Worksheet 19

Name: Jeong Yong Yang, Junho Son UID: U95912941, U64222022

Topics

- Linear Model Evaluation

Linear Model Evaluation

Notice that R^2 only increases with the number of explanatory variables used. Hence the need for an adjusted R^2 that penalizes for insignificant explanatory variables.

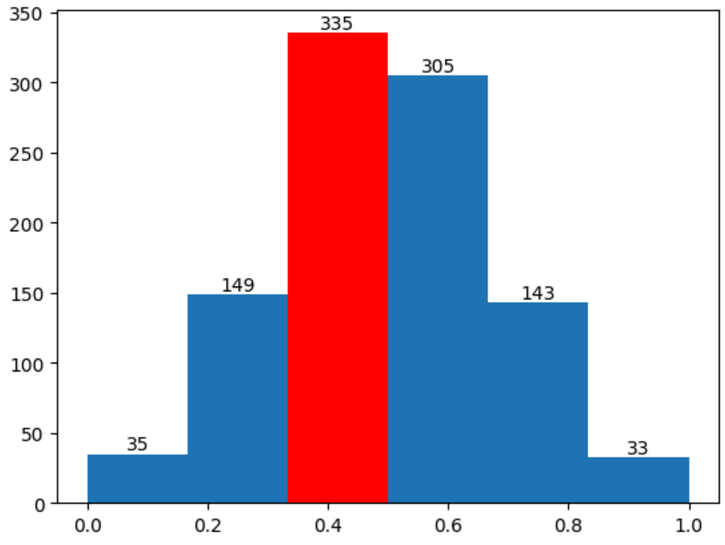
```python
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

SAMPLE_SIZE = 100
beta = [1, 5]
X = -10.0 + 10.0 * np.random.random(SAMPLE_SIZE)
Y = beta[0] + beta[1] * X + np.random.randn(SAMPLE_SIZE)

for i in range(1, 15):
    X_transform = PolynomialFeatures(degree=i, include_bias=False).fit_transform(X.reshape(-1, 1))
    model = LinearRegression()
    model.fit(X_transform, Y)
    print(model.score(X_transform, Y))
```

```
0.9954062656470016
0.9954159177864946
0.9955103152871462
0.9955994798701692
0.9957012434751283
0.9957335582161672
0.9958795013561041
0.9961007167628202
0.9961144218567842
0.9961354512007631
0.9961497434160319
0.9961650184057208
0.9961670316961133
0.9961347497238493
```

a) Hypothesis Testing Sandbox (follow along in class) Notes

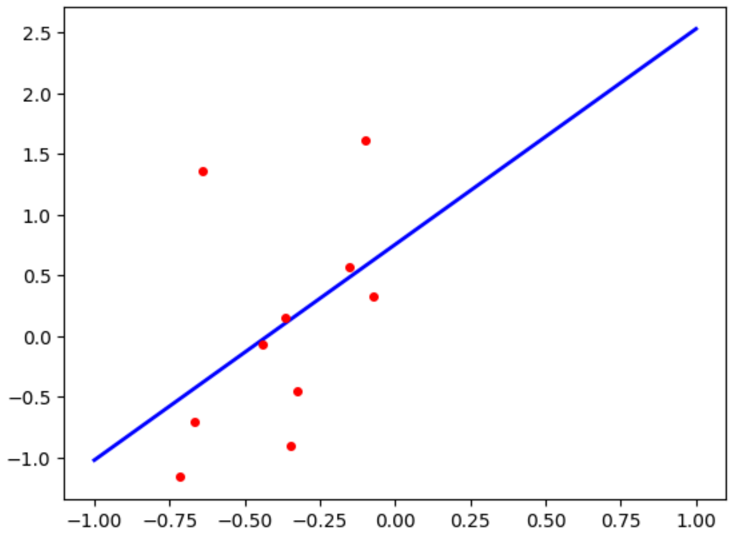
```python
import numpy as np
from scipy.stats import binom
import matplotlib.pyplot as plt

flips = [1, 0, 0, 1, 0]

def num_successes(flips):
    return sum(flips)

print(binom.pmf(num_successes(flips), len(flips), 1/2))

SAMPLE_SIZE = 5
flips = [np.random.choice([0, 1]) for _ in range(SAMPLE_SIZE)]
print(flips)
print(binom.pmf(num_successes(flips), SAMPLE_SIZE, 1/2))

p_est = []

for _ in range(1000):
    flips = [np.random.choice([0, 1]) for _ in range(SAMPLE_SIZE)]
    p_est.append(sum(flips) / SAMPLE_SIZE)

fig, ax = plt.subplots()
_, bins, patches = ax.hist(p_est, bins=SAMPLE_SIZE + 1)
p = np.digitize([2/5], bins)
patches[p[0]-1].set_facecolor('r')
ax.bar_label(patches)
plt.show()
```

```
0.31249999999999983
[0, 0, 0, 1, 1]
0.31249999999999983
```



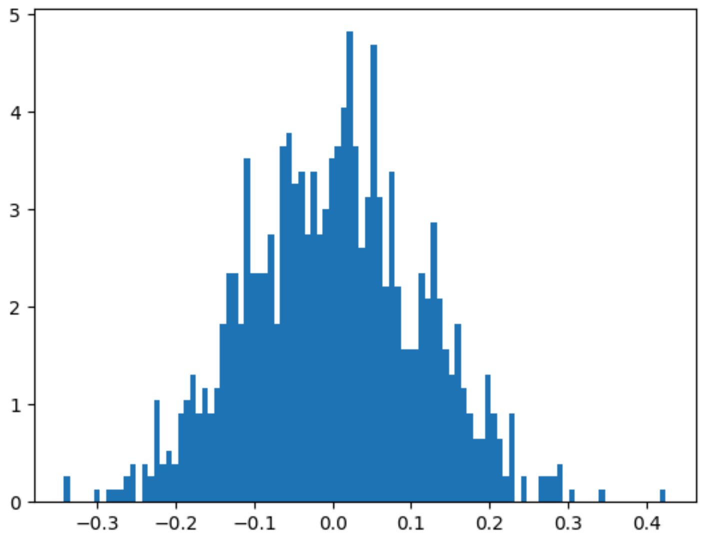
b) Plot a data set and fitted line through the point when there is no relationship between X and y.

```
import numpy as np
import matplotlib.pyplot as plt

SAMPLE_SIZE = 10

xlin = -1.0 + 1.0 * np.random.random(SAMPLE_SIZE)
y = 0.0 + np.random.randn(SAMPLE_SIZE)

intercept = np.ones(np.shape(xlin)[0])
X = np.array([intercept, xlin]).T
beta = np.linalg.inv(X.T @ X) @ X.T @ y

xplot = np.linspace(-1,1,20)
yestplot = beta[0] + beta[1] * xplot
plt.plot(xplot, yestplot,'b-',lw=2)
plt.plot(xlin, y,'ro',markersize=4)
plt.show()
```



c) Using the above code, plot a histogram of the parameter estimates for the slope after generating `1000` independent datasets. Comment on what the plot means. Increase the sample size to see what happens to the plot. Explain.

```
SAMPLE_SIZE = 1000
NUM_TRIALS = 1000
beta_hist = []
for _ in range(NUM_TRIALS):
    xlin = -1.0 + 1.0 * np.random.random(SAMPLE_SIZE)
    y = 0.0 + np.random.randn(SAMPLE_SIZE)

    intercept = np.ones(np.shape(xlin)[0])
    X = np.array([intercept, xlin]).T
    beta = np.linalg.inv(X.T @ X) @ X.T @ y
    beta_hist.append(beta[1])

fig, ax = plt.subplots()
ax.hist(beta_hist, bins=100, density=True)
plt.show()
```



For a sample size of 100, most of the beta is 0, which confirms our estimation but there are some betas that are higher or lower than 0, indicating that there is a relationship between x and y. As the sample size decreases, the values of slopes get much steeper. As the sample size increases, the values of slopes get much flatter.

d) We know that:

$$\hat{\beta} - \beta \sim \mathcal{N}(0, \sigma^2 (X^T X)^{-1})$$

thus for each component k of $\hat{\beta}$ (here there are only two - one slope and one intercept)

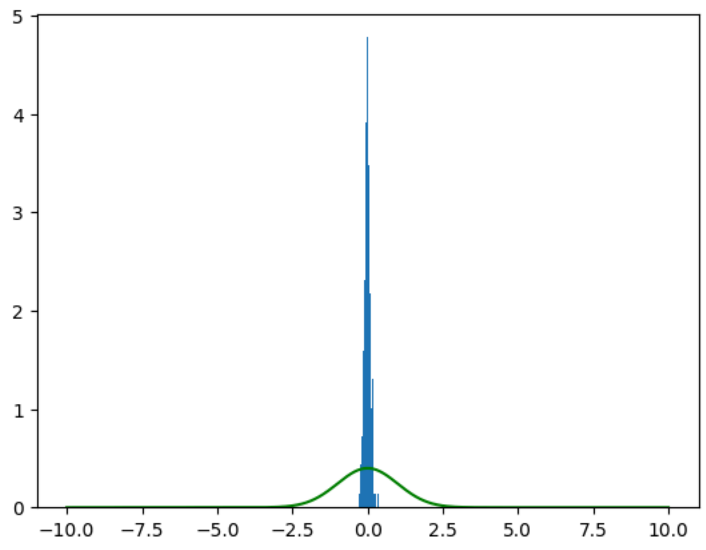
$$\hat{\beta}_k - \beta_k \sim \mathcal{N}(0, \sigma^2 S_{kk})$$

where S_{kk} is the k^{th} diagonal element of $(X^T X)^{-1}$. Thus, we know that

$$z_k = \frac{\hat{\beta}_k - \beta_k}{\sqrt{\sigma^2 S_{kk}}} \sim \mathcal{N}(0, 1)$$

Verify that this is the case through a simulation and compare it to the standard normal pdf by plotting it on top of the histogram.

```
from scipy.stats import norm

beta_hist = []
for _ in range(1000):
    xlin = -1.0 + 1.0 * np.random.random(SAMPLE_SIZE)
    y = 0.0 + np.random.randn(SAMPLE_SIZE)

    intercept = np.ones(np.shape(xlin)[0])
    X = np.array([intercept, xlin]).T
    beta = np.linalg.inv(X.T @ X) @ X.T @ y
    beta_hist.append(beta[1])

xs = np.linspace(-10,10,1000)
fig, ax = plt.subplots()
ax.hist(beta_hist, bins=100, density=True)
ax.plot(xs, norm.pdf(xs), color='green')
plt.show()
```

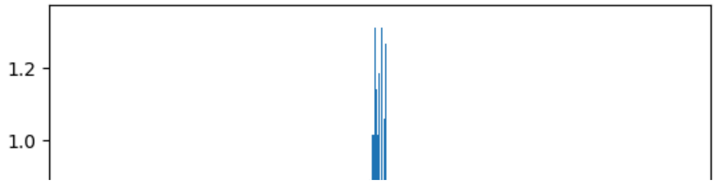


e) Above we normalized $\hat{\beta}$ by subtracting the mean and dividing by the standard deviation. While we know that the estimate of beta is an unbiased estimator, we don't know the standard deviation. So in practice when doing a hypothesis test where we want to assume that $\beta = 0$, we can simply use $\hat{\beta}$ in the numerator. However we don't know the standard deviation and need to use an unbiased estimate of the standard deviation instead. This estimate is the standard error `s`

$$s = \sqrt{\frac{RSS}{n-p}}$$

where p is the number of parameters beta (here there are 2 - one slope and one intercept). This normalized $\hat{\beta}$ can be shown to follow a t-distribution with `n-p` degrees of freedom. Verify this is the case with a simulation.

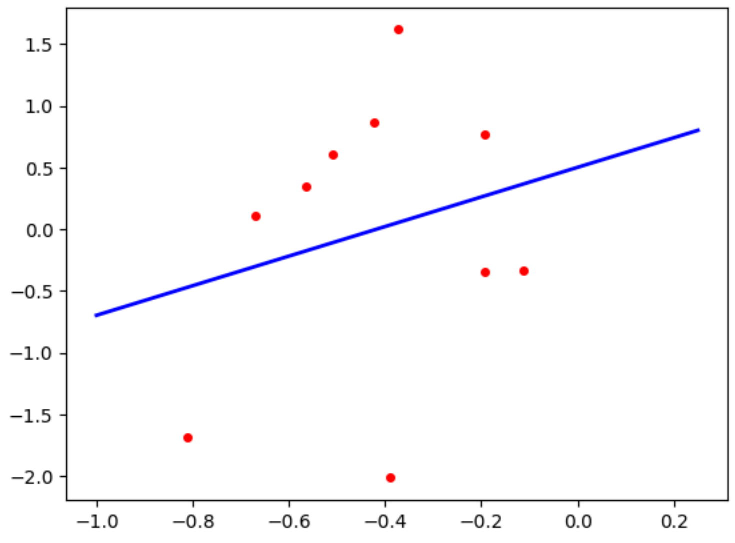
```
from scipy.stats import t

SAMPLE_SIZE = 100
NUM_TRIALS = 1000

def standard_error(ytrue, ypred):
    length = len(ytrue)
    residuals = (ytrue - ypred) ** 2
    rss = 0
    for x in residuals:
      rss += x
    return np.sqrt(rss / (length - 2))

beta_hist = []

for _ in range(NUM_TRIALS):
    xlin = -1.0 + 1.0 * np.random.random(SAMPLE_SIZE)
    y = 0.0 + np.random.randn(SAMPLE_SIZE)

    intercept = np.ones(np.shape(xlin)[0])
    X = np.array((intercept, xlin)).T
    beta = np.linalg.inv(X.T @ X) @ X.T @ y
    standardError = standard_error(y, X @ beta)
    beta_hist.append(beta[1])

xs = np.linspace(-10,10,1000)
fig, ax = plt.subplots()
ax.hist(beta_hist, bins=100, density=True)
ax.plot(xs, t.pdf(xs, SAMPLE_SIZE - 2), color='red')
plt.show()
```



f) You are given the following dataset:

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.array([-0.1920605, -0.11290798, -0.56434374, -0.67052057, -0.19233284, -0.42403586, -0.8114285, -0.38986946, -0.37384161, -0.509302
y = np.array([-0.34063108, -0.33409286, 0.34245857, 0.11062295, 0.76682389, 0.86592388, -1.68912015, -2.01463592, 1.61798563, 0.60557414])

intercept = np.ones(np.shape(x)[0])
X = np.array([intercept, x]).T
beta_hat = np.linalg.inv(X.T @ X) @ X.T @ y

print(beta_hat)

xplot = np.linspace(-1,.25,20)
yestplot = beta_hat[0] + beta_hat[1] * xplot
plt.plot(xplot, yestplot,'b-',lw=2)
plt.plot(x, y,'ro',markersize=4)
plt.show()

```

```
[0.50155603 1.19902827]
```



what is the probability of observing a dataset at least as extreme as the above assuming $\beta = 0$?

```python
length = len(x)

ypred = X @ beta_hat
standardError = standard_error(y, ypred)

tScore = beta_hat[1] / standardError

p_value = 2 * (1 - t.cdf(np.abs(tScore), df = length - 2))

print(f"The probability of observing a dataset at least as extreme as the above assuming beta = 0 is {p_value}")
```

```
The probability of observing a dataset at least as extreme as the above assuming beta = 0 is 0.335417218459404
```

```
코딩을 시작하거나 AI로 코드를 생성하세요.
```