,  9.65 ,
 8.8  , 10.55 , 10.675,  9.95 ,  9.55 ,  8.825,  9.7  ,  9.85 ,
 9.8  ,  9.55 ,  9.275, 10.325,  9.15 ,  9.35 ,  9.15 ,  9.65 ,
10.575,  9.975,  9.55 ,  9.2  ,  9.925,  9.2  ,  9.3  ,  8.775,
 9.325,  9.175,  9.325,  8.975,  9.7  ,  9.5  , 10.225, 10.025,
 8.2  ,  8.2  ,  9.55 ,  9.05 ,  9.6  ,  9.6  , 10.15 ,  9.875,
10.485, 11.485, 10.985,  9.7  ,  9.65 ,  9.35 , 10.05 , 10.1  ,
 9.9  ,  8.95 ,  9.3  ,  9.95 ,  9.45 ,  9.5  ,  8.45 ,  8.8  ,
 8.525,  9.375, 10.2  ,  7.625,  8.375,  9.25 ,  9.4  , 10.55 ,
 8.9  ,  8.8  ,  9.   ,  8.575,  8.575,  9.6  ,  9.375,  9.6  ,
 9.95 ,  9.6  , 10.2  ,  9.85 ,  9.625,  9.025, 10.375, 10.25 ,
 9.3  ,  9.5  ,  9.55 ,  8.55 ,  9.05 ,  9.9  ,  9.8  ,  9.75 ,
10.25 ,  9.1  ,  9.65 , 10.3  ,  8.9  ,  9.95 ,  9.5  ,  9.775,
 9.425,  7.75 ,  7.55 ,  9.1  ,  9.6  ,  9.575,  8.95 ,  9.65 ,
 9.65 ,  9.65 ,  9.525,  9.85 ,  9.05 ,  9.3  ,  8.9  ,  9.45 ,
10.   ,  9.85 ,  9.25 , 10.1  ,  9.125,  9.65 ,  9.1  ,  8.05 ,
 7.4  ,  8.85 ,  9.075,  9.   ,  9.7  ,  8.7  ,  9.45 ,  9.7  ,
 8.35 ,  8.85 ,  9.7  ,  9.45 , 10.3  , 10.   , 10.45 ,  9.45 ,
 8.5  ,  8.3  , 10.   ,  9.225,  9.75 ,  9.15 ,  9.55 ,  9.   ,
 9.275,  9.35 ,  8.95 ,  9.875,  8.45 ,  8.6  ,  9.7  ,  8.55 ,
 9.05 ,  9.6  ,  8.65 ,  9.2  ,  8.95 ,  9.6  ,  9.15 ,  9.4  ,
 8.95 ,  9.95 , 10.55 ,  9.7  ,  8.85 ,  8.8  , 10.   ,  9.05 ,
 8.2  ,  8.1  ,  7.25 ,  8.3  ,  9.15 ,  8.6  ,  9.5  ,  8.05 ,
 9.425,  9.3  ,  9.8  ,  9.3  ,  9.85 ,  9.5  ,  8.65 ,  9.825,
 9.   , 10.45 ,  9.1  ,  9.55 ,  9.05 , 10.   ,  9.35 ,  8.375,
        8.3  ,  8.8  , 10.1  ,  9.5  ,  9.75 , 10.1  ,  9.575,  9.425,
        9.65 ,  8.725,  9.025,  8.5  ,  8.95 ,  9.3  ,  8.85 ,  8.95 ,
        9.8  ,  9.5  ,  8.65 ,  9.1  ,  9.4  ,  8.475,  9.35 ,  7.95 ,
        9.35 ,  8.575,  9.05 ,  8.175,  9.85 ,  7.85 ,  9.85 , 10.1  ,
        9.35 ,  8.85 ,  8.75 ,  9.625,  9.25 ,  9.55 , 10.325,  8.55 ,
        9.675,  9.15 ,  9.   ,  9.65 ,  8.6  ,  8.8  ,  9.   ,  9.95 ,
        8.4  ,  9.35 , 10.3  ,  9.05 ,  9.975,  9.975,  8.65 ,  8.725,
        8.2  ,  7.85 ,  8.775,  8.5  ,  9.4  ])

bd_offspring_fortis = np.array(
        [10.7 ,  9.78,  9.48,  9.6 , 10.27,  9.5 ,  9.  ,  7.46,  7.65,
        8.63,  9.81,  9.4 ,  9.48,  8.75,  7.6 , 10.  , 10.09,  9.74,
        9.64,  8.49, 10.15, 10.28,  9.2 , 10.01,  9.03,  9.94, 10.5 ,
        9.7 , 10.02, 10.04,  9.43,  8.1 ,  9.5 ,  9.9 ,  9.48, 10.18,
       10.16,  9.08, 10.39,  9.9 ,  8.4 , 10.6 ,  8.75,  9.46,  9.6 ,
        9.6 ,  9.95, 10.05, 10.16, 10.1 ,  9.83,  9.46,  9.7 ,  9.82,
       10.34,  8.02,  9.65,  9.87,  9.  , 11.14,  9.25,  8.14, 10.23,
        8.7 ,  9.8 , 10.54, 11.19,  9.85,  8.1 ,  9.3 ,  9.34,  9.19,
        9.52,  9.36,  8.8 ,  8.6 ,  8.  ,  8.5 ,  8.3 , 10.38,  8.54,
        8.94, 10.  ,  9.76,  9.45,  9.89, 10.9 ,  9.91,  9.39,  9.86,
        9.74,  9.9 ,  9.09,  9.69, 10.24,  8.9 ,  9.67,  8.93,  9.3 ,
        8.67,  9.15,  9.23,  9.59,  9.03,  9.58,  8.97,  8.57,  8.47,
        8.71,  9.21,  9.13,  8.5 ,  9.58,  9.21,  9.6 ,  9.32,  8.7 ,
       10.46,  9.29,  9.24,  9.45,  9.35, 10.19,  9.91,  9.18,  9.89,
        9.6 , 10.3 ,  9.45,  8.79,  9.2 ,  8.8 ,  9.69, 10.61,  9.6 ,
        9.9 ,  9.26, 10.2 ,  8.79,  9.28,  8.83,  9.76, 10.2 ,  9.43,
        9.4 ,  9.9 ,  9.5 ,  8.95,  9.98,  9.72,  9.86, 11.1 ,  9.14,
       10.49,  9.75, 10.35,  9.73,  9.83,  8.69,  9.58,  8.42,  9.25,
       10.12,  9.31,  9.99,  8.59,  8.74,  8.79,  9.6 ,  9.52,  8.93,
       10.23,  9.35,  9.35,  9.09,  9.04,  9.75, 10.5 ,  9.09,  9.05,
        9.54,  9.3 ,  9.06,  8.7 ,  9.32,  8.4 ,  8.67,  8.6 ,  9.53,
        9.77,  9.65,  9.43,  8.35,  8.26,  9.5 ,  8.6 ,  9.57,  9.14,
       10.79,  8.91,  9.93, 10.7 ,  9.3 ,  9.93,  9.51,  9.44, 10.05,
       10.13,  9.24,  8.21,  8.9 ,  9.34,  8.77,  9.4 ,  8.82,  8.83,
        8.6 ,  9.5 , 10.2 ,  8.09,  9.07,  9.29,  9.1 , 10.19,  9.25,
        8.98,  9.02,  8.6 ,  8.25,  8.7 ,  9.9 ,  9.65,  9.45,  9.38,
       10.4 ,  9.96,  9.46,  8.26, 10.05,  8.92,  9.5 ,  9.43,  8.97,
        8.44,  8.92, 10.3 ,  8.4 ,  9.37,  9.91, 10.  ,  9.21,  9.95,
        8.84,  9.82,  9.5 , 10.29,  8.4 ,  8.31,  9.29,  8.86,  9.4 ,
        9.62,  8.62,  8.3 ,  9.8 ,  8.48,  9.61,  9.5 ,  9.37,  8.74,
        9.31,  9.5 ,  9.49,  9.74,  9.2 ,  9.24,  9.7 ,  9.64,  9.2 ,
        7.5 ,  7.5 ,  8.7 ,  8.31,  9.  ,  9.74,  9.31, 10.5 ,  9.3 ,
        8.12,  9.34,  9.72,  9.  ,  9.65,  9.9 , 10.  , 10.1 ,  8.  ,
        9.07,  9.75,  9.33,  8.11,  9.36,  9.74,  9.9 ,  9.23,  9.7 ,
        8.2 ,  9.35,  9.49,  9.34,  8.87,  9.03,  9.07,  9.43,  8.2 ,
        9.19,  9.  ,  9.2 ,  9.06,  9.81,  8.89,  9.4 , 10.45,  9.64,
        9.03,  8.71,  9.91,  8.33,  8.2 ,  7.83,  7.14,  8.91,  9.18,
        8.8 ,  9.9 ,  7.73,  9.25,  8.7 ,  9.5 ,  9.3 ,  9.05, 10.18,
        8.85,  9.24,  9.15,  9.98,  8.77,  9.8 ,  8.65, 10.  ,  8.81,
        8.01,  7.9 ,  9.41, 10.18,  9.55,  9.08,  8.4 ,  9.75,  8.9 ,
        9.07,  9.35,  8.9 ,  8.19,  8.65,  9.19,  8.9 ,  9.28, 10.58,
        9.  ,  9.4 ,  8.91,  9.93, 10.  ,  9.37,  7.4 ,  9.  ,  8.8 ,
        9.18,  8.3 , 10.08,  7.9 ,  9.96, 10.4 ,  9.65,  8.8 ,  8.65,
        9.7 ,  9.23,  9.43,  9.93,  8.47,  9.55,  9.28,  8.85,  8.9 ,
        8.75,  8.63,  9.  ,  9.43,  8.28,  9.23, 10.4 ,  9.  ,  9.8 ,
        9.77,  8.97,  8.37,  7.7 ,  7.9 ,  9.5 ,  8.2 ,  8.8 ])
# 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)
# Set margins
plt.margins(0.02)

# 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()
```



It appears as though there is a stronger correlation in G. fortis than in G. scandens. This suggests that beak depth is more strongly inherited in G. fortis.

Correlation of offspring and parental data

- Compute Pearson correlation coefficient, between parents and offspring.
- To get confidence intervals, do a pairs bootstrap.

```
In [23]: r_scandens = pearson_r(bd_parent_scandens, bd_offspring_scandens)
         r_fortis = pearson_r(bd_parent_fortis, bd_offspring_fortis)

         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)

         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('G. scandens:', r_scandens, conf_int_scandens)
         print('G. fortis:', r_fortis, conf_int_fortis)
```

```
G. scandens: 0.4117063629401258 [0.27493603 0.5441022 ]
G. fortis: 0.7283412395518486 [0.6683546 0.7826807]
```

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

- Here the Pearson correlation is a measure of the correlation between parents and offspring, but might not be the best estimate of heritability.
- Define heritability as the ratio of the covariance between parent and offspring to the variance of the parents alone. Estimate the heritability and perform a pairs bootstrap calculation to get the 95% confidence interval.

In [25]:
```python
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]

    #The following is another way.
    #covariance_matrix = np.cov(parents, offspring)/np.cov(parents,parents)
    #return covariance_matrix[0,1]


heritability_scandens = heritability(bd_parent_scandens,
                                     bd_offspring_scandens)
heritability_fortis = heritability(bd_parent_fortis,
                                   bd_offspring_fortis)

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)

conf_int_scandens = np.percentile(replicates_scandens, [2.5, 97.5])
conf_int_fortis = np.percentile(replicates_fortis, [2.5, 97.5])

print('G. scandens:', heritability_scandens, conf_int_scandens)
print('G. fortis:', heritability_fortis, conf_int_fortis)
```

```
G. scandens: 0.5485340868685982 [0.35242481 0.75710845]
G. fortis: 0.7229051911438155 [0.64394464 0.79254422]
```

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.

```
In [28]: perm_replicates = np.empty(10000)

         for i in range(10000):
             bd_parent_permuted = np.random.permutation(bd_parent_scandens)
             perm_replicates[i] = heritability(bd_parent_permuted,bd_offspring_scandens)

         p = np.sum(perm_replicates >= heritability_scandens) / len(perm_replicates)

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

```
p-val = 0.0
```

```
In [ ]:
```