6209130209129209130208176209130-2

June 13, 2023

[9]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tqdm
import re
import itertools
from scipy.stats import ttest_ind
from scipy.stats import t
from scipy.stats import norm
from math import factorial
```

1 #1

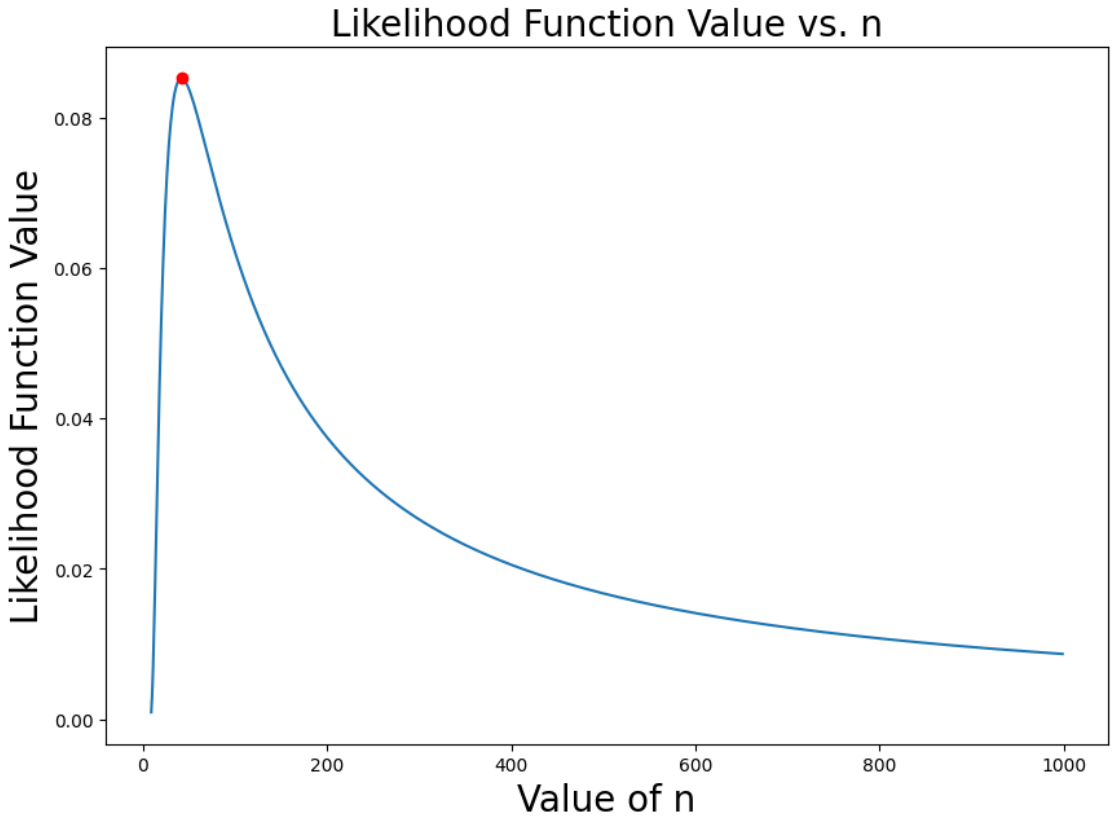
a)

[ ]:
```
!jt -r
```

```
/bin/bash: jt: command not found
```

[4]:
```
def likelihood(n):
    values = np.arange(2, 11)
    a = np.prod((n - values + 2) / n)
    return a * 9 / n

n_values = np.arange(9, 1000)
likelihood_values = [likelihood(n) for n in n_values]

plt.figure(figsize=(10, 7))
plt.plot(n_values, likelihood_values)
plt.xlabel('Value of n', fontsize=20)
plt.ylabel('Likelihood Function Value', fontsize=20)
plt.title('Likelihood Function Value vs. n', fontsize=20)

max_likelihood = np.max(likelihood_values)
max_likelihood_index = np.argmax(likelihood_values)
max_likelihood_n = n_values[max_likelihood_index]
```

```
plt.plot(max_likelihood_n, max_likelihood, 'ro')
plt.show()

print(f"The value of n with the maximum likelihood is {max_likelihood_n}")
```

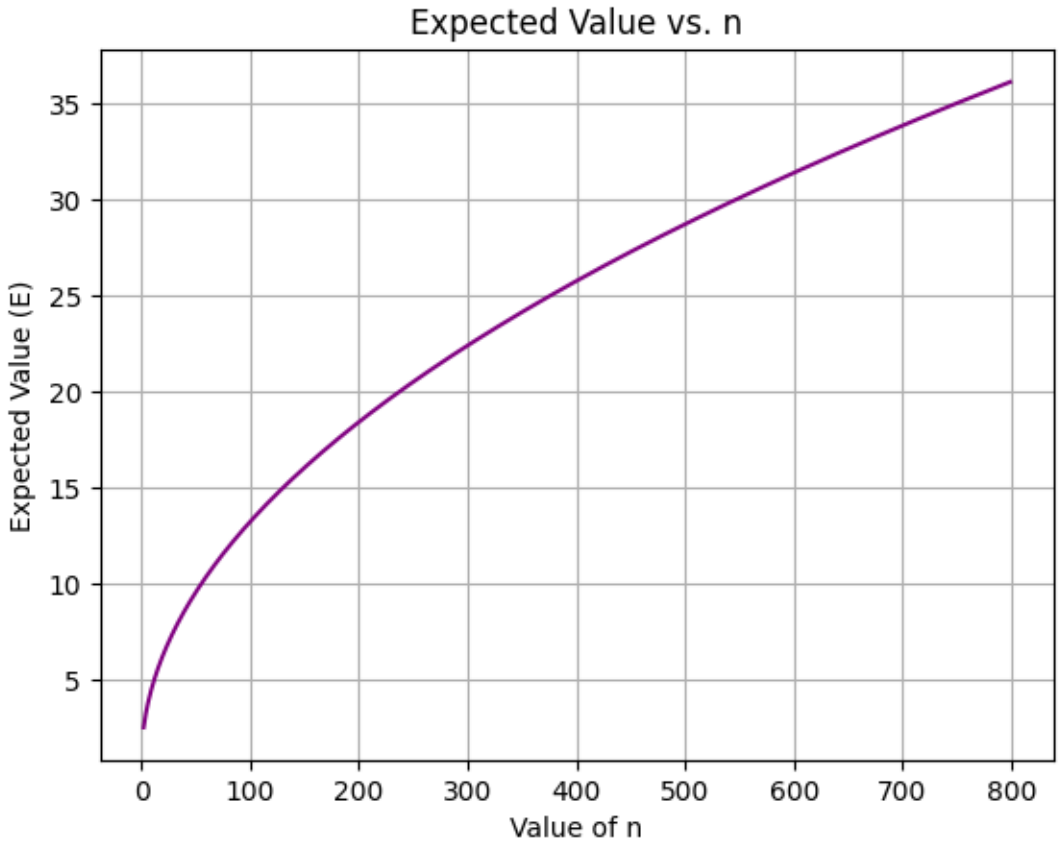



```
The value of n with the maximum likelihood is 42
```

b)

[3]:
```
def find_duplicate(n):
    taxers = np.arange(1, n + 1)
    np.random.shuffle(taxers)
    duplicate = np.argmax(np.bincount(taxers))
    return duplicate

def calculate_P(k, n):
    x_values = np.arange(2, k+1)
    numerator = np.prod((n - x_values + 2) / n)
    probability = numerator * (k - 1) / n
    return probability

def calculate_E(n):
```

```
    k_values = np.arange(2, n+2)
    probabilities = np.vectorize(calculate_P)(k_values, n)
    expected_value = np.sum(k_values * probabilities)
    return expected_value

n_gen = np.arange(2, 800)
E_n = np.vectorize(calculate_E)(n_gen)

plt.plot(n_gen, E_n, color='purple')
plt.grid(True)
plt.xlabel('Value of n')
plt.ylabel('Expected Value (E)')
plt.title('Expected Value vs. n')
plt.show()

n = 100
result = find_duplicate(n)
print(f"The duplicate element in the randomly shuffled array is: {result}")
```

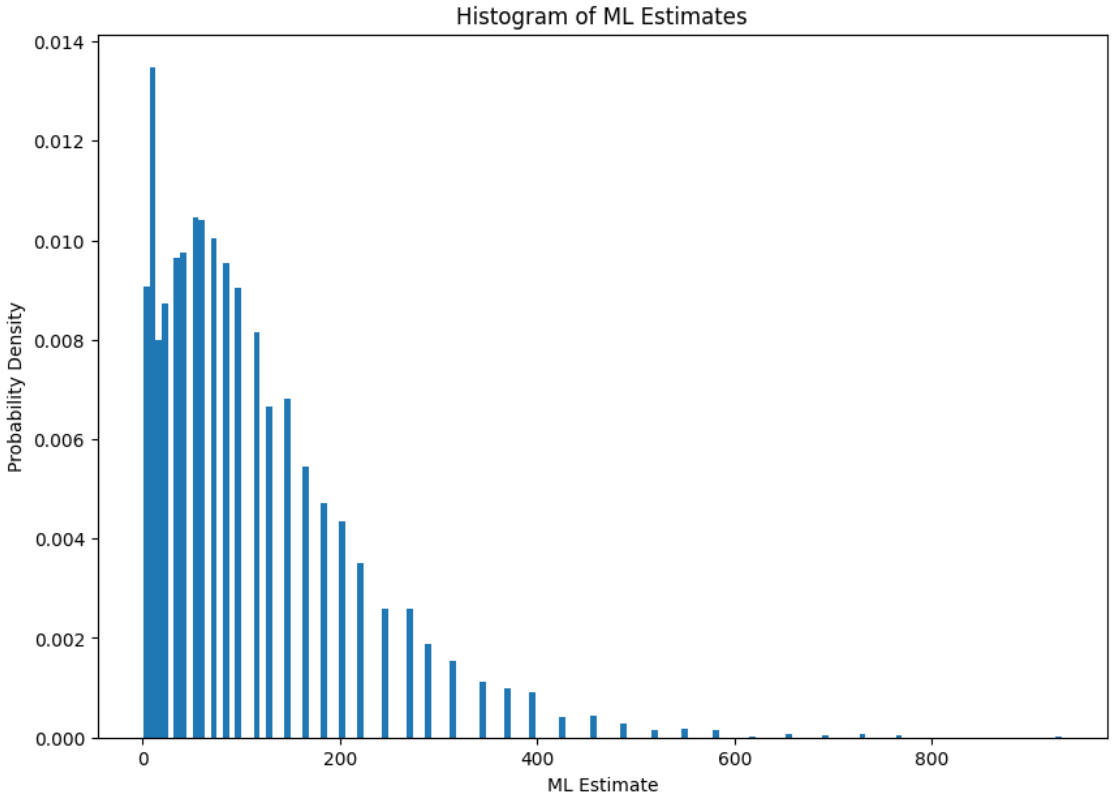


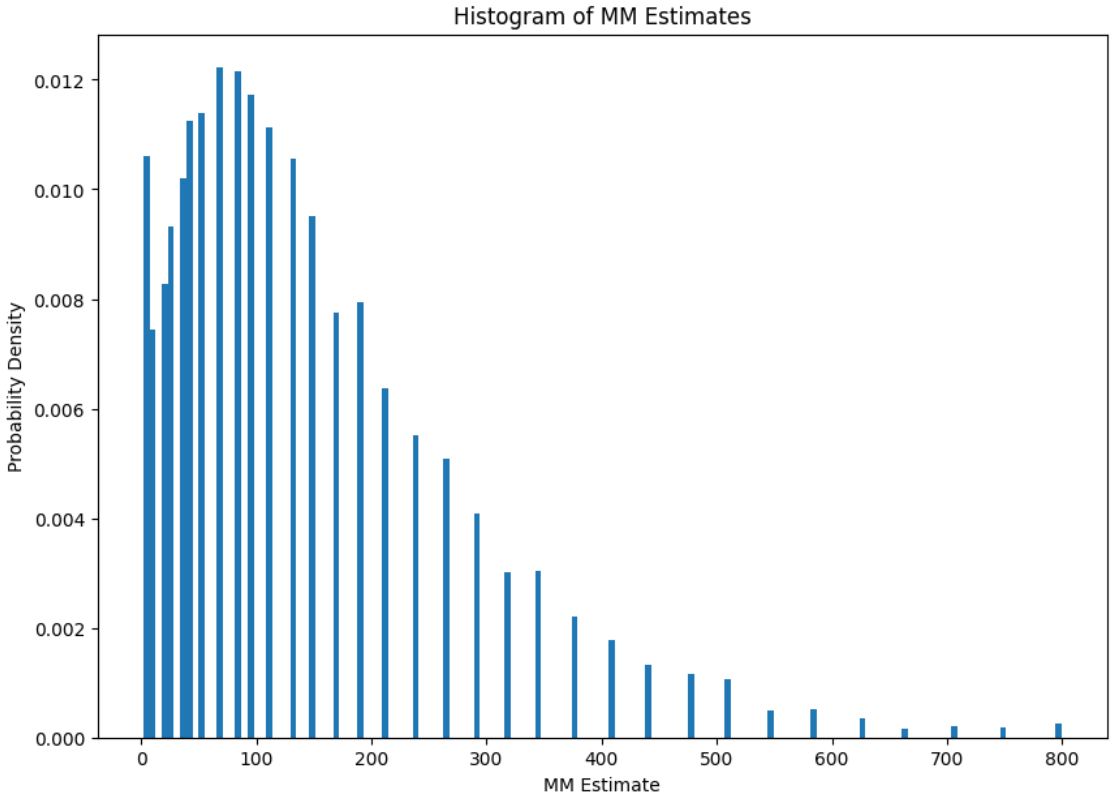

```
The duplicate element in the randomly shuffled array is: 1
```

c)

[5]:
```
np.random.seed(42)

def calculate_k(n):
    taxers = np.arange(1, n+1)
    real = np.random.choice(taxers, len(taxers))
    seen_elements = set()
    for element in real:
        if element in seen_elements:
            duplicate = element
            break
        seen_elements.add(element)
    return np.where(real == duplicate)[0][1] + 1

k_s = np.array([calculate_k(100) for _ in range(10**4)])

ML_est = []
MM_est = []

for k in tqdm.tqdm(k_s):
    n_ki = np.arange(k-1, 1000)
    probabilities = np.array([calculate_P(k, n) for n in range(k-1, 1000)])
    max_indices = np.where(probabilities == np.max(probabilities))[0]
    max_n_ki = n_ki[max_indices]
    ML_est.append(max_n_ki[0])

for k in k_s:
    abs_diff = np.abs(E_n - k)
    min_indices = np.where(abs_diff == np.min(abs_diff))[0]
    n_hat = n_gen[min_indices][0]
    MM_est.append(n_hat)
```

[12]:
```
plt.figure(figsize=(10, 7))
plt.hist(ML_est, bins=150, density=True)
plt.xlabel('ML Estimate')
plt.ylabel('Probability Density')
plt.title('Histogram of ML Estimates')
plt.show()

plt.figure(figsize=(10, 7))
plt.hist(MM_est, bins=150, density=True)
plt.xlabel('MM Estimate')
plt.ylabel('Probability Density')
plt.title('Histogram of MM Estimates')
plt.show()
```


Histogram of ML Estimates
Probability Density
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
200
400
600
800
ML Estimate



[13]:
```
bias_mm = (np.abs(np.mean(MM_est) - 100))
bias_ml = (np.abs(np.mean(ML_est) - 100))
var_mm = np.var(MM_est)
var_ml = np.var(ML_est)
mse_mm = np.mean((MM_est - np.mean(MM_est))**2)
mse_ml = np.mean((ML_est - np.mean(ML_est))**2)

result = pd.DataFrame({
    '    ': ['ML', 'MM'],
    '     ': [bias_ml, bias_mm],
    '     ': [var_ml, var_mm],
    'MSE': [mse_ml, mse_mm]
})
result.set_index('    ', inplace=True)
result
```

[13]:

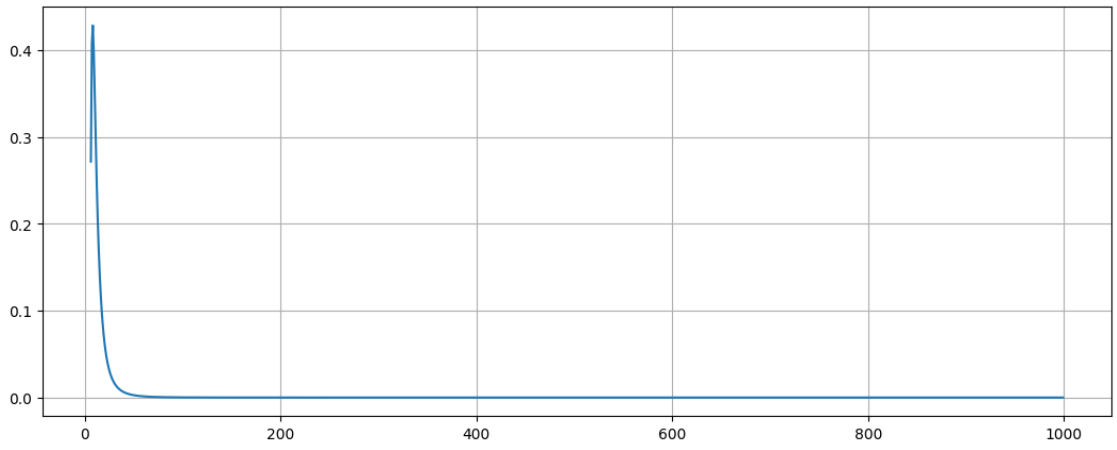
			MSE
ML	3.4775	8729.560894	8729.560894
MM	25.7458	14200.499182	14200.499182

2 #2

a)

[3]:
```
def calculate_P_yand(k, n, m=10):
    prob = np.prod([(n-i)/n for i in range(1, k)])
    s = 0
    combinations = itertools.combinations_with_replacement(np.arange(1, k+1), m
 ↪- k)
    for el in combinations:
        prob_repeat = np.prod(el)
        s += prob_repeat
    prob *= s / (n ** (m - k))
    return prob

plt.figure(figsize=(13, 5))

obl_opred = np.arange(6, 1000)
probs = [calculate_P_yand(6, n) for n in obl_opred]

plt.plot(obl_opred, probs)
plt.grid(True)

max_prob_index = np.argmax(probs)
max_prob = probs[max_prob_index]
max_prob_estimate = obl_opred[max_prob_index]
print(f"ML-    : {max_prob_estimate}")
```

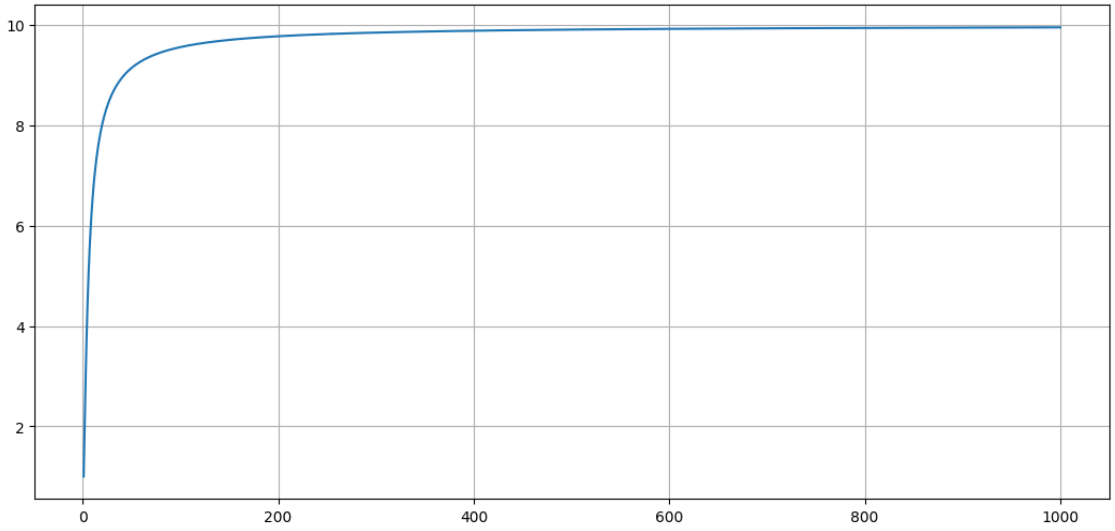
```
ML-    : 8
```



b)

[4]:
```python
def calculate_E_names(n):
    sum_of_p = sum(k * calculate_P_yand(k, n, m=10) for k in range(1, 11))
    return sum_of_p

plt.figure(figsize=(13, 6))

n_vals = np.arange(1, 1001)
E_all = np.array([calculate_E_names(i) for i in n_vals])

plt.plot(n_vals, E_all)
plt.grid(True)

min_distance_index = np.argmin(np.abs(E_all - 6))
min_distance_estimate = n_vals[min_distance_index]
print(f"MM-    : {min_distance_estimate}")
```

```
MM-    : 8
```



c)

[ ]:
```python
def calculate_g(n, m):
    n_names = np.arange(1, n + 1)
    sample_after_10_calls = np.random.choice(n_names, size=m)
    return np.unique(sample_after_10_calls).shape[0]

np.random.seed(19)
k_names = [calculate_g(20, 10) for _ in range(10 ** 4)]

ML_est_names = []
```

```
for k in tqdm.tqdm(k_names):
    n_general = np.arange(k, 1000)
    estimates = [calculate_P_yand(k, n) for n in range(k, 1000)]
    ML_est_names.append(n_general[estimates == np.max(estimates)][0])

ML_est_names = np.array(ML_est_names)
plt.hist(ML_est_names[ML_est_names < 100], density=True, bins=10)
plt.show()
```

[ ]:
```
MM_est_names = []
for k in k_names:
    n_hat_names = n_vals[np.argmin(np.abs(E_all - k))]
    MM_est_names.append(n_hat_names)

MM_est_names = np.array(MM_est_names)
plt.hist(MM_est_names[MM_est_names < 100], bins=10, density=True)
plt.show()

MM = MM_est_names[MM_est_names < 100]
ML = ML_est_names[ML_est_names < 100]

bias_mm_names = abs(100 - np.mean(MM))
bias_ml_names = abs(100 - np.mean(ML))

var_mm_names = np.sum((MM - 100) ** 2) / 10 ** 4
var_ml_names = np.sum((ML - 100) ** 2) / 10 ** 4

mse_mm_names = np.sum((MM - np.mean(MM)) ** 2) / 10 ** 4
mse_ml_names = np.sum((ML - np.mean(ML)) ** 2) / 10 ** 4

result_names = pd.DataFrame({
    '    ': ('ML', 'MM'),
    '     ': (bias_ml_names, bias_mm_names),
    '      ': (var_ml_names, var_mm_names),
    'MSE': (mse_ml_names, mse_mm_names)
})
result_names.set_index('    ')
```

3 #3

a)

[7]:
```
np.random.seed(42)

def CI_mean(x, n):
    means = np.mean(x, axis=1)
```

```
    std = np.std(x, ddof=1, axis=1) / np.sqrt(20)

    lower_bounds = means - 1.96 * std
    upper_bounds = means + 1.96 * std

    res_ci = np.logical_and(lower_bounds <= n, upper_bounds >= n)
    return np.mean(res_ci)

def naive_bootstrap_mean(x, n):
    l_naive = []
    for sample in tqdm.tqdm(x):
        boot_indices = np.random.choice(np.arange(20), size=(10**4, 20))
        means_boot = np.mean(sample[boot_indices], axis=1)

        quantile_l = np.percentile(means_boot, 2.5)
        quantile_r = np.percentile(means_boot, 97.5)

        res_naive = np.logical_and(quantile_l <= n, quantile_r >= n)
        l_naive.append(res_naive)
    return np.mean(l_naive)

def t_bootstrap_mean(x, n):
    l_t = []
    for sample in tqdm.tqdm(x):
        boot_indices = np.random.choice(np.arange(20), size=(10**4, 20))

        means_boot = np.mean(sample[boot_indices], axis=1)
        se_boot = np.std(sample[boot_indices], axis=1, ddof=1) / np.sqrt(20)
        mean_sample = np.mean(sample)

        quantile_l = np.percentile((means_boot - mean_sample) / se_boot, 2.5)
        quantile_r = np.percentile((means_boot - mean_sample) / se_boot, 97.5)

        res_t = np.logical_and(mean_sample - quantile_r * np.std(sample,␣
 ↪ddof=1) / np.sqrt(20) <= n, n <= mean_sample - quantile_l * np.std(sample,␣
 ↪ddof=1) / np.sqrt(20))
        l_t.append(res_t)
    return np.mean(l_t)

samples = np.random.exponential(1, size=(10**4, 20))
```

:

[21]:
```
np.random.seed(42)
print('CI:',CI_mean(samples,1))
print('naive_bootstrap:',naive_bootstrap_mean(samples,1))
print('t_bootstrap:',t_bootstrap_mean(samples,1))
```

```
CI: 0.9036
100%|      | 10000/10000 [00:46<00:00, 216.43it/s]
naive_bootstrap: 0.9032
100%|      | 10000/10000 [01:01<00:00, 161.78it/s]
t_bootstrap: 0.9474
```

b)

[ ]:
```
np.random.seed(42)

t_samples = np.random.standard_t(3, size=(10**4,20))

print('CI:',CI_mean(t_samples,0))
print('naive_bootstrap:',naive_bootstrap_mean(t_samples,0))
print('t_bootstrap:',t_bootstrap_mean(t_samples,0))
```

```
CI: 0.9438
100%|      | 10000/10000 [00:47<00:00, 208.52it/s]
naive_bootstrap: 0.9195
 97%|      | 9737/10000 [01:02<00:01, 164.07it/s]
```

c) t- , .
( ), t- .
, .

4 #4

[ ]:
```
sample = pd.read_csv('    .csv', sep = ';')

pattern_1 = r'^[           ]'
pattern_2 = r'^[     ]'

sample_cons = sample[sample['    '].str.contains(pattern_1, flags=re.
 ↪IGNORECASE, regex=True)]
sample_vow = sample[sample['    '].str.contains(pattern_2, flags=re.IGNORECASE,␣
 ↪regex=True)]

x = sample_cons['     ']
y = sample_vow['     ']

x = np.array(x)
y = np.array(y)
```

a)

[ ]:
```
S , pvalue_welch = ttest_ind(x, y, equal_var=False, alternative='two-sided')

print(f'p_value = {round(pvalue_welch, 2)} =>          ')
```

b)

[ ]:
```
np.random.seed(42)

mean_real = np.mean(x) - np.mean(y)
se_real = np.sqrt((x.std()**2)/x.shape[0] + (y.std()**2)/y.shape[0])

boot_indices_x = np.random.choice(np.arange(283), size=(10**4, 283))
boot_indices_y = np.random.choice(np.arange(49), size=(10**4, 49))

means_bootstrap_diff = np.mean(x[boot_indices_x], axis=1) - np.
 ↪mean(y[boot_indices_y], axis=1)

quantile_l_naive = np.percentile(means_bootstrap_diff, 2.5)
quantile_r_naive = np.percentile(means_bootstrap_diff, 97.5)

print(f'                  : {np.logical_and( quantile_l_naive <= mean_real,␣
 ↪mean_real <= quantile_r_naive)}.')
print('   :               .')
print(f'p_value: {2*(np.min([np.mean((mean_real < means_bootstrap_diff)), np.
 ↪mean(mean_real >= means_bootstrap_diff)]))} =>          ')
```

```
                    : True.
  :                    .
     p_value: 0.9964 =>                          5%.
```

c)

[ ]:
```
np.random.seed(42)

mean_real = np.mean(x) - np.mean(y)

boot_indices_x = np.random.choice(np.arange(283), size=(10**4, 283))
boot_indices_y = np.random.choice(np.arange(49), size=(10**4, 49))

means_bootstrap = np.mean(x[boot_indices_x], axis=1) - np.
 ↪mean(y[boot_indices_y], axis=1)
se_bootstrap = np.sqrt(((np.std(x[boot_indices_x], axis=1, ddof=1))**2 /(283)
                + np.std(y[boot_indices_y], axis=1, ddof=1))**2 /(49))

boots_sample = (means_bootstrap - mean_real) / se_bootstrap
```

```
quantile_l_t = np.percentile(boots_sample, 2.5)
quantile_r_t = np.percentile(boots_sample, 97.5)

print(f'                        : {np.logical_and( quantile_l_t <= S, S <=␣
 ↪quantile_r_t)}')
print('   :                       .')
print(f'p_value: {2*(np.min([np.mean((S < boots_sample)), np.mean(S >=␣
 ↪boots_sample)]))} =>          ')
```

```
                        : True
  :                        .
      p_value: 0.3926 =>                              5%.
```

d)

[ ]:
```
np.random.seed(19)

a1 = np.zeros_like(y)
a2 = np.ones_like(x)
a = np.hstack((a2, a1))
w = np.hstack((x, y))
deltas_list = []

for i in range(10**4):
    a_p = np.random.permutation(a)
    delta_hat = np.mean(w[a_p == 1]) - np.mean(w[a_p == 0])
    deltas_list.append(delta_hat)

quantile_l_permutation = np.percentile(deltas_list, 2.5)
quantile_r_permutation = np.percentile(deltas_list, 97.5)

deltas_list = np.array(deltas_list)

print(f'                    : {np.logical_and(quantile_l_permutation <=␣
 ↪mean_real, quantile_r_permutation >= mean_real)}.')
print('   :       0          .')
print(f'p_value: {2*(np.min([np.mean(( mean_real < deltas_list)), np.mean(␣
 ↪mean_real >= deltas_list)]))} =>          ')
```

```
                        : True.
  :                        .
      p_value: 0.3738 =>                              5%.
```

5 #5

[ ]:
```
med_more_cons = sample[(sample['    '].str.contains(pattern_1, flags=re.
 ↪IGNORECASE, regex=True)) &
                       (sample['     '] > np.median(sample['     ']))].count()
med_less_cons = sample[(sample['    '].str.contains(pattern_1, flags=re.
 ↪IGNORECASE, regex=True)) &
                       (sample['     '] <= np.median(sample['     ']))].count()
med_more_vow = sample[(sample['    '].str.contains(pattern_2, flags=re.
 ↪IGNORECASE, regex=True)) &
                       (sample['     '] > np.median(sample['     ']))].count()
med_less_vow = sample[(sample['    '].str.contains(pattern_2, flags=re.
 ↪IGNORECASE, regex=True)) &
                       (sample['     '] <= np.median(sample['     ']))].count()

matrix = np.array([[med_more_cons[0], med_less_cons[0]], [med_more_vow[0],␣
 ↪med_less_vow[0]]])

index_labels = ['Consonant', 'Vowel']
column_labels = ['> Median', '<= Median']

contingency_matrix = pd.DataFrame(matrix, index=index_labels,␣
 ↪columns=column_labels)
contingency_matrix
```

a)

[ ]:
```
rv = norm(loc = 0, scale = 1)

stat_hat_odds = np.log(21/28) - np.log(145/138)
se_hat_odds = np.sqrt(1/145+1/138+1/21+1/28)

stat_observed_standard = (stat_hat_odds - 0)/(se_hat_odds)
print(f' CI: {np.exp([stat_hat_odds - 1.96*se_hat_odds, stat_hat_odds + 1.
 ↪96*se_hat_odds])} \n p_value: {2*np.min([rv.cdf(stat_observed_standard),␣
 ↪1-rv.cdf(stat_observed_standard)])}')
```

b)

[ ]:
```
stat_hat_risk = np.log(21/49) - np.log(145/283)
se_hat_risk = np.sqrt(1/145 - 1/283 + 1/21 - 1/49)

stat_observed_risk_standard = (stat_hat_risk - 0)/(se_hat_risk)

print(f' CI: {np.exp([stat_hat_risk - 1.96*se_hat_risk, stat_hat_risk + 1.
 ↪96*se_hat_risk])} ')
```

```
print(f' p-value: {2*np.min([rv.cdf(stat_observed_risk_standard), 1-rv.
 ↪cdf(stat_observed_risk_standard)])}')
```

```
CI: [0.59374922 1.17836612]
p-value: 0.3070947928050546
```

c)

[ ]:
```
np.random.seed(42)

odds = []

p_pass_cons = contingency_matrix.iloc[0][0] / (contingency_matrix.iloc[0][0] +␣
 ↪contingency_matrix.iloc[0][1])
p_pass_vow = contingency_matrix.iloc[1][0] / (contingency_matrix.iloc[1][0] +␣
 ↪contingency_matrix.iloc[1][1])

OR_obs = (p_pass_cons / (1 - p_pass_cons)) / (p_pass_glas / (1 - p_pass_glas))

for _ in range(10**4):

    x_cons = np.random.choice(sample_cons['     '], size=sample_cons.shape[0])
    y_cons = np.random.choice(sample_vow['     '], size=sample_vow.shape[0])

    med_more_cons_b = np.sum(x_cons > np.median(sample['     ']))
    med_less_cons_b = np.sum(x_cons <= np.median(sample['     ']))
    med_more_vow_b = np.sum(y_cons > np.median(sample['     ']))
    med_less_vow_b = np.sum(y_cons <= np.median(sample['     ']))

    matrix = np.array([[med_more_cons_b, med_less_cons_b], [med_more_vow_b,␣
 ↪med_less_vow_b]])

    p_a = matrix[0][0] / np.sum(matrix[0])
    p_b = matrix[1][0] / np.sum(matrix[1])

    odd = (p_a / (1 - p_a)) / (p_b / (1 - p_b))
    odds.append(odd)

odds = np.array(odds)

ql = np.percentile(odds, 2.5)
qr = np.percentile(odds, 97.5)

print(f'                          : {np.logical_and(OR_obs <= qr, OR_obs >= ql)}')
print('   :              .')
print(f'p-value: {2*(np.min([np.mean(( OR_obs < odds)), np.mean( OR_obs >=␣
 ↪odds)]))} =>         ')
```

: True
: .
p-value: 0.9898 => .

6 #6

a)

[ ]:
```
sample_6 = sample.copy()
sample_6['    '] = sample_6['    '].apply(len)
betas = sample_6['     '].mean()/sample_6['    '].mean()
print(betas)
sample_6
```

b)

[ ]:
```
np.random.seed(19)

x_len = sample_6['    ']
y_res = sample_6['     ']
corr_obs = np.corrcoef(x_len, y_res)[0][1]

corrs_dist = []

for _ in range(10**4):
    x_ = np.random.permutation(x_len)
    corr_hat = np.corrcoef(x_, y_res)[0][1]
    corrs_dist.append(corr_hat)

corrs_dist = np.array(corrs_dist)

q_corr_r = np.quantile(corrs_dist, 0.975)
q_corr_l = np.quantile(corrs_dist, 0.025)

print(f'                  : {np.logical_and(0 <= q_corr_r, 0 >= q_corr_l)}.')
print(f'p-value: {2*(np.min([np.mean(( corr_obs < corrs_dist)), np.mean(␣
 ↪corr_obs >= corrs_dist)]))} =>         ')
```

7 #7-8

[ ]:
```
https://chat.openai.com/share/ea9844e1-2204-40d8-bdb7-5eb5ec577ff9
```

, . : https://youtu.be/AyWM-le_vLE
(- -).
: https://www.youtube.com/@3blue1brown. : https://youtu.be/7ESK5SaP-bc

[ ]:
```
          ,                                          .        : https://youtu.be/
↪AyWM-le_vLE (  -                                                    -      ␣
↪                          ).                  : https://www.youtube.com/
↪@3blue1brown.     : https://youtu.be/7ESK5SaP-bc
```