Solving the Bootstrap Question from Problem Set 1

Econometrics 1 @ FGV EPGE - 2025

Instructor: Raul Guarini Riva

TA: Taric Latif Padovani

This notebook solves Problem 3 (bootstrap practice) from PS1. For each item, we spell out the analytical answer, document how the simulation is implemented, and summarize the numerical results.

```
# Importing packages
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import matplotlib as mpl
from IPython import get_ipython

# Stuff to get nice pictures
mpl.rcParams['font.family'] = 'sans-serif'
mpl.rcParams['font.sans-serif'] = ['Arial']
ip = get_ipython()
if ip is not None:
    ip.run_line_magic('config', 'InlineBackend.figure_format = \'retina\'')
```

```
# True Parameters
alpha = 2
beta = 5
np.random.seed(123)  # For reproducibility (set any seed you want)
```

Data-Generating Process

We assume the structural model

$$y_i = \alpha + \beta x_i + \varepsilon_i,$$

with (x_i, ε_i) i.i.d., $x_i \sim \mathcal{N}(0, 3)$, and conditional heteroskedasticity

$$\varepsilon_i \mid x_i \sim \mathcal{N}\left(0,\, 2 + x_i^2\right).$$

The population parameters are fixed at $\alpha = 2$ and $\beta = 5$. The heteroskedastic variance motivates the bootstrap exercise: we want to understand how different standard-error estimators behave when the usual homoskedastic assumption fails.

Helper Functions

The functions below generate synthetic samples, run OLS, and compute three flavours of standard errors.

- `simulate_sample` draws (x_i, y_i) pairs from the DGP.
- `run_regression` returns the OLS estimates and residuals.
- `compute_naive_se` implements the textbook homoskedastic formula.
- `compute_robust_se` delivers the Eicker–Huber–White (HC0) estimator.
- `bootstrap_reg` resamples observations with replacement and re-estimates β.
- `compute_confidence_sets` packages the three 95% confidence intervals.

```
# Define a function that takes residuals and X and provide standard error for beta_hat
def simulate_sample(n=100):
    X_sample = np.random.normal(0, 3, n)
    sigma_epsilon_sample = np.sqrt(2 + X_sample**2)
```

```
    epsilon_sample = np.random.normal(0, sigma_epsilon_sample)
    y_sample = alpha + beta * X_sample + epsilon_sample
    return y_sample, X_sample

def run_regression(y_sample, x_sample):
    X = np.column_stack((np.ones(len(y_sample)), x_sample))
    beta_hat = np.linalg.inv(X.T @ X) @ (X.T @ y_sample)
    residuals = y_sample - X @ beta_hat
    return beta_hat, residuals

def compute_naive_se(residuals, x_sample):
    n = len(residuals)
    X = np.column_stack((np.ones(n), x_sample))
    sigma_hat = np.sqrt(np.sum(residuals ** 2) / (n - 2))
    se = sigma_hat * np.sqrt(np.diag(np.linalg.inv(X.T @ X)))
    se = se[1]
    return se

def compute_robust_se(residuals, x_sample):
    n = len(residuals)
    X = np.column_stack((np.ones(n), x_sample))
    bread = X.T @ X
    bread_inv = np.linalg.inv(bread)
    omega = np.diag(residuals ** 2)
    meat = X.T @ omega @ X
    se = np.sqrt(np.diag(bread_inv @ meat @ bread_inv))
    se = se[1]
    return se

def bootstrap_reg(y_sample, x_sample, n_boot=1000):
    n = len(y_sample)
    beta_boot = np.zeros((n_boot, 2))
    for i in range(n_boot):
        indices = np.random.choice(n, n, replace=True)
        y_resampled = y_sample[indices]
        x_resampled = x_sample[indices]
        beta_boot[i], _ = run_regression(y_resampled, x_resampled)
    return beta_boot, np.std(beta_boot, axis=0)

def compute_confidence_sets(y, x_sample):
    beta_hat, residuals = run_regression(y, x_sample)
    se_naive = compute_naive_se(residuals, x_sample)
    se_robust = compute_robust_se(residuals, x_sample)

    # Bootstrap confidence intervals
    beta_boot, se_boot = bootstrap_reg(y, x_sample)

    critical_value = 1.96  # For 95% confidence interval
    ci_naive = (beta_hat[1] - critical_value * se_naive,
                beta_hat[1] + critical_value * se_naive)
    ci_robust = (beta_hat[1] - critical_value * se_robust,
                 beta_hat[1] + critical_value * se_robust)
    ci_boot = (np.percentile(beta_boot[:, 1], 2.5),
               np.percentile(beta_boot[:, 1], 97.5))

    ci_array = np.array([ci_naive, ci_robust, ci_boot])
```

```
    return ci_array

def report_nice_sets(ci_array):
    print(f"Naive CI: {np.round(ci_array[0], 3)}")
    print(f"Robust CI: {np.round(ci_array[1], 3)}")
    print(f"Bootstrap CI: {np.round(ci_array[2], 3)}")
```

Item A — Conditional Distribution

Conditioning on x_i simply plugs the realized regressor into the mean and variance of the normal shock. Because (α, β) are fixed constants and $\varepsilon_i \mid x_i \sim \mathcal{N}(0, 2 + x_i^2)$, we obtain

$$y_i \mid x_i \sim \mathcal{N}(\alpha + \beta x_i,\, 2 + x_i^2).$$

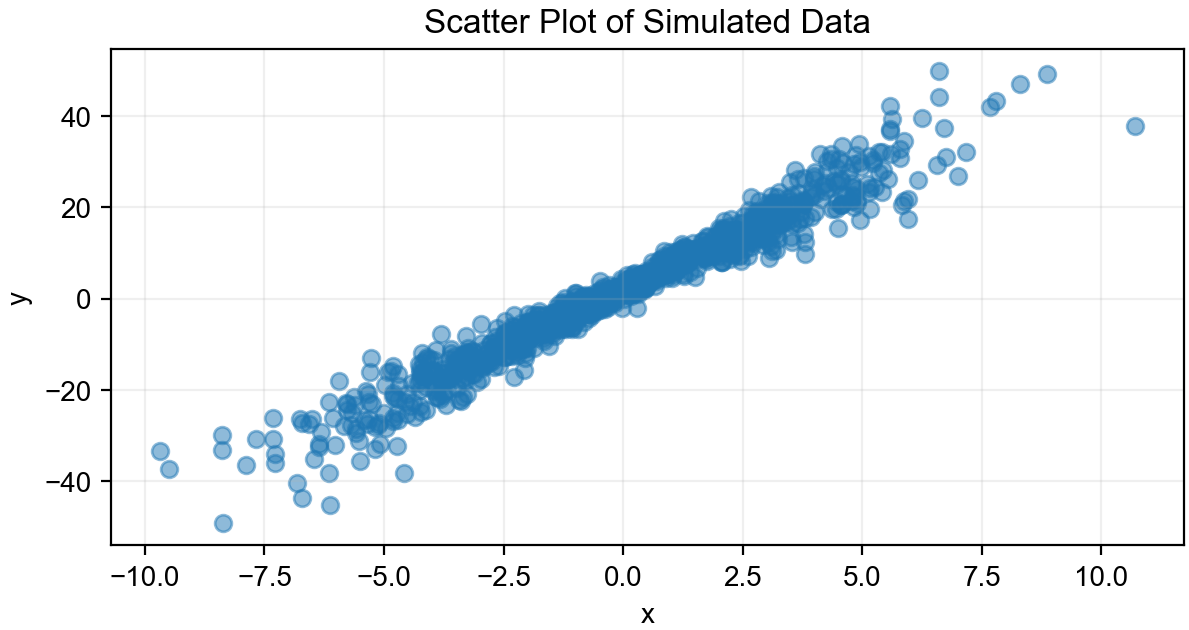
This provides the analytical answer requested in part (a). The remaining items verify through simulation that our estimators behave as expected when the variance changes with x_i.

Item B — Simulating One Sample

We draw $n = 100$ observations from the DGP and visualise the sample. The scatter plot should already hint at the non-constant variance structure.

```
y, X = simulate_sample(n=1000)

plt.figure(figsize=(6, 3))
plt.scatter(X, y, alpha=0.5)
plt.tight_layout()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Scatter Plot of Simulated Data')
plt.grid(alpha = 0.2)
plt.show()
```



The simulated cloud clearly fans out as $|x_i|$ grows, mirroring the variance function $2 + x_i^2$. This visual diagnostic foreshadows why homoskedastic standard errors will undercover β.

Item C — OLS Estimates

Running OLS on the simulated sample delivers point estimates for (α, β). We report them next to check whether the estimator recovers the true coefficients in finite samples.

```
beta_hat, residuals = run_regression(y, X)
print(f"Estimated coefficients: {np.round(beta_hat, 3)}")
```

```
Estimated coefficients: [1.959 4.935]
```

With one simulated sample of size $n = 100$, OLS lands close to the true coefficients $(\alpha, \beta) = (2, 5)$. Sampling error remains, but the unbiasedness of OLS under the classical assumptions (which still hold here despite heteroskedasticity) shows up in practice.

Item D — Comparing Confidence Intervals

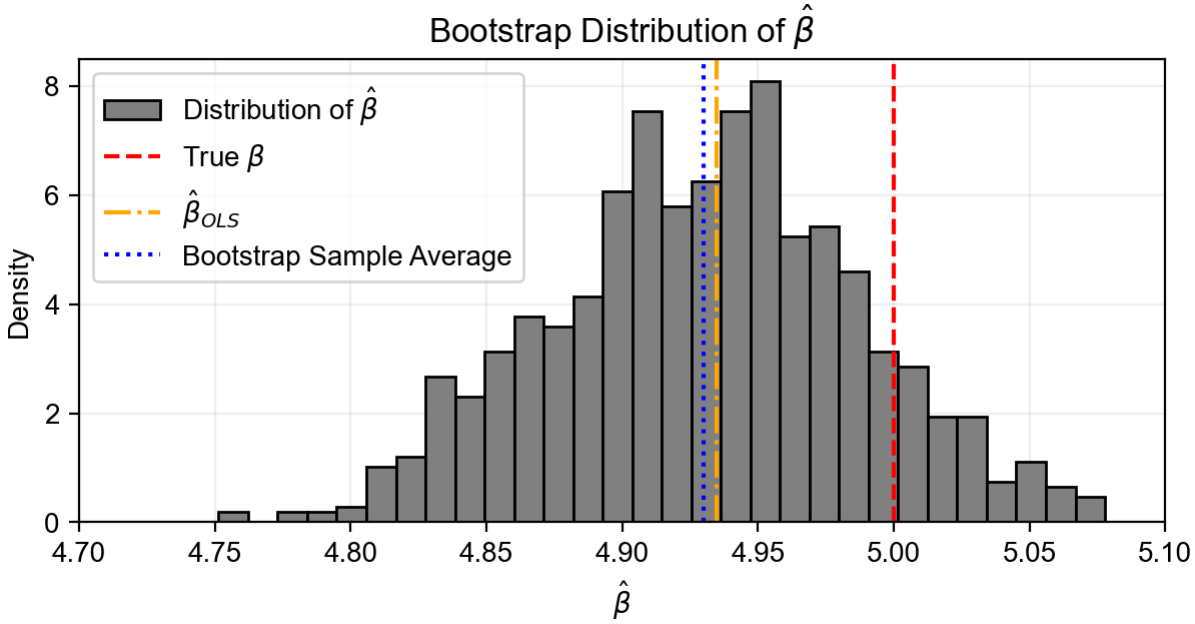
We compute three 95% symmetric intervals for β: the homoskedastic textbook version, the Eicker–Huber–White robust version, and a percentile bootstrap interval based on $B = 1000$ resamples.

```
confidence_sets = compute_confidence_sets(y, X)
report_nice_sets(confidence_sets)
```

```
Naive CI: [4.867 5.002]
Robust CI: [4.822 5.048]
Bootstrap CI: [4.821 5.044]
```

```
%%time

np.random.seed(123)
beta_hat_boot, se_boot = bootstrap_reg(y, X, n_boot=1000)
plt.figure(figsize = (7, 3))
plt.hist(beta_hat_boot[:, 1],
         density=True,
         ec = "k",
         color="grey",
         bins = 30,
         label=r'Distribution of $\hat{\beta}$')
plt.axvline(x=beta, color='red', linestyle='--', label=r'True $\beta$')
plt.axvline(x=beta_hat[1], color="orange", ls="-.", label=r"$\hat{\beta}_{OLS}$")
plt.axvline(x=np.mean(beta_hat_boot[:, 1]), color="blue", ls=":", label="Bootstrap Sample Average")
plt.xlabel(r'$\hat{\beta}$')
plt.ylabel('Density')
plt.xlim(4.7, 5.1)
plt.title(r'Bootstrap Distribution of $\hat{\beta}$')
plt.grid(alpha=0.2)
plt.gca().set_axisbelow(True)
plt.legend()
plt.show()
```




```
CPU times: user 174 ms, sys: 8.79 ms, total: 183 ms
Wall time: 189 ms
```

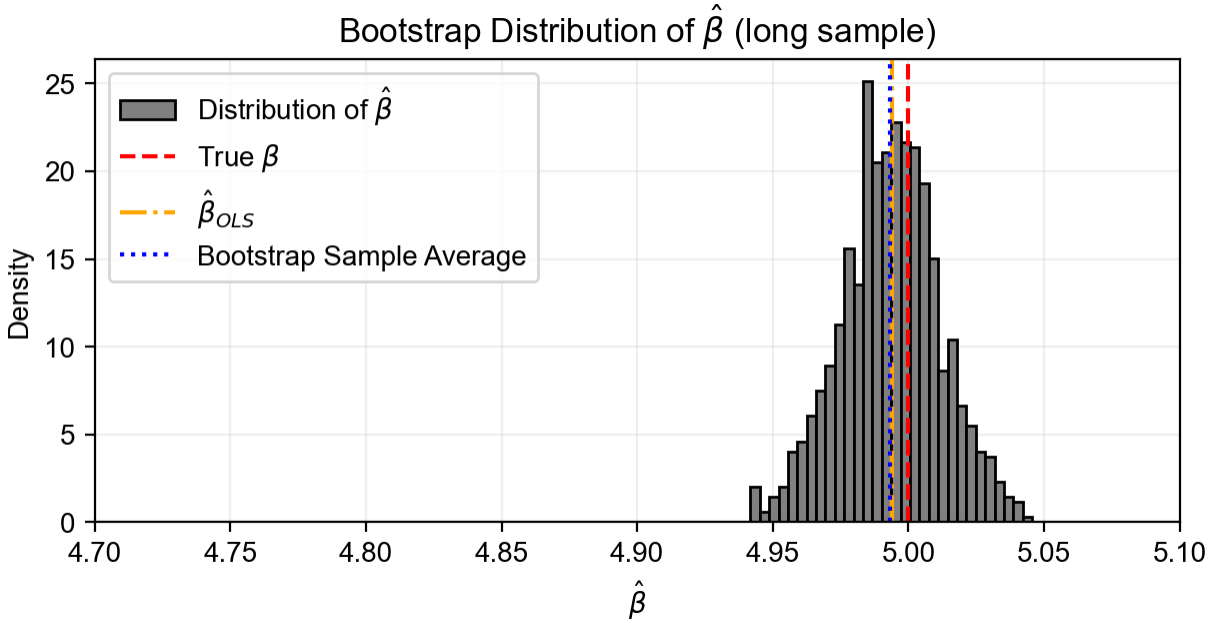
Notice how the distribution is not centered around the true β. This is the expected behavior even if $m \to \infty$. The bootstrap distribution is never centered on the true parameter because, under the empirical distribution, that is *not* the population parameter. When $n \to \infty$, then the OLS estimator will converge in probability to the true parameter and everything will overlap:

```
%%time

np.random.seed(123)

# Let's redo it with larger n (this was not necessary for the problem set)
y_long, X_long = simulate_sample(n=10000)
beta_hat_long, residuals_long = run_regression(y_long, X_long)
beta_hat_boot_long, se_boot_long = bootstrap_reg(y_long, X_long, n_boot=1000)

# Plotting
plt.figure(figsize = (7, 3))
plt.hist(beta_hat_boot_long[:, 1],
         density=True,
         ec = "k",
         color="grey",
         bins = 30,
         label=r'Distribution of $\hat{\beta}$')
plt.axvline(x=beta, color='red', linestyle='--', label=r'True $\beta$')
plt.axvline(x=beta_hat_long[1], color="orange", ls="-.", label=r"$\hat{\beta}_{OLS}$")
plt.axvline(x=np.mean(beta_hat_boot_long[:, 1]), color="blue", ls=":", label="Bootstrap Sample Average")
plt.xlabel(r'$\hat{\beta}$')
plt.ylabel('Density')
plt.xlim(4.7, 5.1)
plt.title(r'Bootstrap Distribution of $\hat{\beta}$ (long sample)')
plt.grid(alpha=0.2)
plt.gca().set_axisbelow(True)
plt.legend()
plt.show()
```




```
CPU times: user 361 ms, sys: 7.1 ms, total: 368 ms
Wall time: 368 ms
```

Item E — Monte Carlo Coverage Study

Finally, we repeat the entire procedure $m = 500$ times to approximate the finite-sample coverage probabilities of the three intervals.

```
%%time
m = 500

sets = np.zeros((m, 3, 2))
beta_included = np.zeros((m, 3))
for i in range(m):
    y, X = simulate_sample(n=100)
    sets[i] = compute_confidence_sets(y, X)
    for j in range(3):
        beta_included[i, j] = sets[i, j, 0] <= beta and beta <= sets[i, j, 1]

average_inclusion = np.mean(beta_included, axis=0)
```

```
CPU times: user 9.23 s, sys: 38.1 ms, total: 9.27 s
Wall time: 9.32 s
```

```
print("Average Coverage Probability of Beta in Confidence Sets:")
print(f"Naive: {np.round(average_inclusion[0], 3)}")
print(f"Robust: {np.round(average_inclusion[1], 3)}")
print(f"Bootstrap: {np.round(average_inclusion[2], 3)}")
```

```
Average Coverage Probability of Beta in Confidence Sets:
Naive: 0.802
Robust: 0.92
Bootstrap: 0.932
```

Across $m = 500$ Monte-Carlo replications, the heteroskedasticity-robust and bootstrap intervals cover the true β much more reliably than the naive interval. The naive procedure falls short because its standard errors ignore the x_i-dependent variance, leading to under-coverage. Both the robust and bootstrap approaches stay close to the

nominal 95% target in this design.

Takeaways

- Analytical derivations confirm the conditional Gaussian distribution required in part (a).
- Simulation diagnostics reveal how heteroskedasticity manifests in the data.
- Comparing confidence intervals highlights the pitfalls of naive standard errors and the practical value of robust or bootstrap corrections.
- The bootstrap percentile interval performs competitively in this setting, delivering coverage on par with the analytic robust alternative.