Reppeto530Week7

January 27, 2024

0.1 Chapter 7

Brian Reppeto 530 Prof. Jim Week 7 HW

[1]:
```
# import libraries

import numpy as np

import pandas as pd
import thinkstats2
import thinkplot
```

[2]:
```
# import first.py file and dropna's from agepreg and ttl wgt

import first

live, firsts, others = first.MakeFrames()
live = live.dropna(subset=['agepreg', 'totalwgt_lb'])
```

[3]:
```
# define the corr function to calc the corr coeff between to sets of data␣
↪expressed as arrays

def Corr(xs, ys):
    xs = np.asarray(xs)
    ys = np.asarray(ys)

    meanx, varx = thinkstats2.MeanVar(xs)
    meany, vary = thinkstats2.MeanVar(ys)

    corr = Cov(xs, ys, meanx, meany) / np.sqrt(varx * vary)
    return corr
```

[4]:
```
# define the cov function to calc the covariance between to sets of data␣
↪expressed as arrays

def Cov(xs, ys, meanx=None, meany=None):
    xs = np.asarray(xs)
    ys = np.asarray(ys)
```

```
    if meanx is None:
        meanx = np.mean(xs)
    if meany is None:
        meany = np.mean(ys)

    cov = np.dot(xs-meanx, ys-meany) / len(xs)
    return cov
```

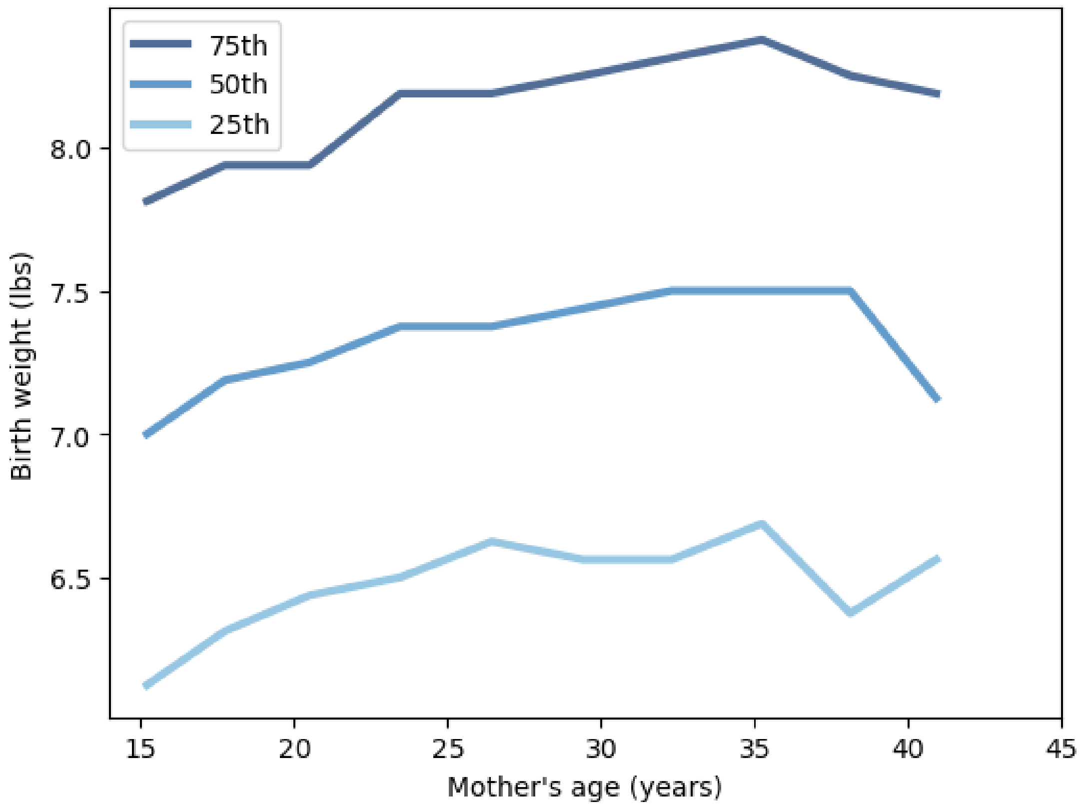
[5]:
```
# define the spearmancorr function to calc the sperman rank corr coeff between␣
↪to sets of data

def SpearmanCorr(xs, ys):
    xranks = pd.Series(xs).rank()
    yranks = pd.Series(ys).rank()
    return Corr(xranks, yranks)
```

[6]:
```
# use the corr & Spearmancorr functions to calc the corr coeff between age and␣
↪weights

ages = live.agepreg
weights = live.totalwgt_lb
print('Corr', Corr(ages, weights))
print('SpearmanCorr', SpearmanCorr(ages, weights))
```

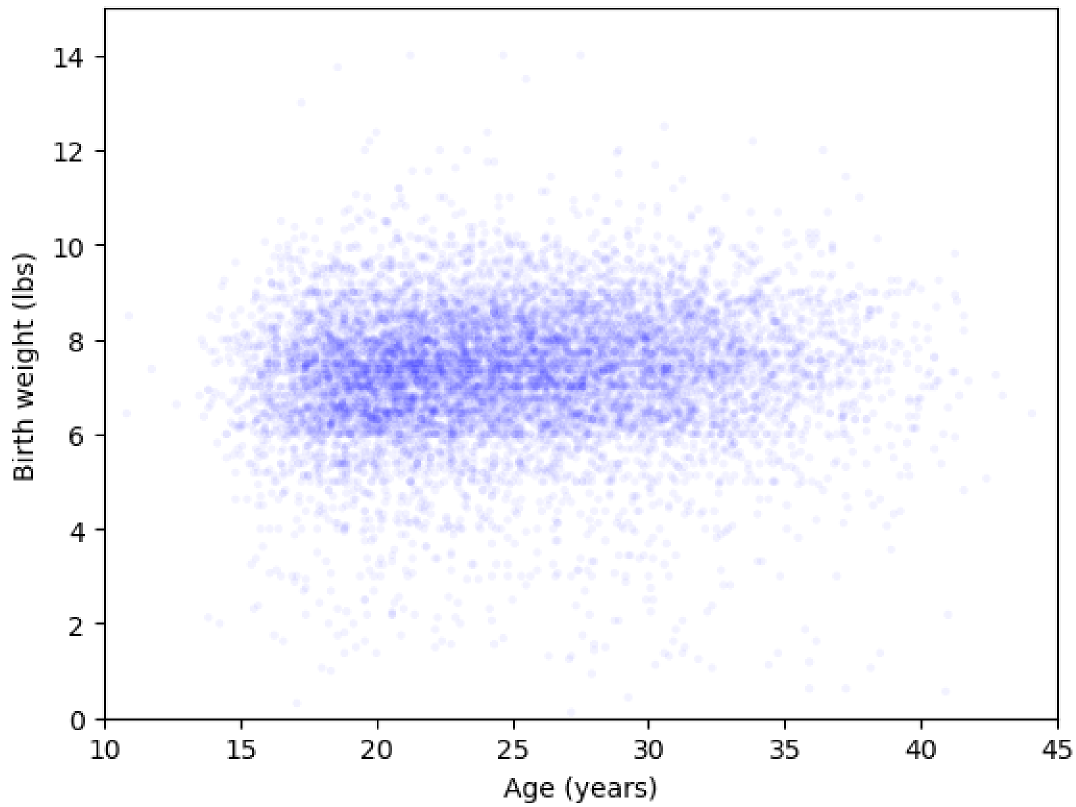
```
Corr 0.06883397035410882
SpearmanCorr 0.09461004109658226
```

[7]:
```
# define the binnedpercentiles function using NP to create a visualiztion of␣
↪percentiles (75,50,25) for
# the relationship between the ages of mothers and % birth weights across age␣
↪groups

def BinnedPercentiles(df):

    bins = np.arange(10, 48, 3)
    indices = np.digitize(df.agepreg, bins)
    groups = df.groupby(indices)

    ages = [group.agepreg.mean() for i, group in groups][1:-1]
    cdfs = [thinkstats2.Cdf(group.totalwgt_lb) for i, group in groups][1:-1]

    thinkplot.PrePlot(3)
    for percent in [75, 50, 25]:
        weights = [cdf.Percentile(percent) for cdf in cdfs]
        label = '%dth' % percent
        thinkplot.Plot(ages, weights, label=label)
```

```
    thinkplot.Config(xlabel="Mother's age (years)",
                     ylabel='Birth weight (lbs)',
                     xlim=[14, 45], legend=True)

BinnedPercentiles(live)
```



[8]:
```
# create a scatterplot of the relationship between ages of mothers and birth␣
↪weights

thinkplot.Scatter(ages, weights, alpha=0.05, s=10)
thinkplot.Config(xlabel='Age (years)',
                     ylabel='Birth weight (lbs)',
                     xlim=[10, 45],
                     ylim=[0, 15],
                     legend=False)
```



[9]:
```
# How woule you characterize the relationship between these variables?

# 1) Pearson's correlation is approximately 0.07, while Spearman's is around 0.
↪09. The disparity
# between them implies potential outlier influence or a non-linear relationship.

# 2) Examining percentiles of weight versus age reveals a non-linear␣
↪association. Birth weight demonstrates
# a more rapid increase within the mother's age range of 15 to 25. Beyond that,␣
↪the impact is not as defined.

# 3) The scatterplot depicts a subtle connection between the age and␣
↪birthweight.  However, the
# visibility is not clear.
```

Exercise 8-1

[10]:
```
# import library

import random
```

[11]:
```python
# create a function to calc the mean error between a list of estimates and the␣
↪corresponding actual values

def MeanError(estimates, actual):
    errors = [estimate-actual for estimate in estimates]
    return np.mean(errors)
```

[12]:
```python
# create a function to a simulation  an experiment generating random samples␣
↪from a normal distribution
# with mean (mu) 0 and standard deviation sigma 1

def Estimate(n=7, iters=100000):
    mu = 0
    sigma = 1

    means = []
    medians = []
    for _ in range(iters):
        xs = [random.gauss(mu, sigma) for i in range(n)]
        xbar = np.mean(xs)
        median = np.median(xs)
        means.append(xbar)
        medians.append(median)

    print('Experiment 1')
    print('mean error xbar', MeanError(means, mu))
    print('mean error median', MeanError(medians, mu))

Estimate()
```

```
Experiment 1
mean error xbar 0.0025231123253278754
mean error median 0.0020778403260392343
```

[13]:
```python
# create a function to calc the root mean squared error between a list of␣
↪estimated values
# and the corresponding actual values the accuracy of predictions or estimates

def RMSE(estimates, actual):

    e2 = [(estimate-actual)**2 for estimate in estimates]
    mse = np.mean(e2)
    return np.sqrt(mse)
```

[14]:
```python
#  create a function to estimate the variance of a normal distribution using␣
↪two different methods:
```

```
# biased and unbiased variance estimators. The root mean squared error (RMSE)␣
↪between the estimated
# variances and the true variance is then calculated.

def Estimate2(n=7, iters=100000):

    mu = 0
    sigma = 1

    estimates1 = []
    estimates2 = []
    for _ in range(iters):
        xs = [random.gauss(mu, sigma) for i in range(n)]
        biased = np.var(xs)
        unbiased = np.var(xs, ddof=1)
        estimates1.append(biased)
        estimates2.append(unbiased)

    print('Experiment 2')
    print('RMSE biased', RMSE(estimates1, sigma**2))
    print('RMSE unbiased', RMSE(estimates2, sigma**2))

Estimate2()
```

```
Experiment 2
RMSE biased 0.5152344523990781
RMSE unbiased 0.5768632975185001
```

[15]:
```
# 1. As the sample size increases, both the sample mean and median consistently␣
↪exhibit
# reduced mean errors. Consequently, there is no apparent bias observed in␣
↪either estimator based
# on the conducted experiment.

# 2. The biased variance estimator consistently demonstrates a lower RMSE
# compared to the unbiased estimator. The magnitude of this difference remains␣
↪stable, showing
# approximately a 10% reduction for the biased estimator. This trend persists␣
↪with increasing sample size.
```

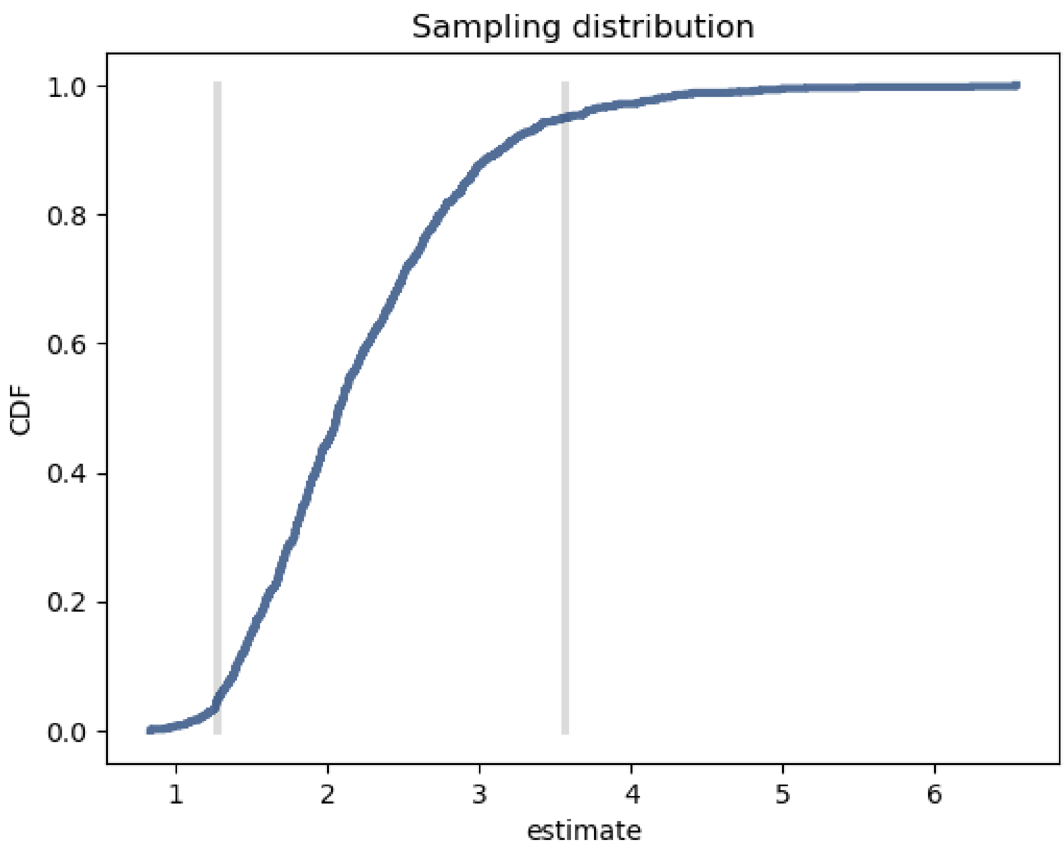
Exercise 8-2

[16]:
```
# create a function to estimate the rate parameter (lambda) of an exponential␣
↪distribution
# The simulation generates multiple samples of size n from an exponential␣
↪distribution with
# a known lambda. It then calculates the mean of each sample and estimates
```

```
# lambda using the reciprocal of the sample mean. For this function I used a␣
↪size of 10

def SimulateSample(lam=2, n=10, iters=1000):

    def VertLine(x, y=1):
        thinkplot.Plot([x, x], [0, y], color='0.8', linewidth=3)

    estimates = []
    for _ in range(iters):
        xs = np.random.exponential(1.0/lam, n)
        lamhat = 1.0 / np.mean(xs)
        estimates.append(lamhat)

    stderr = RMSE(estimates, lam)
    print('standard error', stderr)

    cdf = thinkstats2.Cdf(estimates)
    ci = cdf.Percentile(5), cdf.Percentile(95)
    print('confidence interval', ci)
    VertLine(ci[0])
    VertLine(ci[1])

    # plot the CDF
    thinkplot.Cdf(cdf)
    thinkplot.Config(xlabel='estimate',
                     ylabel='CDF',
                     title='Sampling distribution')

    return stderr

SimulateSample()
```

```
standard error 0.7811292923639753
confidence interval (1.279072833508718, 3.562764916541147)
```

```
[16]: 0.7811292923639753
```

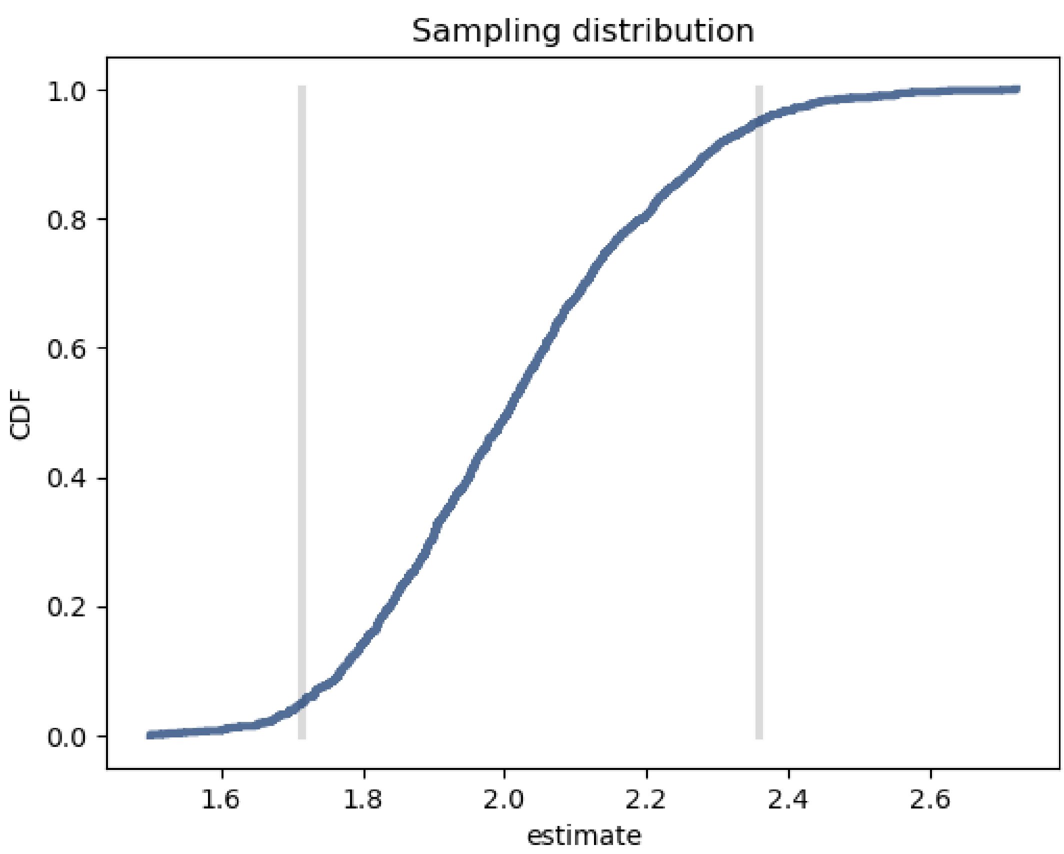


[20]:
```
# create a function to estimate the rate parameter (lambda) of an exponential␣
↪distribution
# The simulation generates multiple samples of size n from an exponential␣
↪distribution with
# a known lambda. It then calculates the mean of each sample and estimates
# lambda using the reciprocal of the sample mean. For this function I used a␣
↪size of 100

def SimulateSample1(lam=2, n=100, iters=1000):

    def VertLine(x, y=1):
        thinkplot.Plot([x, x], [0, y], color='0.8', linewidth=3)

    estimates = []
    for _ in range(iters):
        xs = np.random.exponential(1.0/lam, n)
        lamhat = 1.0 / np.mean(xs)
        estimates.append(lamhat)
```

```
    stderr = RMSE(estimates, lam)
    print('standard error', stderr)

    cdf = thinkstats2.Cdf(estimates)
    ci = cdf.Percentile(5), cdf.Percentile(95)
    print('confidence interval', ci)
    VertLine(ci[0])
    VertLine(ci[1])

    # plot the CDF
    thinkplot.Cdf(cdf)
    thinkplot.Config(xlabel='estimate',
                     ylabel='CDF',
                     title='Sampling distribution')

    return stderr

SimulateSample1()
```

```
standard error 0.20143914367675628
confidence interval (1.7146963619598594, 2.361003153832125)
```

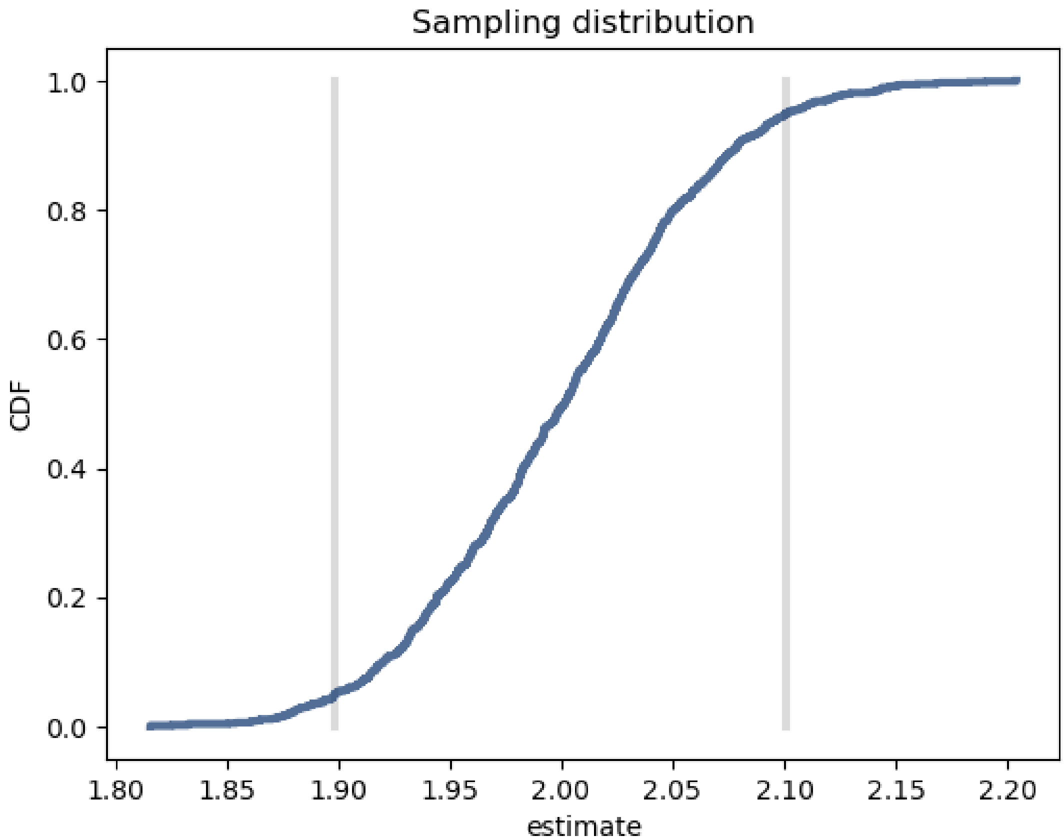
[20]: 0.20143914367675628



[21]:
```
# create a function to estimate the rate parameter (lambda) of an exponential␣
↪distribution
# The simulation generates multiple samples of size n from an exponential␣
↪distribution with
# a known lambda. It then calculates the mean of each sample and estimates
# lambda using the reciprocal of the sample mean. For this function I used a␣
↪size of 1000

def SimulateSample1(lam=2, n=1000, iters=1000):

    def VertLine(x, y=1):
        thinkplot.Plot([x, x], [0, y], color='0.8', linewidth=3)

    estimates = []
    for _ in range(iters):
        xs = np.random.exponential(1.0/lam, n)
        lamhat = 1.0 / np.mean(xs)
        estimates.append(lamhat)

    stderr = RMSE(estimates, lam)
    print('standard error', stderr)

    cdf = thinkstats2.Cdf(estimates)
    ci = cdf.Percentile(5), cdf.Percentile(95)
    print('confidence interval', ci)
    VertLine(ci[0])
    VertLine(ci[1])

    # plot the CDF
    thinkplot.Cdf(cdf)
    thinkplot.Config(xlabel='estimate',
                     ylabel='CDF',
                     title='Sampling distribution')

    return stderr

SimulateSample1()
```

```
standard error 0.061880399210896164
confidence interval (1.8983709137797495, 2.100987806449984)
```

[21]:
```
0.061880399210896164
```



[18]:
```
# the sample size 10 has a
# standard error 0.7811292923639753 and
# confidence interval (1.2790728335087l8, 3.562764916541147)


# the sample size 100 has a
# standard error 0.2014391436767S628 and
# confidence interval (1.7146963619598594, 2.361003153832125)


# the sample size 1000 has a
# standard error 0.061880399210896164 and
# confidence interval (1.8983709137797495, 2.100987806449984)
```

[ ]: