

Python Code for QSS Chapter 7: Uncertainty

Kosuke Imai, Python code by Jeff Allen

First Printing

```
[ ]: import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import seaborn as sns
```

Section 7.1: Estimation

Section 7.1.1: Unbiasedness and Consistency

```
[ ]: # simulation parameters
n = 100 # sample size
mu0 = 0 # mean of  $Y_i(0)$ 
sd0 = 1 # standard deviation of  $Y_i(0)$ 
mu1 = 1 # mean of  $Y_i(1)$ 
sd1 = 1 # standard deviation of  $Y_i(1)$ 

# generate a sample
Y0 = stats.norm.rvs(size=n, loc=mu0, scale=sd0)
Y1 = stats.norm.rvs(size=n, loc=mu1, scale=sd1)
tau = Y1 - Y0 # individual treatment effect
# true value of the sample average treatment effect
SATE = tau.mean()
SATE
```

```
[ ]: 0.9774441662972356
```

```
[ ]: # repeatedly conduct randomized controlled trials
sims = 5000 # repeat 5,000 times, we could do more
diff_means = np.zeros(sims) # container
sample_vector = np.concatenate((np.ones(int(n/2)), np.zeros(int(n/2))))

for i in range(sims):
    # randomize the treatment by sampling of a vector of 0's and 1's
    treat = np.random.choice(sample_vector, size=n, replace=False)
    # difference-in-means
    diff_means[i] = Y1[treat==1].mean() - Y0[treat==0].mean()
```

```
# estimation of error for SATE
est_error = diff_means - SATE

est_error.mean()
```

```
[ ]: -0.00033103445440258773
```

```
[ ]: pd.Series(est_error).describe().round(5)
```

```
[ ]: count    5000.00000
      mean      -0.00033
      std       0.13850
      min      -0.43851
      25%      -0.09571
      50%      -0.00059
      75%       0.09296
      max       0.48181
      dtype: float64
```

```
[ ]: # PATE simulation
PATE = mu1 - mu0
diff_means = np.zeros(sims)

for i in range(sims):
    # generate a sample for each simulation
    Y0 = stats.norm.rvs(size=n, loc=mu0, scale=sd0)
    Y1 = stats.norm.rvs(size=n, loc=mu1, scale=sd1)
    treat = np.random.choice(sample_vector, size=n, replace=False)
    diff_means[i] = Y1[treat==1].mean() - Y0[treat==0].mean()

# estimation error for PATE
est_error = diff_means - PATE

# unbiased
est_error.mean()
```

```
[ ]: -0.0038793719921513016
```

```
[ ]: pd.Series(est_error).describe().round(5)
```

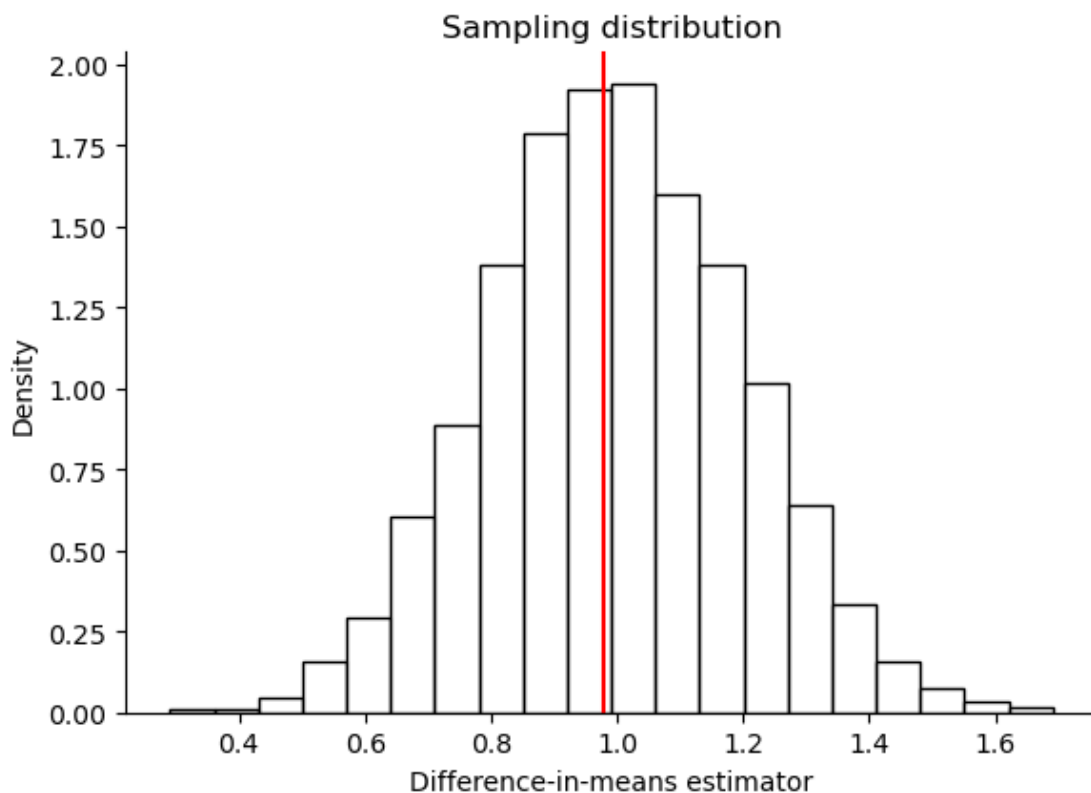
```
[ ]: count    5000.00000
      mean     -0.00388
      std      0.20088
      min     -0.70899
      25%     -0.14017
      50%     -0.00700
      75%      0.13517
      max      0.69117
```

dtype: float64

Section 7.1.2: Standard Error

```
[ ]: sns.displot(  
    diff_means, stat='density', color='white', edgecolor='black',  
    height=4, aspect=1.5, bins=20  
) .set(title='Sampling distribution', xlabel='Difference-in-means estimator')  
  
plt.axvline(SATE, color='red') # true value of SATE
```

```
[ ]: <matplotlib.lines.Line2D at 0x230fa977bb0>
```



```
[ ]: diff_means.std(ddof=1)
```

```
[ ]: 0.2008806087644372
```

```
[ ]: np.sqrt(((diff_means - SATE)**2).mean())
```

```
[ ]: 0.20172694067923444
```

```
[ ]: # PATE simulation with standard error
sims = 5000
diff_means = np.zeros(sims)
se = np.zeros(sims)

for i in range(sims):
    # generate a sample for each simulation
    Y0 = stats.norm.rvs(size=n, loc=mu0, scale=sd0)
    Y1 = stats.norm.rvs(size=n, loc=mu1, scale=sd1)
    # randomize treatment by sampling the vector of 0's and 1's created above
    treat = np.random.choice(sample_vector, size=n, replace=False)
    diff_means[i] = Y1[treat==1].mean() - Y0[treat==0].mean()
    se[i] = (np.sqrt(Y1[treat==1].var(ddof=1) / (n/2) +
                    Y0[treat==0].var(ddof=1) / (n/2)))

diff_means.std(ddof=1)
```

```
[ ]: 0.19898811338479394
```

```
[ ]: se.mean()
```

```
[ ]: 0.19944482309407496
```

Section 7.1.3: Confidence Intervals

```
[ ]: n = 1000 # sample size
x_bar = 0.6 # point estimate
s_e = np.sqrt(x_bar * (1-x_bar) / n) # standard error

# 99% confidence intervals; display as a tuple
((x_bar - stats.norm.ppf(0.995) * s_e).round(5),
 (x_bar + stats.norm.ppf(0.995) * s_e).round(5))
```

```
[ ]: (0.5601, 0.6399)
```

```
[ ]: # 95% confidence intervals
((x_bar - stats.norm.ppf(0.975) * s_e).round(5),
 (x_bar + stats.norm.ppf(0.975) * s_e).round(5))
```

```
[ ]: (0.56964, 0.63036)
```

```
[ ]: # 90% confidence intervals
((x_bar - stats.norm.ppf(0.95) * s_e).round(5),
 (x_bar + stats.norm.ppf(0.95) * s_e).round(5))
```

```
[ ]: (0.57452, 0.62548)
```

```
[ ]: # empty container matrices for 2 sets of confidence intervals
ci95 = np.zeros(sims*2).reshape(sims, 2)
ci90 = np.zeros(sims*2).reshape(sims, 2)

# 95 percent confidence intervals
ci95[:,0] = diff_means - stats.norm.ppf(0.975) * se # lower limit
ci95[:,1] = diff_means + stats.norm.ppf(0.975) * se # upper limit

# 90 percent confidence intervals
ci90[:,0] = diff_means - stats.norm.ppf(0.95) * se # lower limit
ci90[:,1] = diff_means + stats.norm.ppf(0.95) * se # upper limit

# coverage rate for 95% confidence interval
((ci95[:,0] <= 1) & (ci95[:,1] >= 1)).mean()
```

```
[ ]: 0.9506
```

```
[ ]: # coverage rate for 90% confidence interval
((ci90[:,0] <= 1) & (ci90[:,1] >= 1)).mean()
```

```
[ ]: 0.8966
```

```
[ ]: p = 0.6 # true parameter value
n = np.array([50, 100, 1000]) # 3 sample sizes to be examined
alpha = 0.05
sims = 5000 # number of simulations
results = np.zeros(len(n)) # a container for results

for i in range(len(n)):
    ci_results = np.zeros(sims) # a container for whether CI contains truth
    # loop for repeated hypothetical survey sampling
    for j in range(sims):
        data = stats.binom.rvs(n=1, p=p, size=n[i]) # simple random sampling
        x_bar = data.mean() # sample proportion as an estimate
        s_e = np.sqrt(x_bar * (1-x_bar) / n[i]) # standard errors
        ci_lower = x_bar - stats.norm.ppf(1-alpha/2) * s_e
        ci_upper = x_bar + stats.norm.ppf(1-alpha/2) * s_e
        ci_results[j] = (p >= ci_lower) & (p <= ci_upper)
    # proportion of CIs that contain the true value
    results[i] = ci_results.mean()

results
```

```
[ ]: array([0.943 , 0.9488, 0.9518])
```

Section 7.1.4: Margin of Error and Sample Size Calculation in Polls

```
[ ]: MoE = np.array([0.01, 0.03, 0.05]) # the desired margin of error
p = np.arange(0.01, 1, 0.01)
n = 1.96**2 * p * (1-p) / MoE[0]**2
n2 = 1.96**2 * p * (1-p) / MoE[1]**2
n3 = 1.96**2 * p * (1-p) / MoE[2]**2

fig, ax = plt.subplots(figsize=(6,4))

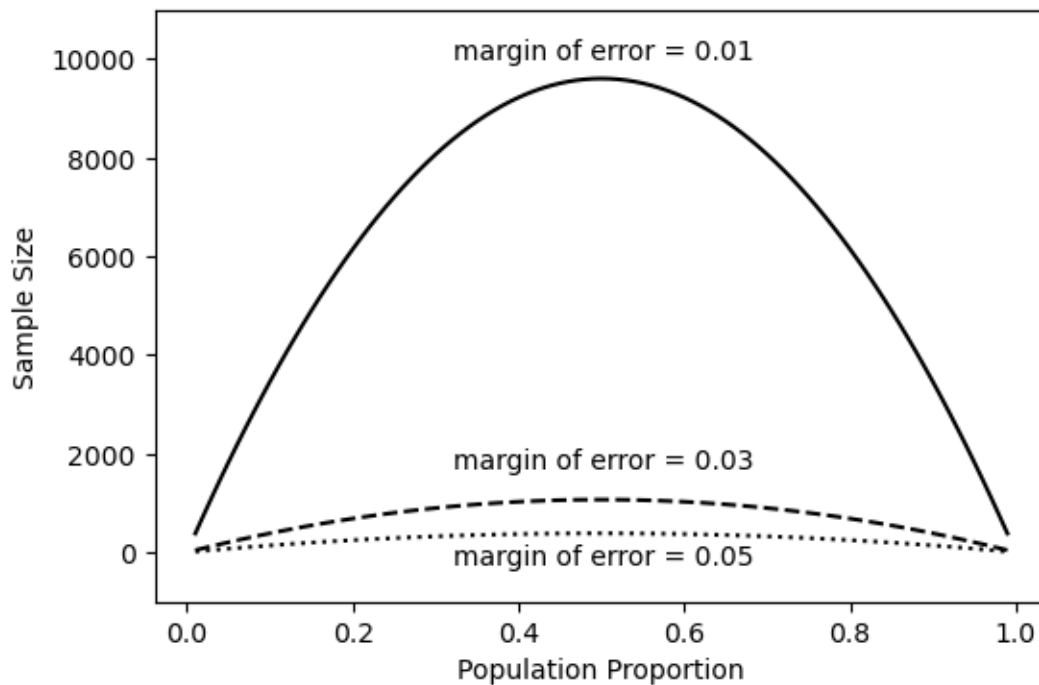
sns.lineplot(x=p, y=n, ax=ax, color='black').set(
    ylim=(-1000, 11000), xlabel='Population Proportion', ylabel='Sample Size'
)

sns.lineplot(x=p, y=n2, ax=ax, color='black', linestyle='--')

sns.lineplot(x=p, y=n3, ax=ax, color='black', linestyle=':')

# Add text labels
ax.text(0.32, 10000, 'margin of error = 0.01', fontsize=10)
ax.text(0.32, 1700, 'margin of error = 0.03', fontsize=10)
ax.text(0.32, -250, 'margin of error = 0.05', fontsize=10)
```

```
[ ]: Text(0.32, -250, 'margin of error = 0.05')
```



```
[ ]: # election and polling results, by state
pres08 = pd.read_csv('pres08.csv')
polls08 = pd.read_csv('polls08.csv')

# convert to a date object
polls08['middate'] = pd.to_datetime(polls08['middate'])

# number of days to the election
from datetime import datetime
election_day = datetime.strptime('2008-11-04', '%Y-%m-%d')
polls08['days_to_election'] = (election_day - polls08['middate']).dt.days

# extract unique state names which the loop will iterate through
st_names = polls08['state'].unique()

# create an empty 51 X 3 placeholder Data Frame
poll_pred = pd.DataFrame(np.zeros(51*3).reshape(51, 3), index=st_names)

# loop across the 50 states plus DC
for i in range(len(st_names)):
    # subset the ith state
    state_data = polls08[polls08['state']==st_names[i]]
    # further subset the latest polls within the state
    latest = (state_data['days_to_election']==
              state_data['days_to_election'].min())
    # compute the mean of the latest polls and store it
    poll_pred.iloc[i, 0] = state_data['Obama'][latest].mean() / 100

# upper and lower confidence limits
n = 1000 # sample size
alpha = 0.05
se = np.sqrt(poll_pred.iloc[:,0] * (1-poll_pred.iloc[:,0]) / n) # standard error
poll_pred.iloc[:,1] = poll_pred.iloc[:,0] - stats.norm.ppf(1-alpha/2) * se
poll_pred.iloc[:,2] = poll_pred.iloc[:,0] + stats.norm.ppf(1-alpha/2) * se

[ ]: # plot the results
fig, ax = plt.subplots(figsize=(6,4))

sns.scatterplot(
    x = pres08['Obama'] / 100, y = poll_pred.iloc[:,0].reset_index(drop=True),
    ax=ax, color='white', edgecolor='black'
).set(xlabel="Obama's vote share", ylabel='Poll prediction',
      xlim=(0, 1), ylim=(0, 1))

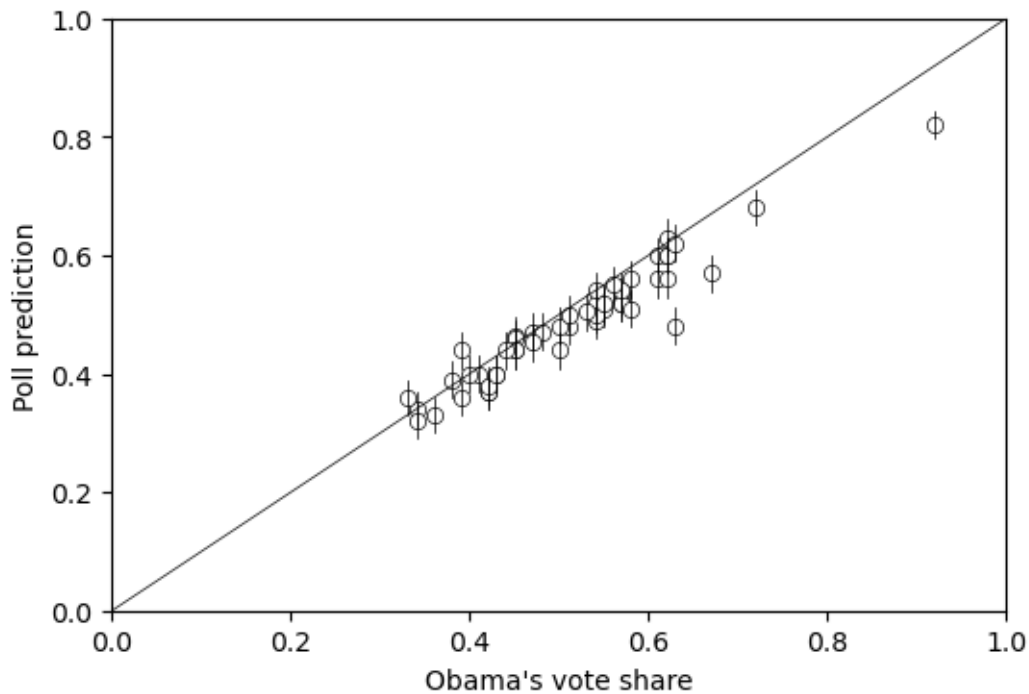
ax.axline((0, 0), slope=1, color='black', linewidth=0.5)

# adding 95% confidence intervals for each state
```

```

for i in range(len(st_names)):
    ax.plot(
        [pres08['Obama'][i] / 100] * 2,
        [poll_pred.iloc[i,1], poll_pred.iloc[i,2]],
        color='black', linewidth=0.5
    )

```



```

[ ]: # proportion of confidence intervals that contain the election day outcome
# reset index: can only compare identically-labeled Series objects
((poll_pred.iloc[:,1].reset_index(drop=True) <= pres08['Obama'] / 100) &
 (poll_pred.iloc[:,2].reset_index(drop=True) >= pres08['Obama'] / 100)).mean()

```

```

[ ]: 0.5882352941176471

```

```

[ ]: # bias
bias=(poll_pred.iloc[:,0].reset_index(drop=True) - pres08['Obama']/100).mean()
bias

```

```

[ ]: -0.026797385620915028

```

```

[ ]: # bias corrected estimate
poll_bias = poll_pred.iloc[:,0] - bias

# bias corrected standard error

```



```

se_bias = np.sqrt(poll_bias * (1-poll_bias) / n)

# bias corrected confidence intervals
ci_bias_lower = poll_bias - stats.norm.ppf(1-alpha/2) * se_bias
ci_bias_upper = poll_bias + stats.norm.ppf(1-alpha/2) * se_bias

# proportion of bias corrected CIs that contain election day outcome
((ci_bias_lower.reset_index(drop=True) <= pres08['Obama'] / 100) &
 (ci_bias_upper.reset_index(drop=True) >= pres08['Obama'] / 100)).mean()

```

```
[ ]: 0.7647058823529411
```

Section 7.1.5: Analysis of Randomized Controlled Trials

In Progress