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
```

[ ]: 1.0016257277464635

[ ]:
```
# 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()
```

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

est_error.mean()
```

[ ]: -0.001826897294085793

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

[ ]:
```
count    5000.00000
mean       -0.00183
std         0.14876
min        -0.53608
25%        -0.10447
50%        -0.00051
75%         0.09936
max         0.52945
dtype: float64
```

[ ]:
```python
# 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()
```

[ ]: -0.0016801318610299334

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

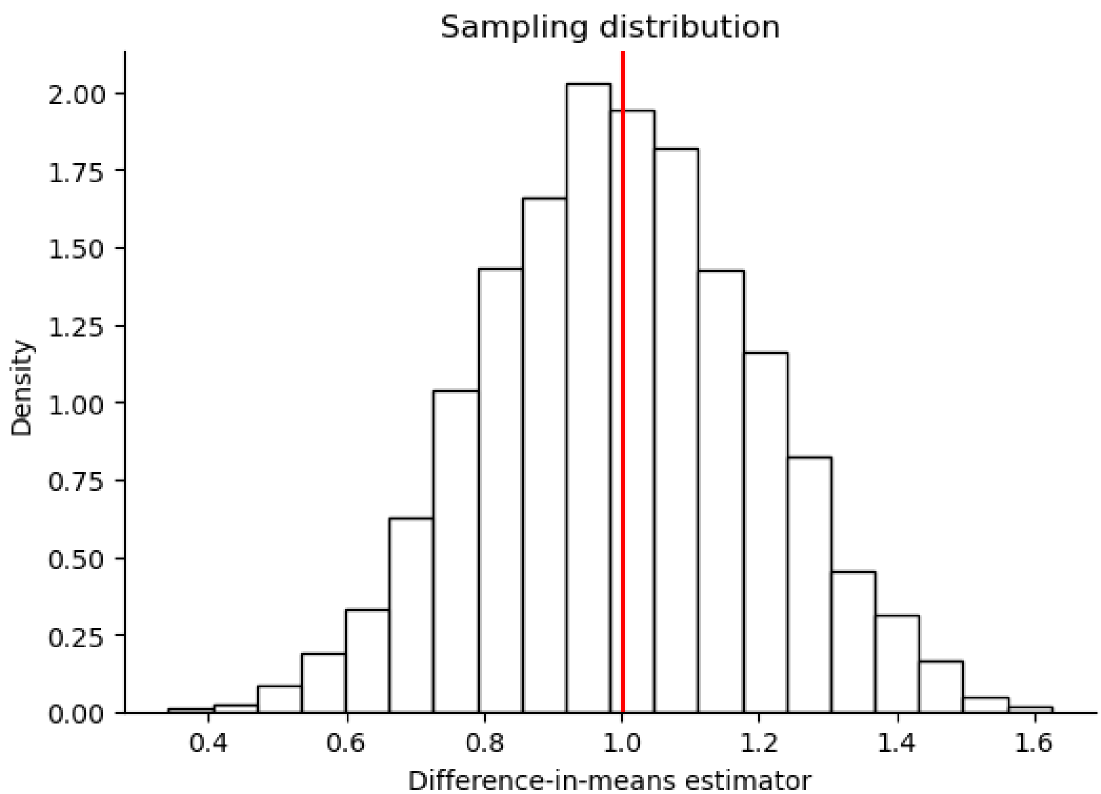
[ ]:
```
count    5000.00000
mean       -0.00168
std         0.19679
min        -0.65788
25%        -0.13752
50%        -0.00548
75%         0.13323
max         0.62456
```

```
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 0x20bb68f8640>
```



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

[ ]:
```
0.19679000287560444
```

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

[ ]:
```
0.19679809114541358
```

[ ]:
```
# 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)
```

[ ]: 0.2002686905359311

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

[ ]: 0.19964814031468717

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()
```

[ ]: 0.9502

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

[ ]: 0.8956

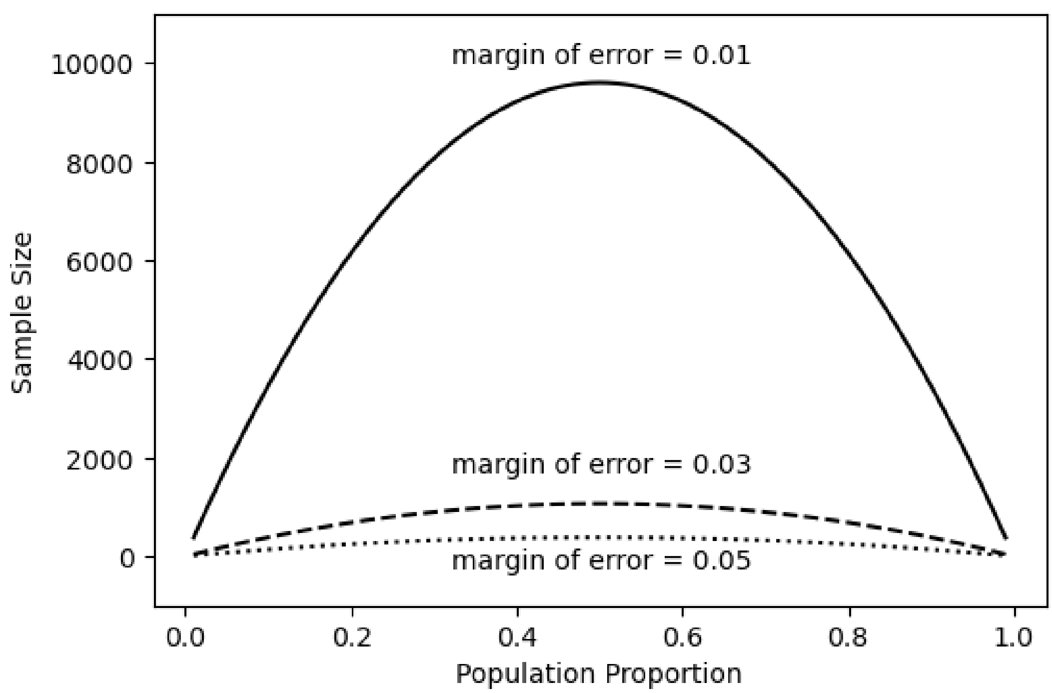
[ ]:
```
p = 0.6 # true parameter value
n = np.array([50, 100, 1000]) # 3 sample sizes to be examined
alpha = 0.05
sims = 5000 # number of simulations
results = np.zeros(len(n)) # a container for results

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([0.9372, 0.9514, 0.9416])

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)
```

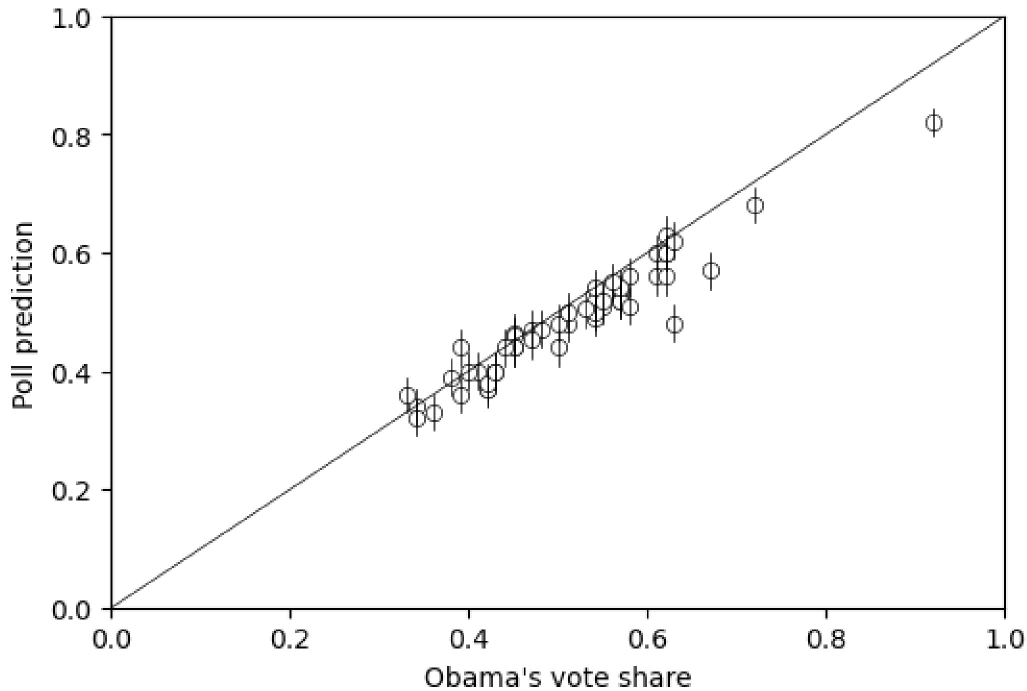
[ ]:
```
Text(0.32, -250, 'margin of error = 0.05')
```



[ ]:
```python
# 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
```

[ ]:
```python
# 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
```

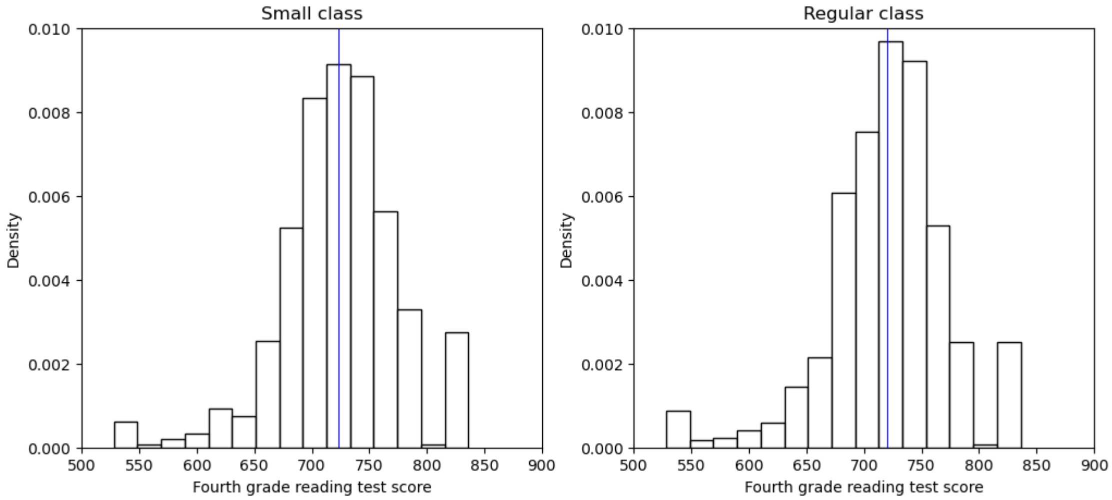
```
se_bias = np.sqrt(poll_bias * (1-poll_bias) / n)

# 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

[ ]:
```
STAR = pd.read_csv('STAR.csv')

fig, axs = plt.subplots(1, 2, figsize=(12,5))

sns.histplot(
    STAR['g4reading'][STAR.classtype==1], stat = 'density', ax=axs[0],
    color='white', edgecolor='black', bins=15
).set(ylim=(0, 0.01), xlim=(500, 900), title='Small class',
      xlabel='Fourth grade reading test score')

axs[0].axvline(STAR['g4reading'][STAR.classtype==1].mean(),
               color='blue', linewidth=0.75)

sns.histplot(
    STAR['g4reading'][STAR.classtype==2], stat = 'density', ax=axs[1],
    color='white', edgecolor='black', bins=15
).set(ylim=(0, 0.01), xlim=(500, 900), title='Regular class',
      xlabel='Fourth grade reading test score')

axs[1].axvline(STAR['g4reading'][STAR.classtype==2].mean(),
                color='blue', linewidth=0.75)
```

[ ]: <matplotlib.lines.Line2D at 0x20bb83655a0>



[ ]:
```
# estimate and standard error for small class size
n_small = (STAR['classtype']==1 & STAR['g4reading'].notnull()).sum()
est_small = STAR['g4reading'][STAR.classtype==1].mean()
se_small = STAR['g4reading'][STAR.classtype==1].std() / np.sqrt(n_small)
est_small, se_small
```

[ ]: (723.3911845730028, 1.9130122952458233)

[ ]:
```
# estimate and standard error for regular class size
n_regular = ((STAR['classtype']==2) &
             (STAR['classtype'].notnull()) &
             (STAR['g4reading'].notnull())).sum()
est_regular = STAR['g4reading'][STAR.classtype==2].mean()
se_regular = STAR['g4reading'][STAR.classtype==2].std() / np.sqrt(n_regular)
est_regular, se_regular
```

[ ]: (719.88995215311, 1.8388496908502467)

[ ]:
```
alpha = 0.05

# 95% confidence intervals for small class size
ci_small = (est_small - stats.norm.ppf(1-alpha/2) * se_small,
            est_small + stats.norm.ppf(1-alpha/2) * se_small)
ci_small
```

[ ]: (719.6417493723386, 727.1406197736669)

[ ]:
```
# 95% confidence intervals for regular class size
ci_regular = (est_regular - stats.norm.ppf(1-alpha/2) * se_regular,
              est_regular + stats.norm.ppf(1-alpha/2) * se_regular)
```

```
ci_regular
```

[ ]: (716.2858729860609, 723.4940313201591)

[ ]:
```
# difference in means estimator
ate_est = est_small - est_regular
ate_est
```

[ ]: 3.5012324198927445

[ ]:
```
# standard error and 95% confidence interval
ate_se = np.sqrt(se_small**2 + se_regular**2)
ate_se
```

[ ]: 2.653485298112982

[ ]:
```
ate_ci = (ate_est - stats.norm.ppf(1-alpha/2) * ate_se,
          ate_est + stats.norm.ppf(1-alpha/2) * ate_se)
ate_ci
```

[ ]: (-1.699503197915229, 8.701968037700718)

Section 7.1.6: Analysis Based on Student's t-Distribution

[ ]:
```
# 95% CI for small class
(est_small - stats.t.ppf(0.975, df=n_small-1) * se_small,
 est_small + stats.t.ppf(0.975, df=n_small-1) * se_small)
```

[ ]: (719.635479522832, 727.1468896231735)

[ ]:
```
# 95% CI based on the central limit theorem
ci_small
```

[ ]: (719.6417493723386, 727.1406197736669)

[ ]:
```
# 95% CI for regular class
(est_regular - stats.t.ppf(0.975, df=n_regular-1) * se_regular,
 est_regular + stats.t.ppf(0.975, df=n_regular-1) * se_regular)
```

[ ]: (716.2806412822123, 723.4992630240077)

[ ]:
```
# 95% CI based on the central limit theorem
ci_regular
```

[ ]: (716.2858729860609, 723.4940313201591)

[ ]:
```
test_result = stats.ttest_ind(
    STAR['g4reading'][STAR.classtype==1],
```

```
    STAR['g4reading'][STAR.classtype==2],
    # override default equal_var=True; False is Welch t-test
    equal_var=False,
    # override default nan_policy='propagate'
    nan_policy='omit')

# we can extract results from the test_result object
test_result.pvalue
```

[ ]: 0.18720319784556907

[ ]:
```
# find the confidence interval
ci = test_result.confidence_interval(confidence_level=0.95)

# print summary of results
print(
    'Welch Two Sample t-test\n'
    f"t-stat: {test_result.statistic:.4f}\n"
    f"p-value: {test_result.pvalue:.4f}\n"
    f"df: {test_result.df:.1f}\n"
    f"95% confidence interval: ({ci[0]:.4f}, {ci[1]:.4f})"
)
```

```
Welch Two Sample t-test
t-stat: 1.3195
p-value: 0.1872
df: 1541.2
95% confidence interval: (-1.7036, 8.7061)
```

Section 7.2: Hypothesis Testing

Section 7.2.1: Tea-Testing Experiment

[ ]:
```
from math import comb

# truth: enumerate the number of assignment combinations
true = np.array(
    [comb(4,0) * comb(4,4),
     comb(4,1) * comb(4,3),
     comb(4,2) * comb(4,2),
     comb(4,3) * comb(4,1),
     comb(4,4) * comb(4,0)]
)

true
```

[ ]: array([1, 16, 36, 16, 1])

[ ]:
```
# compute probability: divide it by the total number of events
true = pd.Series(true / true.sum(), index=[0,2,4,6,8])

true
```

[ ]:
```
0    0.014286
2    0.228571
4    0.514286
6    0.228571
8    0.014286
dtype: float64
```

[ ]:
```
# simulations
sims=1000
# lady's guess: M stands for 'Milk first', T stands for 'Tea first'
guess=np.array(['M', 'T', 'T', 'M', 'M', 'T', 'T', 'M'])
sample_vector=np.array(['T']*4 + ['M']*4)
correct=pd.Series(np.zeros(sims)) # place holder for number of correct guesses

for i in range(sims):
    # randomize which cups get Milk/Tea first
    cups=np.random.choice(sample_vector, size=len(sample_vector), replace=False)
    correct[i]=(guess==cups).sum() # number of correct guesses

# estimated probability for each number of correct guesses
correct.value_counts(normalize=True).sort_index()
```

[ ]:
```
0.0    0.014
2.0    0.258
4.0    0.514
6.0    0.202
8.0    0.012
Name: proportion, dtype: float64
```

[ ]:
```
# comparison with analytical answers; the differences are small
correct.value_counts(normalize=True).sort_index() - true
```

[ ]:
```
0.0   -0.000286
2.0    0.029429
4.0   -0.000286
6.0   -0.026571
8.0   -0.002286
dtype: float64
```

Section 7.2.2: The General Framework

[ ]:
```
# all correct
x = pd.DataFrame({'M': [4,0], 'T': [0,4]}, index=['M','T'])
# six correct
y = pd.DataFrame({'M': [3,1], 'T': [1,3]}, index=['M','T'])

x
```

[ ]:

	M	T
M	4	0
T	0	4

[ ]:
```
y
```

[ ]:

	M	T
M	3	1
T	1	3

[ ]:
```
# one-sided test for 8 correct guesses
fisher_one=stats.fisher_exact(x.values, alternative='greater')

print(
    "Fisher's Exact Test for Count Data: One-Sided\n"
    f"P-value: {fisher_one.pvalue:.5f}"
)
```

```
Fisher's Exact Test for Count Data: One-Sided
P-value: 0.01429
```

[ ]:
```
# two-sided test for 6 correct guesses
fisher_two=stats.fisher_exact(y.values)

print(
    "Fisher's Exact Test for Count Data: Two-Sided\n"
    f"P-value: {fisher_two.pvalue:.5f}"
)
```

```
Fisher's Exact Test for Count Data: Two-Sided
P-value: 0.48571
```

Section 7.2.3: One-Sample Tests

[ ]:
```
n = 1018
x_bar = 550 / n
se = np.sqrt(0.5 * 0.5 / n) # standard deviation of sampling distribution

# upper red area in the figure
upper = 1 - stats.norm.cdf(x_bar, loc=0.5, scale=se)
```

```
# lower red area in the figure; identical to the upper area
lower = stats.norm.cdf(0.5 - (x_bar - 0.5), loc=0.5, scale=se)

# two-sided p-value
upper + lower
```

[ ]: 0.010168663287718531

[ ]:
```
2 * upper
```

[ ]: 0.010168663287718482

[ ]:
```
# one-sided p-value
upper
```

[ ]: 0.005084331643859241

[ ]:
```
z_score = (x_bar - 0.5) / se
z_score
```

[ ]: 2.5700404773096097

[ ]:
```
# one-sided p-value
1 - stats.norm.cdf(z_score)
```

[ ]: 0.005084331643859241

[ ]:
```
# two-sided p-value
2 * (1 - stats.norm.cdf(z_score))
```

[ ]: 0.010168663287718482

[ ]:
```
# 99% confidence interval contains 0.5
(x_bar - stats.norm.ppf(0.995) * se, x_bar + stats.norm.ppf(0.995) * se)
```

[ ]: (0.499909283428347, 0.58064081480348)

[ ]:
```
# 95% confidence interval does not contain 0.5
(x_bar - stats.norm.ppf(0.975) * se, x_bar + stats.norm.ppf(0.975) * se)
```

[ ]: (0.5095604956138589, 0.5709896026179682)

[ ]:
```
from statsmodels.stats.proportion import proportions_ztest, proportion_confint

# no continuity correction to get the same p-value as above
stat, pval = proportions_ztest(count=550, nobs=n, value=0.5, prop_var=0.5)
ci = proportion_confint(count=550, nobs=n)
```

```
print(
    'One-sample z-test without continuity correction\n'
    'Alternative hypothesis: true p is not equal to 0.5\n'
    f"Sample proportion: {x_bar:.4f}\n"
    f"Test statistic: {stat:.4f}\n"
    f"P-value: {pval:.4f}\n"
    f"95% confidence interval: ({ci[0]:.4f}, {ci[1]:.4f})"
)
```

```
One-sample z-test without continuity correction
Alternative hypothesis: true p is not equal to 0.5
Sample proportion: 0.5403
Test statistic: 2.5700
P-value: 0.0102
95% confidence interval: (0.5097, 0.5709)
```

The continuity correction factor subtracts 0.5 from the sample proportion before computing the test statistic. The correction factor is not built into the one-sample z-tests available in Python. However, we can build a function to implement the correction using the hypothesis testing logic we have developed.

[ ]:
```
# Define a function to implement the continuity correction factor
def proportions_ztest_correct(count, nobs, value, conf_level=0.95):

    # compute the p-value
    prop = count / nobs
    correction = 0.5 / nobs # Yates' continuity correction
    adjusted_prop = prop - correction
    se_null = np.sqrt(value * (1-value) / nobs) # SE under the null hypothesis
    z_score = np.abs(adjusted_prop-value) / se_null
    # assume a two-tailed test, but we could generalize this
    pval = 2 * (1 - stats.norm.cdf(z_score))

    # compute the confidence interval
    se_sample = np.sqrt(adjusted_prop * (1-adjusted_prop) / nobs)
    alpha = 1-conf_level
    ci_lower = adjusted_prop - stats.norm.ppf(1-alpha/2) * se_sample
    ci_upper = adjusted_prop + stats.norm.ppf(1-alpha/2) * se_sample
    conf_print = int(conf_level * 100)

    print(
        'One-sample z-test with continuity correction\n'
        f"Alternative hypothesis: true p is not equal to {value}\n"
        f"Sample proportion: {prop:.4f}\n"
        f"Test statistic: {z_score:.4f}\n"
        f"P-value: {pval:.4f}\n"
        f"{conf_print}% confidence interval: ({ci_lower:.4f}, {ci_upper:.4f})")
```

```
[ ]: proportions_ztest_correct(count=550, nobs=n, value=0.5)
```

```
One-sample z-test with continuity correction
Alternative hypothesis: true p is not equal to 0.5
Sample proportion: 0.5403
Test statistic: 2.5387
P-value: 0.0111
95% confidence interval: (0.5092, 0.5704)
```

```
[ ]: proportions_ztest_correct(count=550, nobs=n, value=0.5, conf_level=0.99)
```

```
One-sample z-test with continuity correction
Alternative hypothesis: true p is not equal to 0.5
Sample proportion: 0.5403
Test statistic: 2.5387
P-value: 0.0111
99% confidence interval: (0.4995, 0.5800)
```

```
[ ]: # two-sided one-sample t-test
star_ttest = stats.ttest_1samp(STAR.g4reading, popmean=710, nan_policy='omit')

ci_lower = star_ttest.confidence_interval()[0]
ci_upper = star_ttest.confidence_interval()[1]

print(
    'One-sample t-test\n'
    'Alternative hypothesis: true mean is not equal to 710\n'
    f"Sample mean: {STAR.g4reading.mean():.3f}\n"
    f"t-statistic: {star_ttest.statistic:.3f}\n"
    f"df: {star_ttest.df}\n"
    f"P-value: {star_ttest.pvalue:.5f}\n"
    f"95% confidence interval: ({ci_lower:.3f}, {ci_upper:.3f})"
)
```

```
One-sample t-test
Alternative hypothesis: true mean is not equal to 710
Sample mean: 721.248
t-statistic: 10.407
df: 2352
P-value: 0.00000
95% confidence interval: (719.128, 723.367)
```

Section 7.2.4: Two-Sample Tests

```
[ ]: # one-sided p-value
stats.norm.cdf(-np.abs(ate_est), loc=0, scale=ate_se)
```

```
[ ]: 0.09350361332918433
```

```
[ ]: # two-sided p-value
     2 * stats.norm.cdf(-np.abs(ate_est), loc=0, scale=ate_se)
```

```
[ ]: 0.18700722665836866
```

```
[ ]: # testing the null of zero average treatment effect
     ttest = stats.ttest_ind(STAR.g4reading[STAR.classtype==1],
                             STAR.g4reading[STAR.classtype==2],
                             equal_var=False, nan_policy='omit')

     ci_lower = ttest.confidence_interval()[0]
     ci_upper = ttest.confidence_interval()[1]

     print(
         'Welch Two Sample t-test\n'
         'Alternative hypothesis: true difference in means is not equal to 0\n'
         f"Sample mean difference: {ate_est:.3f}\n"
         f"t-statistic: {ttest.statistic:.3f}\n"
         f"df: {ttest.df:.1f}\n"
         f"P-value: {ttest.pvalue:.5f}\n"
         f"95% confidence interval: ({ci_lower:.3f}, {ci_upper:.3f})"
     )
```

```
Welch Two Sample t-test
Alternative hypothesis: true difference in means is not equal to 0
Sample mean difference: 3.501
t-statistic: 1.319
df: 1541.2
P-value: 0.18720
95% confidence interval: (-1.704, 8.706)
```

```
[ ]: resume = pd.read_csv('resume.csv')

     # organize the data in a cross-tab
     x = pd.crosstab(resume.race, resume.call)
     x
```

```
[ ]: call      0    1
     race
     black  2278  157
     white  2200  235
```

```
[ ]: # one-sided test with continuity correction factor
     result = stats.chi2_contingency(x)

     print(
         '2-sample test for equality of proportions with continuity correction\n'
         'Alternative hypothesis: greater\n'
```

```
    f"X-squared: {result.statistic:.3f}\n"
    f"P-value: {result.pvalue/2}\n"
)
```

```
2-sample test for equality of proportions with continuity correction
Alternative hypothesis: greater
X-squared: 16.449
P-value: 2.4987891949816276e-05
```

[ ]:
```
# sample size
n0 = (resume.race=='black').sum()
n1 = (resume.race=='white').sum()

# sample proportions
p = resume['call'].mean() # overall
p0 = resume['call'][resume.race=='black'].mean()
p1 = resume['call'][resume.race=='white'].mean()

# point estimate
est = p1 - p0
est
```

[ ]: 0.032032854209445585

[ ]:
```
# standard error
se = np.sqrt(p * (1 - p) * (1 / n0 + 1 / n1))
se
```

[ ]: 0.007796894036170457

[ ]:
```
# z-statistic
zstat = est / se
zstat
```

[ ]: 4.108412152434346

[ ]:
```
# one-sided p-value
stats.norm.cdf(-abs(zstat))
```

[ ]: 1.9919434187925383e-05

[ ]:
```
result_uncorrected = stats.chi2_contingency(x, correction=False)

print(
    '2-sample test for equality of proportions without continuity correction\n'
    'Alternative hypothesis: greater\n'
    f"X-squared: {result_uncorrected.statistic:.3f}\n"
```

```
    f"P-value: {result_uncorrected.pvalue/2}"
)
```

```
2-sample test for equality of proportions without continuity correction
Alternative hypothesis: greater
X-squared: 16.879
P-value: 1.991943418792538e-05
```

Section 7.2.5: Pitfalls of Hypothesis Testing

Section 7.2.6: Power Analysis

[ ]:
```
# set the parameters
n = 250
p_star = 0.48 # data generating process
p = 0.5 # null value
alpha = 0.05

# critical value
cr_value = stats.norm.ppf(1-alpha/2)

# standard errors under the hypothetical data generating process
se_star = np.sqrt(p_star * (1 - p_star) / n)

# standard error under the null
se = np.sqrt(p * (1 - p) / n)

# power
(stats.norm.cdf(p - cr_value * se, loc=p_star, scale=se_star) +
 1 - stats.norm.cdf(p + cr_value * se, loc=p_star, scale=se_star))
```

[ ]: 0.09673113765989816

[ ]:
```
# parameters
n1 = 500
n0 = 500
p1_star = 0.05
p0_star = 0.1

# overall call back rate as a weighted average
p = (n1 * p1_star + n0 * p0_star) / (n1 + n0)
# standard error under the null
se = np.sqrt(p * (1 - p) * (1 / n1 + 1 / n0))
# standard error under the hypothetical data generating process
se_star = np.sqrt(p1_star * (1 - p1_star) / n1 + p0_star * (1 - p0_star) / n0)

(stats.norm.cdf(-cr_value * se, loc=(p1_star-p0_star), scale=se_star) +
 1 - stats.norm.cdf(cr_value * se, loc=(p1_star-p0_star), scale=se_star))
```

[ ]: 0.8522799668094607

```python
[ ]: from statsmodels.stats.proportion import (power_proportions_2indep,
                                           samplesize_proportions_2indep_onetail)

     power = power_proportions_2indep(
         diff=(p0_star - p1_star), prop2=p1_star, alpha=0.05, nobs1=n1
     )
     print(power)
```

```
power = 0.8522799668094605
p_pooled = 0.07500000000000001
std_null = 0.37249161064378356
std_alt = 0.37080992435478316
nobs1 = 500
nobs2 = 500
nobs_ratio = 1
alpha = 0.05
```

```python
[ ]: samplesize_proportions_2indep_onetail(
         diff=(p0_star - p1_star), prop2=p1_star, alpha=0.05, power=0.9
     )
```

[ ]: 581.082053834476

```python
[ ]: from statsmodels.stats.power import TTestPower, TTestIndPower

     TTestPower().solve_power(effect_size=0.25, nobs=100, alpha=0.05)
```

[ ]: 0.696980269099517

```python
[ ]: TTestPower().solve_power(effect_size=0.25, power=0.9, alpha=0.05)
```

[ ]: 170.05107691102737

```python
[ ]: TTestIndPower().solve_power(effect_size=0.25, power=0.9, alpha=0.05,
                                 alternative='larger')
```

[ ]: 274.72216286128617

Section 7.3: Linear Regression Model with Uncertainty

Section 7.3.1: Linear Regression as a Generative Model

```python
[ ]: import statsmodels.formula.api as smf

     minwage = pd.read_csv('minwage.csv')

     # compute proportion of full employment before minimum wage increase
```

```python
minwage['fullPropBefore'] = minwage['fullBefore'] / (
    minwage['fullBefore'] + minwage['partBefore']
)

# same thing after minimum wage increase
minwage['fullPropAfter'] = minwage['fullAfter'] / (
    minwage['fullAfter'] + minwage['partAfter']
)

# an indicator for NJ: 1 if it's located in NJ and 0 if in PA
minwage['NJ'] = np.where(minwage['location']=='PA', 0, 1)

minwage_model = 'fullPropAfter ~ -1 + NJ + fullPropBefore + wageBefore + chain'

fit_minwage = smf.ols(minwage_model, data=minwage).fit()

# regression result
fit_minwage.params
```

[ ]:
```
chain[burgerking]   -0.115625
chain[kfc]          -0.150800
chain[roys]         -0.206386
chain[wendys]       -0.220133
NJ                   0.054220
fullPropBefore       0.168794
wageBefore           0.081334
dtype: float64
```

[ ]:
```python
minwage_model1 = 'fullPropAfter ~ NJ + fullPropBefore + wageBefore + chain'

fit_minwage1 = smf.ols(minwage_model1, data=minwage).fit()

fit_minwage1.params
```

[ ]:
```
Intercept         -0.115625
chain[T.kfc]      -0.035175
chain[T.roys]     -0.090761
chain[T.wendys]   -0.104507
NJ                 0.054220
fullPropBefore     0.168794
wageBefore         0.081334
dtype: float64
```

[ ]:
```python
fit_minwage.predict(minwage.iloc[[0]])
```

[ ]:
```
0    0.270937
dtype: float64
```

[ ]:
```
fit_minwage1.predict(minwage.iloc[[0]])
```

[ ]:
```
0    0.270937
dtype: float64
```

Section 7.3.2: Unbiasedness of Estimated Coefficients

Section 7.3.3: Standard Errors of Estimated Coefficients

Section 7.3.4: Inference About Coefficients

[ ]:
```
women = pd.read_csv('women.csv')

fit_women = smf.ols('water ~ reserved', data=women).fit()

print(fit_women.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  water   R-squared:                       0.017
Model:                            OLS   Adj. R-squared:                  0.014
Method:                 Least Squares   F-statistic:                     5.493
Date:                Wed, 15 Nov 2023   Prob (F-statistic):             0.0197
Time:                        23:49:05   Log-Likelihood:                -1586.1
No. Observations:                 322   AIC:                             3176.
Df Residuals:                     320   BIC:                             3184.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     14.7383      2.286      6.446      0.000      10.240      19.236
reserved       9.2524      3.948      2.344      0.020       1.486      17.019
==============================================================================
Omnibus:                      398.104   Durbin-Watson:                   1.990
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            26829.354
Skew:                           5.690   Prob(JB):                         0.00
Kurtosis:                      46.246   Cond. No.                         2.41
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[ ]:
```
# 95% confidence intervals
fit_women.conf_int().rename(columns={0:'2.5%', 1:'97.5%'})
```

[ ]:
```
                2.5%      97.5%
Intercept  10.240240  19.236395
```

```
reserved    1.485608  17.019238
```

[ ]:
```
print(fit_minwage.summary(slim=True))
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:           fullPropAfter   R-squared:                       0.070
Model:                             OLS   Adj. R-squared:                  0.054
No. Observations:                  358   F-statistic:                     4.401
Covariance Type:             nonrobust   Prob (F-statistic):           0.000264
================================================================================
=====
                        coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
-----
chain[burgerking]    -0.1156      0.179     -0.646      0.518      -0.467
0.236
chain[kfc]           -0.1508      0.183     -0.824      0.411      -0.511
0.209
chain[roys]          -0.2064      0.187     -1.105      0.270      -0.574
0.161
chain[wendys]        -0.2201      0.188     -1.168      0.243      -0.591
0.150
NJ                    0.0542      0.033      1.633      0.103      -0.011
0.120
fullPropBefore        0.1688      0.057      2.981      0.003       0.057
0.280
wageBefore            0.0813      0.039      2.090      0.037       0.005
0.158
================================================================================
=====

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[ ]:
```
# confidence interval just for the 'NJ' variable
fit_minwage.conf_int().rename(columns={0:'2.5%', 1:'97.5%'}).loc['NJ']
```

[ ]:
```
2.5%    -0.011093
97.5%    0.119533
Name: NJ, dtype: float64
```

Section 7.3.5: Inference About Predictions

In Progress