exercise4_109301060

June 4, 2024

1 Exercises for Lecture 4

1.0.1 109301060

1.0.2 Q1

a.

[ ]:
```
import statsmodels.formula.api as smf
import scipy.stats as stats
import pandas as pd
import numpy as np

## a.
## Set sample size and number of iterations:
n = 100                        ## sample size
r = 50                       ## number of iteration

## Set true parameters
beta0 = 0.4
beta1 = -1.46
beta2 = 2.5

df = n - 2 - 1
cv_005 = stats.t.ppf(1 - 0.05, df = df)                 ## Critical value for
 ↪95% CI
cv_001 = stats.t.ppf(1 - 0.01, df = df)             ## Critical value for 99% CI

## Create empty vectors to store the simulation results
beta0_result_005 = np.empty(r)
beta1_result_005 = np.empty(r)
beta2_result_005 = np.empty(r)

beta0_result_001 = np.empty(r)
beta1_result_001 = np.empty(r)
beta2_result_001 = np.empty(r)

## Set random seed
np.random.seed(1234567)
```

```python
## Generate samples of x1 and x2
x1 = stats.uniform.rvs(size = n)
x2 = stats.norm.rvs(loc = 0.78, size = n)

for i in range(r):

    u = np.random.normal(size = n)                ## Generate u
    y = beta0 + beta1 * x1 + beta2 * x2 + u    ## Generate y
    datax = pd.DataFrame({'y':y, 'x1':x1, 'x2':x2})

    reg = smf.ols(formula = 'y ~ x1 + x2', data = datax)
    results = reg.fit()
    beta_hat = results.params
    bse = results.bse

    ## Whether the true parameters lie in the 95% CI's?
    beta0_result_005[i] = (beta0 >= (beta_hat['Intercept'] - cv_005*bse['Intercept'])) and (beta0 <= (beta_hat['Intercept'] + cv_005*bse['Intercept']))
    beta1_result_005[i] = (beta1 >= (beta_hat['x1'] - cv_005*bse['x1'])) and (beta1 <= (beta_hat['x1'] + cv_005*bse['x1']))
    beta2_result_005[i] = (beta2 >= (beta_hat['x2'] - cv_005*bse['x2'])) and (beta2 <= (beta_hat['x2'] + cv_005*bse['x2']))

    ## Whether the true parameters lie in the 99% CI's?
    beta0_result_001[i] = (beta0 >= (beta_hat['Intercept'] - cv_001*bse['Intercept'])) and (beta0 <= (beta_hat['Intercept'] + cv_001*bse['Intercept']))
    beta1_result_001[i] = (beta1 >= (beta_hat['x1'] - cv_001*bse['x1'])) and (beta1 <= (beta_hat['x1'] + cv_001*bse['x1']))
    beta2_result_001[i] = (beta2 >= (beta_hat['x2'] - cv_001*bse['x2'])) and (beta2 <= (beta_hat['x2'] + cv_001*bse['x2']))


## Print out the results
## Alpha = 0.05
print(np.mean(beta0_result_005), np.mean(beta1_result_005), np.mean(beta2_result_005))
## Alpha = 0.01
print(np.mean(beta0_result_001), np.mean(beta1_result_001), np.mean(beta2_result_001))
```

```
0.94 0.9 0.92
1.0 0.98 0.96
```

b.

[ ]:

```python
import statsmodels.formula.api as smf
import scipy.stats as stats
import pandas as pd
import numpy as np

## a.
## Set sample size and number of iterations:
n = 100                         ## sample size
r = 200                        ## number of iteration

## Set true parameters
beta0 = 0.4
beta1 = -1.46
beta2 = 2.5

df = n - 2 - 1
cv_005 = stats.t.ppf(1 - 0.05, df = df)                ## Critical value for␣
 ↪95% CI
cv_001 = stats.t.ppf(1 - 0.01, df = df)            ## Critical value for 99% CI

## Create empty vectors to store the simulation results
beta0_result_005 = np.empty(r)
beta1_result_005 = np.empty(r)
beta2_result_005 = np.empty(r)

beta0_result_001 = np.empty(r)
beta1_result_001 = np.empty(r)
beta2_result_001 = np.empty(r)

## Set random seed
np.random.seed(1234567)

## Generate samples of x1 and x2
x1 = stats.uniform.rvs(size = n)
x2 = stats.norm.rvs(loc = 0.78, size = n)

for i in range(r):

    u = np.random.normal(size = n)               ## Generate u
    y = beta0 + beta1 * x1 + beta2 * x2 + u    ## Generate y
    datax = pd.DataFrame({'y':y, 'x1':x1, 'x2':x2})

    reg = smf.ols(formula = 'y ~ x1 + x2', data = datax)
    results = reg.fit()
    beta_hat = results.params
    bse = results.bse
```

```python
    ## Whether the true parameters lie in the 95% CI's?
    beta0_result_005[i] = (beta0 >= (beta_hat['Intercept'] - cv_005*bse['Intercept'])) and (beta0 <= (beta_hat['Intercept'] + cv_005*bse['Intercept']))
    beta1_result_005[i] = (beta1 >= (beta_hat['x1'] - cv_005*bse['x1'])) and (beta1 <= (beta_hat['x1'] + cv_005*bse['x1']))
    beta2_result_005[i] = (beta2 >= (beta_hat['x2'] - cv_005*bse['x2'])) and (beta2 <= (beta_hat['x2'] + cv_005*bse['x2']))

    ## Whether the true parameters lie in the 99% CI's?
    beta0_result_001[i] = (beta0 >= (beta_hat['Intercept'] - cv_001*bse['Intercept'])) and (beta0 <= (beta_hat['Intercept'] + cv_001*bse['Intercept']))
    beta1_result_001[i] = (beta1 >= (beta_hat['x1'] - cv_001*bse['x1'])) and (beta1 <= (beta_hat['x1'] + cv_001*bse['x1']))
    beta2_result_001[i] = (beta2 >= (beta_hat['x2'] - cv_001*bse['x2'])) and (beta2 <= (beta_hat['x2'] + cv_001*bse['x2']))


## Print out the results
## Alpha = 0.05
print(np.mean(beta0_result_005), np.mean(beta1_result_005), np.mean(beta2_result_005))
## Alpha = 0.01
print(np.mean(beta0_result_001), np.mean(beta1_result_001), np.mean(beta2_result_001))
```

```
0.895 0.89 0.915
1.0 0.975 0.97
```

c.

[ ]:

```python
import statsmodels.formula.api as smf
import scipy.stats as stats
import pandas as pd
import numpy as np

## a.
## Set sample size and number of iterations:
n = 100                          ## sample size
r = 1000                         ## number of iteration

## Set true parameters
beta0 = 0.4
beta1 = -1.46
beta2 = 2.5
```

```
df = n - 2 - 1
cv_005 = stats.t.ppf(1 - 0.05, df = df)                ## Critical value for␣
 ↪95% CI
cv_001 = stats.t.ppf(1 - 0.01, df = df)             ## Critical value for 99% CI

## Create empty vectors to store the simulation results
beta0_result_005 = np.empty(r)
beta1_result_005 = np.empty(r)
beta2_result_005 = np.empty(r)

beta0_result_001 = np.empty(r)
beta1_result_001 = np.empty(r)
beta2_result_001 = np.empty(r)

## Set random seed
np.random.seed(1234567)

## Generate samples of x1 and x2
x1 = stats.uniform.rvs(size = n)
x2 = stats.norm.rvs(loc = 0.78, size = n)

for i in range(r):

    u = np.random.normal(size = n)               ## Generate u
    y = beta0 + beta1 * x1 + beta2 * x2 + u     ## Generate y
    datax = pd.DataFrame({'y':y, 'x1':x1, 'x2':x2})

    reg = smf.ols(formula = 'y ~ x1 + x2', data = datax)
    results = reg.fit()
    beta_hat = results.params
    bse = results.bse

    ## Whether the true parameters lie in the 95% CI's?
    beta0_result_005[i] = (beta0 >= (beta_hat['Intercept'] -␣
 ↪cv_005*bse['Intercept'])) and (beta0 <= (beta_hat['Intercept'] +␣
 ↪cv_005*bse['Intercept']))
    beta1_result_005[i] = (beta1 >= (beta_hat['x1'] - cv_005*bse['x1'])) and␣
 ↪(beta1 <= (beta_hat['x1'] + cv_005*bse['x1']))
    beta2_result_005[i] = (beta2 >= (beta_hat['x2'] - cv_005*bse['x2'])) and␣
 ↪(beta2 <= (beta_hat['x2'] + cv_005*bse['x2']))

    ## Whether the true parameters lie in the 99% CI's?
    beta0_result_001[i] = (beta0 >= (beta_hat['Intercept'] -␣
 ↪cv_001*bse['Intercept'])) and (beta0 <= (beta_hat['Intercept'] +␣
 ↪cv_001*bse['Intercept']))
    beta1_result_001[i] = (beta1 >= (beta_hat['x1'] - cv_001*bse['x1'])) and␣
 ↪(beta1 <= (beta_hat['x1'] + cv_001*bse['x1']))
```

```
    beta2_result_001[i] = (beta2 >= (beta_hat['x2'] - cv_001*bse['x2'])) and␣
↪(beta2 <= (beta_hat['x2'] + cv_001*bse['x2']))


## Print out the results
## Alpha = 0.05
print(np.mean(beta0_result_005), np.mean(beta1_result_005), np.
↪mean(beta2_result_005))
## Alpha = 0.01
print(np.mean(beta0_result_001), np.mean(beta1_result_001), np.
↪mean(beta2_result_001))
```

```
0.883 0.879 0.899
0.98 0.979 0.978
```

1.0.3 Q2

a.

[ ]:

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats
import numpy as np

## Import data
vote1 = woo.data('vote1')

## a. Descriptive statistics
vote1[['voteA','lexpendA','lexpendB','prtystrA']].describe()
```

[ ]:

	voteA	lexpendA	lexpendB	prtystrA
count	173.000000	173.000000	173.000000	173.000000
mean	50.502890	5.025556	4.944369	49.757225
std	16.784761	1.601602	1.571143	9.983650
min	16.000000	-1.197328	-0.072571	22.000000
25%	36.000000	4.402246	4.095244	44.000000
50%	50.000000	5.492164	5.400558	50.000000
75%	65.000000	6.125580	6.110837	56.000000
max	84.000000	7.293476	7.344844	71.000000

b.

β_1 is the percentage point increase in *voteA* for a one unit increase in *lexpendA*, holding other variables constant.

c.

$H_0 : \beta_1 + \beta_2 = 0$

d.

```
[ ]: ## d. Fit the regression
reg = smf.ols(formula = 'voteA ~ lexpendA + lexpendB + prtystrA', data = vote1)
results = reg.fit()

print(f'results.summary():\n{results.summary()}\n')
```

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                  voteA   R-squared:                       0.793
Model:                            OLS   Adj. R-squared:                  0.789
Method:                 Least Squares   F-statistic:                     215.2
Date:                Tue, 04 Jun 2024   Prob (F-statistic):           1.76e-57
Time:                        09:55:07   Log-Likelihood:                -596.86
No. Observations:                 173   AIC:                             1202.
Df Residuals:                     169   BIC:                             1214.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     45.0789      3.926     11.481      0.000      37.328      52.830
lexpendA       6.0833      0.382     15.919      0.000       5.329       6.838
lexpendB      -6.6154      0.379    -17.463      0.000      -7.363      -5.868
prtystrA       0.1520      0.062      2.450      0.015       0.030       0.274
==============================================================================
Omnibus:                        8.900   Durbin-Watson:                   1.604
Prob(Omnibus):                  0.012   Jarque-Bera (JB):                8.832
Skew:                           0.493   Prob(JB):                       0.0121
Kurtosis:                       3.505   Cond. No.                         344.
==============================================================================

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

From the regression output, we cannot reject the null hypothesis that $\beta_1 + \beta_2 = 0$ at 5% significance level.

```
[ ]: ## Test H0: beta1 + beta 2 = 0 in b.
## Use method f_test
hypotheses = ['lexpendA + lexpendB = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.fvalue                        ## Extract the F statistic
fpval = ftest.pvalue                            ## Extract the p-value

print(f'fstat: {round(fstat, 3)}\n')
```

```
print(f'fpval: {round(fpval, 3)}\n')
```

```
fstat: 0.996

fpval: 0.32
```

The p-value is larger than 0.05, so we cannot reject the null hypothesis at 5% significance level.

e.

```
[ ]: ## e. Estimate a modified model:
vote1['lB_lA'] = vote1['lexpendB'] - vote1['lexpendA']
reg_mod = smf.ols(formula = 'voteA ~ lB_lA + prtystrA', data = vote1)
results_mod = reg_mod.fit()
print(f'results_mod.summary():\n{results_mod.summary()}\n')

## From the result, p-value of the t statistic (coefficient of lexpendA) is␣
↪also about 0.32 and we cannot reject the null
## at 5% significant level. We do not have enough evidence to say that there is␣
↪no offset effect.

## Verify whether t^2 = F
tvalues = results_mod.tvalues
tvalues_sq = tvalues['lB_lA']**2
print(round(tvalues_sq,4), round(fstat,4))
```

```
results_mod.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                  voteA   R-squared:                       0.791
Model:                            OLS   Adj. R-squared:                  0.789
Method:                 Least Squares   F-statistic:                     322.3
Date:                Tue, 04 Jun 2024   Prob (F-statistic):           1.42e-58
Time:                        09:55:07   Log-Likelihood:                -597.37
No. Observations:                 173   AIC:                             1201.
Df Residuals:                     170   BIC:                             1210.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     42.7028      3.122     13.677      0.000      36.539      48.866
lB_lA         -6.3517      0.272    -23.394      0.000      -6.888      -5.816
prtystrA       0.1464      0.062      2.370      0.019       0.024       0.268
==============================================================================
Omnibus:                       11.793   Durbin-Watson:                   1.599
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               12.607
Skew:                           0.554   Prob(JB):                      0.00183
```

```
Kurtosis:                       3.721   Cond. No.                       270.
==============================================================================

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

547.2726 0.9963
```

f.

```
[ ]: ## f. Test H0: beta1 + beta2 = 1.5
## Use the method f_test
hypotheses = ['lexpendA + lexpendB = 1.5']
ftest = results.f_test(hypotheses)
fstat = ftest.fvalue
fpval = ftest.pvalue

print(f'fstat: {round(fstat, 3)}\n')
print(f'fpval: {round(fpval, 3)}\n')

## Estimate a modified model
vote1['lA_lB'] = vote1['lexpendA'] - vote1['lexpendB']
reg_mod = smf.ols(formula = 'voteA ~ lA_lB + prtystrA', data = vote1)
results_mod = reg_mod.fit()
print(f'results_mod.summary():\n{results_mod.summary()}\n')

## Manually calculate t statistic
n = vote1.shape[0]
k = 3
df = n - k - 1
b_lexpendA = results_mod.params['lA_lB']
se_lexpendA = results_mod.bse['lA_lB']
tstat = round((b_lexpendA - 1.5) / se_lexpendA, 3)
tpval = 2*stats.t.cdf(-abs(tstat), df = df)

print(f'tstat: {round(tstat, 3)}\n')
print(f'tpval: {round(tpval, 3)}\n')                    ## should be the same␣
 ↪value as fpval

## Use method t_test
hypothesis = 'lA_lB = 1.5'
ttest = results_mod.t_test(hypothesis)
tstat = ttest.statistic[0][0]
tpval = ttest.pvalue

print(f'tstat: {round(tstat, 3)}\n')
```

```
print(f'tpval: {np.around(tpval, 3)}\n')                    ## should be the same␣
 ↪value as fpval
```

```
fstat: 14.531

fpval: 0.0

results_mod.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                  voteA   R-squared:                       0.791
Model:                            OLS   Adj. R-squared:                  0.789
Method:                 Least Squares   F-statistic:                     322.3
Date:                Tue, 04 Jun 2024   Prob (F-statistic):           1.42e-58
Time:                        09:55:07   Log-Likelihood:                -597.37
No. Observations:                 173   AIC:                             1201.
Df Residuals:                     170   BIC:                             1210.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     42.7028      3.122     13.677      0.000      36.539      48.866
lA_lB          6.3517      0.272     23.394      0.000       5.816       6.888
prtystrA       0.1464      0.062      2.370      0.019       0.024       0.268
==============================================================================
Omnibus:                       11.793   Durbin-Watson:                   1.599
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               12.607
Skew:                           0.554   Prob(JB):                      0.00183
Kurtosis:                       3.721   Cond. No.                         270.
==============================================================================

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

tstat: 17.869

tpval: 0.0

tstat: 17.869

tpval: 0.0
```

1.0.4 Q3

a.

```python
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf
import scipy.stats as stats

## Import data
hprice1 = woo.data('hprice1')

## a. Descriptive statistics
hprice1[['price','sqrft','bdrms']].describe()
```

	price	sqrft	bdrms
count	88.000000	88.000000	88.000000
mean	293.546034	2013.693182	3.568182
std	102.713445	577.191583	0.841393
min	111.000000	1171.000000	2.000000
25%	230.000000	1660.500000	3.000000
50%	265.500000	1845.000000	3.000000
75%	326.250000	2227.000000	4.000000
max	725.000000	3880.000000	7.000000

b.

```python
## b. Fit the regression
reg = smf.ols(formula = 'np.log(price) ~ sqrft + bdrms', data = hprice1)
results = reg.fit()

## Estimate theta1: 150 sqrft bed room added
theta1 = 150*results.params['sqrft'] + results.params['bdrms']
theta1 = round(theta1, 4)
print(f'Percentage change:\n{theta1*100}%\n')
```

```
Percentage change:
8.58%
```

c.

```python
## c.
## Calculate standard error of theta1 and construct CI's
hprice1['new_var'] = hprice1['sqrft'] - 150*hprice1['bdrms']
reg_mod = smf.ols(formula = 'np.log(price) ~ new_var + bdrms', data = hprice1)
results_mod = reg_mod.fit()
print(round(results_mod.bse['bdrms'],4))
```

```
0.0268
```

standard error of the percentage change in b is 0.0268

```
[ ]: ## Now the slope parameters of bdrms is theta1
bse = results_mod.bse['bdrms']
n = hprice1.shape[0]
k = 2
df = n - k - 1
cv_005 = stats.t.ppf(1 - 0.025, df = df)          ## Critical value for 95% CI
cv_001 = stats.t.ppf(1 - 0.005, df = df)          ## Critical value for 99% CI

ci_005_up = theta1 + cv_005*bse
ci_005_low = theta1 - cv_005*bse
ci_005 = [round(ci_005_low, 3), round(ci_005_up, 3)]
print(f'ci_005:\n{ci_005}\n')                     ## 95% CI for theta1

ci_001_up = theta1 + cv_001*bse
ci_001_low = theta1 - cv_001*bse
ci_001 = [round(ci_001_low, 3), round(ci_001_up, 3)]
print(f'ci_001:\n{ci_001}\n')                     ## 99% CI for theta1
```

```
ci_005:
[0.033, 0.139]

ci_001:
[0.015, 0.156]
```

Based on the confidence intervals do not contain zero. This provides statistical evidence that adding a 150-square-foot bedroom significantly increases the house price, with the percentage increase ranging from approximately 3.3% to 15.6% (95% confidence interval).