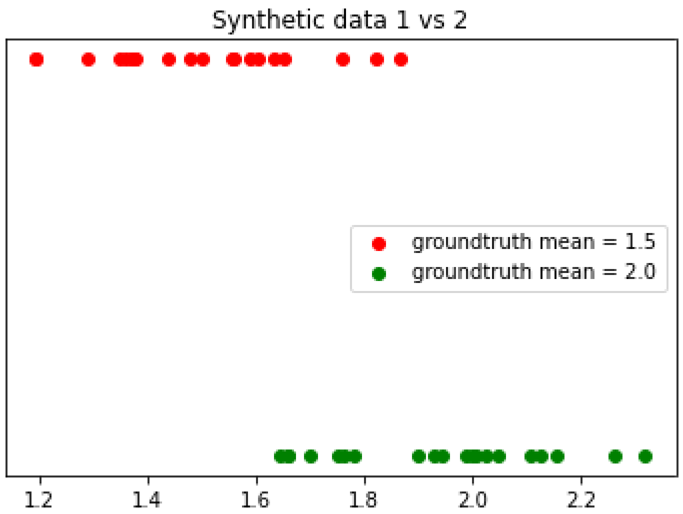
Part 1

In [1]:
```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

rng = np.random.default_rng(seed=22197823)
```

q1.a

In [2]:
```
# generate sample data
std_gt = 0.2
mean_1_gt = 1.5
mean_2_gt = 2.0
N = 20

group1 = rng.normal(loc=mean_1_gt, scale=std_gt, size=N)
group2 = rng.normal(loc=mean_2_gt, scale=std_gt, size=N)

mean_1_calc = group1.mean()
mean_2_calc = group2.mean()
std_1_calc = group1.std()
std_2_calc = group2.std()

print(f'Data 1: mean : {mean_1_calc:.02f}, std: {std_1_calc:.02f}. Expected sum of
print(f'Data 2: mean : {mean_2_calc:.02f}, std: {std_2_calc:.02f}. Expected mean ar
print(f'\nExpected sum of square difference (SSD) from the mean : {(N * std_gt**2)
print(f'Data 1 SSD : {((mean_1_gt - group1)**2).sum():.2f}')
print(f'Data 2 SSD : {((mean_2_gt - group2)**2).sum():.2f}')

plt.scatter(group1, 2 * np.ones(N), color='r', label=f'groundtruth mean = {mean_1_g
plt.scatter(group2, 1 * np.ones(N), color='g', label=f'groundtruth mean = {mean_2_g
plt.yticks([])
plt.legend()
plt.title('Synthetic data 1 vs 2')
plt.show()
```

```
Data 1: mean : 1.50, std: 0.19. Expected sum of sqare: 1.5, 0.2
Data 2: mean : 1.94, std: 0.20. Expected mean and std: 2.0, 0.2

Expected sum of square difference (SSD) from the mean : 0.80
Data 1 SSD : 0.70
Data 2 SSD : 0.86
```



q1.b

In [3]:
```
t_statistic_gt, p_val = stats.ttest_ind(group1, group2)
print(f'T statistic: {t_statistic_gt:.2f}, p-value: {p_val:.12f}')
```

```
T statistic: -7.04, p-value: 0.000000021510
```

In [4]:
```
print('2 sided tail test')
2*stats.t.cdf(t_statistic_gt, 38)
```

```
2 sided tail test
```

Out[4]:
```
2.1510080928959593e-08
```

q1.c i)

In [5]:
```
# for Y = X1 * B1 + X2 * B2 + e, what is the design matrix?
# synthetic data is from 2 groups

# design matrix:
X = np.zeros(shape=(40,2))
X[:20, 0] = 1
X[20:, 1] = 1

Y = np.hstack([group1, group2])

rank = np.linalg.matrix_rank(X)
print(f'rank of C(X) = {rank}')
```

```
rank of C(X) = 2
```

q1.c ii)

In [6]:
```
Px = X @ np.linalg.inv(X.T @ X) @ X.T
print(Px)
print(f'Trace(Px) = {np.trace(Px):.2f}')
```

```
[[0.05 0.05 0.05 ... 0.   0.   0.  ]
 [0.05 0.05 0.05 ... 0.   0.   0.  ]
 [0.05 0.05 0.05 ... 0.   0.   0.  ]
 ...
 [0.   0.   0.   ... 0.05 0.05 0.05]
 [0.   0.   0.   ... 0.05 0.05 0.05]
 [0.   0.   0.   ... 0.05 0.05 0.05]]
Trace(Px) = 2.00
```

q1.c iii)

In [7]:
```
# use Px to find Y_hat
Y_hat = Px @ Y
print(f'error between Y and Y_hat: {(Y - Y_hat).sum():.2f}')
```

```
error between Y and Y_hat: -0.00
```

In [8]:
```
X.shape
```

Out[8]:
```
(40, 2)
```

q1.c iv)

In [9]:
```
d = Px.shape[0]
Rx = np.identity(d) - Px

eps = 1e-9

if np.abs((Rx @ Rx - Rx).sum()) < eps and np.abs((Rx - Rx.T).sum()) < eps:
    print(f'Rx = (I - Px) has passed numerical tests for being a perpendicular proj
else:
    print(f'Rx FAILED a numerical test for being a perpendicular projection operato
```

```
Rx = (I - Px) has passed numerical tests for being a perpendicular projection oper
ator
```

q1.c v)

In [10]:
```
error_hat = Rx @ Y
error_hat

error_space_dim = np.linalg.matrix_rank(Rx)

print(f'error space dim: {error_space_dim}')
text = 'error_hat = ['
for e in error_hat[:-1]:
    text += f'{e:.3f}, '
text += f'{error_hat[-1]:.3f}]'
print(text)
```

```
error space dim: 38
error_hat = [0.090, 0.263, 0.060, 0.001, -0.120, -0.129, -0.305, -0.140, -0.124,
0.106, 0.370, 0.058, 0.154, -0.019, -0.061, 0.324, -0.149, -0.210, -0.303, 0.135,
0.108, 0.169, -0.177, 0.323, 0.216, 0.007, 0.381, -0.279, 0.052, -0.009, -0.039, -
0.240, 0.070, -0.158, -0.280, -0.188, 0.086, 0.189, 0.062, -0.292]
```

q1.c vi)

In [11]:
```
# normalise the vectors then calc the angle
numerator = np.dot(error_hat, Y_hat)
divisor = np.sqrt(np.dot(error_hat, error_hat) * np.dot(Y_hat, Y_hat))
angle = np.arccos(numerator / divisor) / np.pi

print(f'angle between Y_hat and error_hat is {angle:.2f} * pi')
print(f'we expect error_hat and Y_hat to be perpendicular, so the angle should be 0
```

```
angle between Y_hat and error_hat is 0.50 * pi
we expect error_hat and Y_hat to be perpendicular, so the angle should be 0.5 * pi
```

q1.c vii)

In [12]:
```
M = np.linalg.inv(X.T @ X)
beta = M @ X.T @ Y
Y_hat_1 = X @ beta
diff = Y_hat - Y_hat_1
print(f'Difference when calculating Y_hat using Y = X @ Beta: {diff.sum():.2f}')
print(beta)
```

```
Difference when calculating Y_hat using Y = X @ Beta: -0.00
[1.49781955 1.93797768]
```

q1.c viii)

In [13]:
```
numerator = np.dot(error_hat, error_hat)
n = X.shape[0]
divisor = n - np.linalg.matrix_rank(X)
var_hat = numerator / divisor
var_hat
```

Out[13]:
```
0.0390570392238516
```

q1.c ix)

In [14]:
```
S_beta = var_hat * np.linalg.inv(X.T @ X)
std_beta1 = np.sqrt(S_beta[0,0])
std_beta2 = np.sqrt(S_beta[1,1])

print(f'standard deviation for Beta_1 : {std_beta1:.4f}, for Beta_2 : {std_beta2:.4
print(S_beta)
```

```
standard deviation for Beta_1 : 0.0442, for Beta_2 : 0.0442
[[0.00195285 0.        ]
 [0.         0.00195285]]
```

q1.c x)

```
In [15]: # calculate the contrast vector lmbda and the reduced model X_0

lmbda = np.asarray([1, -1])
X_0 = X @ np.asarray([1, 1])
X_0 = X_0.reshape((-1,1))
```

q1.c xi)

```
In [16]: # calculate the error from the reduced model

Px_0 = X_0 @ np.linalg.inv(X_0.T @ X_0) @ X_0.T
d = Px_0.shape[0]
I = np.identity(d)
Rx_0 = (I - Px_0)
error_0_hat = Rx_0 @ Y

# SSR = sum(Y_mean - Y_hat)**2
# we have error_hat = Y - Y_hat -> so introduce Y_error = Y_mean - Y
v1 = np.trace(Px - Px_0)
v2 = np.trace(I - Px)
Y_error = Y.mean() - Y
SSR_X0 = np.square(Y_error + error_0_hat).sum()
SSR_X = np.square(Y_error + error_hat).sum()

F_numerator = (SSR_X0 - SSR_X) / v1
F_denominator = SSR_X / v2

F_statistic = F_numerator / F_denominator
print(f'F statistic comparing the reduced model to the full model: {F_statistic:.2f

V = lmbda @ beta
S_V = np.sqrt(lmbda.reshape((2,1)).T @ S_beta @ lmbda.reshape((2,1)))
t_df = np.squeeze(V/S_V)

print(f'the degrees of freedom of the F statistic is ({v1:.0f}, {v2:.0f})')
print(f'p-value = {1 - stats.f.cdf(-F_statistic, v1, v2)}')
```

```
F statistic comparing the reduced model to the full model: -38.00
the degrees of freedom of the F statistic is (1, 38)
p-value = 3.3873272231588203e-07
```

q1.c xii)

```
In [17]: # calculate the t-statistic

numerator = lmbda @ beta
denominator = np.sqrt(lmbda.reshape((1,-1)) @ S_beta @ lmbda.reshape((-1,1)))[0,0]
t_statistic = numerator / denominator
print(f't-statistic for different means: {t_statistic:.2f}')
print(f'difference between t-statistic calculated at the begining : {t_statistic -
```

```
t-statistic for different means: -7.04
difference between t-statistic calculated at the begining : 0.00000
```

q1.c xiv)

In [18]:
```
# calcualte error from ground truth (Y_gt)
Y_gt = np.ones(Y.shape[0])
Y_gt[:20] = 1.5
Y_gt[20:] = 2.0

error = Y_gt - Y
error_projected_CX = Px @ error
error_projected_CX
```

Out[18]:
```
array([0.00218045, 0.00218045, 0.00218045, 0.00218045, 0.00218045,
       0.00218045, 0.00218045, 0.00218045, 0.00218045, 0.00218045,
       0.00218045, 0.00218045, 0.00218045, 0.00218045, 0.00218045,
       0.00218045, 0.00218045, 0.00218045, 0.00218045, 0.00218045,
       0.06202232, 0.06202232, 0.06202232, 0.06202232, 0.06202232,
       0.06202232, 0.06202232, 0.06202232, 0.06202232, 0.06202232,
       0.06202232, 0.06202232, 0.06202232, 0.06202232, 0.06202232,
       0.06202232, 0.06202232, 0.06202232, 0.06202232, 0.06202232])
```

In [19]:
```
beta - np.array([1.5,2])
```

Out[19]:
```
array([-0.00218045, -0.06202232])
```

q1.c xv)

In [20]:
```
# error projected into not(C(x))

error_projected_not_CX = Rx @ error
error_projected_not_CX

text = 'error_projected_not_CX = ['
for e in error_projected_not_CX[:-1]:
    text += f'{e:.3f}, '
text += f'{error_hat[-1]:.3f}]'
print(text)
print(f'error_hat diff to this: {(error_hat + error_projected_not_CX).sum()}')
```

```
error_projected_not_CX = [-0.090, -0.263, -0.060, -0.001, 0.120, 0.129, 0.305, 0.1
40, 0.124, -0.106, -0.370, -0.058, -0.154, 0.019, 0.061, -0.324, 0.149, 0.210, 0.3
03, -0.135, -0.108, -0.169, 0.177, -0.323, -0.216, -0.007, -0.381, 0.279, -0.052,
0.009, 0.039, 0.240, -0.070, 0.158, 0.280, 0.188, -0.086, -0.189, -0.062, -0.292]
error_hat diff to this: -5.4088677980956845e-15
```

q1.d i)

In [21]:
```
X_intercept = np.zeros(shape=(Y.shape[0], 3))
X_intercept[:,   0] = 1
X_intercept[:20, 1] = 1
X_intercept[20:, 2] = 1

print(f'design matrix X has rank {np.linalg.matrix_rank(X_intercept)}')
```

```
design matrix X has rank 2
```

q1.d ii)

In [22]:
```
Z = X_intercept
Px_intercept = Z @ np.linalg.pinv(Z.T @ Z) @ Z.T
Px_intercept
```

Out[22]:
```
array([[ 5.0000000e-02,  5.0000000e-02,  5.0000000e-02, ...,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00],
       [ 5.0000000e-02,  5.0000000e-02,  5.0000000e-02, ...,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00],
       [ 5.0000000e-02,  5.0000000e-02,  5.0000000e-02, ...,
         0.0000000e+00,  0.0000000e+00,  0.0000000e+00],
       ...,
       [-6.9388939e-18, -6.9388939e-18, -6.9388939e-18, ...,
         5.0000000e-02,  5.0000000e-02,  5.0000000e-02],
       [-6.9388939e-18, -6.9388939e-18, -6.9388939e-18, ...,
         5.0000000e-02,  5.0000000e-02,  5.0000000e-02],
       [-6.9388939e-18, -6.9388939e-18, -6.9388939e-18, ...,
         5.0000000e-02,  5.0000000e-02,  5.0000000e-02]])
```

q1.d iii)

In [23]:
```
lmbda_intercept = np.asarray([0,1,-1]).reshape((1,-1))
mult = np.asarray([[1, 0],
                   [0, 1],
                   [0, 1]])
X_0_intercept = X_intercept @ mult
X_0_intercept.shape
```

Out[23]:
```
(40, 2)
```

q1.d iv)

In [24]:
```
# calculate the t-statistic

d = Y.shape[0]
I = np.identity(d)
Rx_intercept = (I - Px_intercept)
error_hat_intercept = Rx_intercept @ Y
M = np.linalg.pinv(X_intercept.T @ X_intercept)
beta_intercept = M @ X_intercept.T @ Y

numerator = (np.dot(error_hat_intercept, error_hat_intercept))
denominator = d - np.linalg.matrix_rank(X_intercept)
var_hat_intercept = numerator / denominator

S_beta_intercept = var_hat_intercept * M

numerator = lmbda_intercept @ beta_intercept
denominator = np.sqrt(lmbda_intercept @ S_beta_intercept @ lmbda_intercept.T)
t_statistic_intercept = (numerator / denominator)[0,0]
t_statistic_intercept
```

Out[24]:
```
-7.043022482674422
```

In [25]:
```
beta_intercept
```

Out[25]:
```
array([1.14526574, 0.3525538 , 0.79271194])
```

q1.e i)

In [26]:
```
X_e= np.zeros(shape=(Y.shape[0], 2))
X_e[:,    0] = 1
X_e[:20, 1] = 1

print(f'design matrix X has rank {np.linalg.matrix_rank(X_e)}')
```

```
design matrix X has rank 2
```

q1.e ii)

In [27]:
```
lmbda_e = np.asarray([0, 1]).reshape((2, -1))
```

q1.e iii)

In [28]:
```
# calculate the t-statistic
Z = X_e
Px_e = Z @ np.linalg.pinv(Z.T @ Z) @ Z.T
Px_e

d = Y.shape[0]
I = np.identity(d)
Rx_e = (I - Px_e)
error_hat_e = Rx_e @ Y
M = np.linalg.pinv(X_e.T @ X_e)
beta_e = M @ X_e.T @ Y

numerator = (np.dot(error_hat_e, error_hat_e))
denominator = d - np.linalg.matrix_rank(X_e)
var_hat_e = numerator / denominator

S_beta_e = var_hat_e * M

numerator = np.dot(lmbda_e.flatten(), beta_e)
denominator = np.sqrt(lmbda_e.T @ S_beta_e @ lmbda_e)
t_statistic_e = (numerator / denominator)[0,0]
print(t_statistic_e)
```

```
-7.043022482674408
```

q2.a i)

In [29]:
```
# now computing the ttest for null hypothesis 2 samples come from the same
# distribution with the same mean
t_statistic_1sample, p_val_1sample = stats.ttest_rel(group1, group2)
print(f'ttest for 1 sample distribution: t = {t_statistic_1sample:.2f}, p-value = 
print(f'ttest for 2 sample distribution: t = {t_statistic:.2f}, p-value = {p_val:.
```

```
ttest for 1 sample distribution: t = -6.13, p-value = 0.00000678
ttest for 2 sample distribution: t = -7.04, p-value = 0.00000002
```

q2.b i)

In [30]:
```
# create the design matrix
X_2b = np.zeros(shape=(40,22))
X_2b[:,   0] = 1
X_2b[20:, 1] = 1
for i in range(20):
    X_2b[i,    i+2] = 1
    X_2b[20+i, i+2] = 1

print(f'rank of X is: {np.linalg.matrix_rank(X_2b)}')
```

```
rank of X is: 21
```

In [31]:
```
# create lmbda
lmbda_2b = np.zeros(22)
lmbda_2b[1] = -1
lmbda_2b = lmbda_2b.reshape((22,1))
```

q2.b iii)

In [32]:
```
# calculate the t-statistic
Z = X_2b
Px_2b = Z @ np.linalg.pinv(Z.T @ Z) @ Z.T
Px_2b

d = Y.shape[0]
I = np.identity(d)
Rx_2b = (I - Px_2b)
error_hat_2b = Rx_2b @ Y
M = np.linalg.pinv(X_2b.T @ X_2b)
beta_2b = M @ X_2b.T @ Y

numerator = (np.dot(error_hat_2b, error_hat_2b))
denominator = d - np.linalg.matrix_rank(X_2b)
var_hat_2b = numerator / denominator

S_beta_2b = var_hat_2b * M

numerator = np.dot(lmbda_2b.flatten(), beta_2b)
denominator = np.sqrt(lmbda_2b.T @ S_beta_2b @ lmbda_2b)
t_statistic_2b = (numerator / denominator)[0,0]
t_statistic_2b
```

Out[32]:
```
-6.13254658116648
```

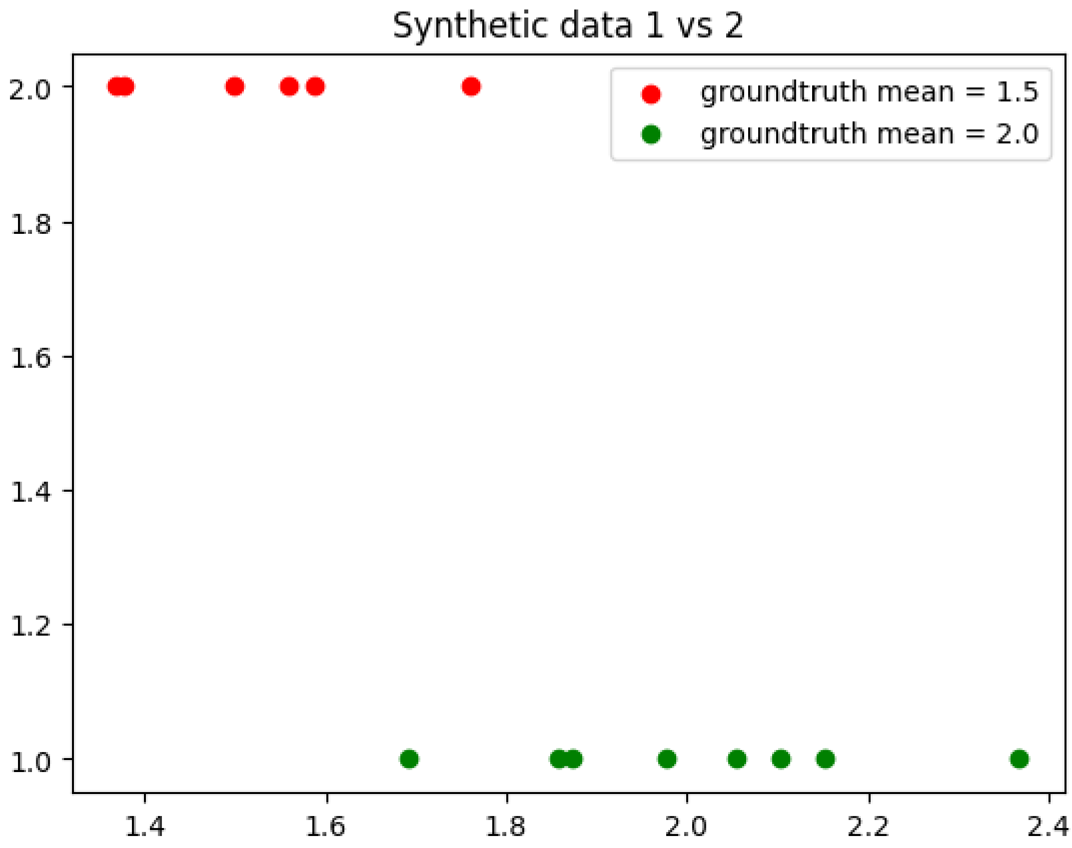
Part 2 - Q1

In [1]:
```
import numpy as np
import scipy.stats as stats
import scipy.special as special
import matplotlib.pyplot as plt
from itertools import combinations
import random
from statsmodels.distributions.empirical_distribution import ECDF

rng = np.random.default_rng(seed=22197823)
random.seed(22197823)
```

q1.a)

In [2]:
```
# generate sample data
std_gt = 0.2
mean_1_gt = 1.5
mean_2_gt = 2.0
N_1 = 6
N_2 = 8

group1 = rng.normal(loc=mean_1_gt, scale=std_gt, size=N_1)
group2 = rng.normal(loc=mean_2_gt, scale=std_gt, size=N_2)

mean_1_calc = group1.mean()
mean_2_calc = group2.mean()
std_1_calc = group1.std()
std_2_calc = group2.std()

print(f'Data 1: mean : {mean_1_calc:.02f}, std: {std_1_calc:.02f}. Expected mean ar
print(f'Data 2: mean : {mean_2_calc:.02f}, std: {std_2_calc:.02f}. Expected mean ar

plt.scatter(group1, 2 * np.ones(N_1), color='r', label=f'groundtruth mean = {mean_1
plt.scatter(group2, 1 * np.ones(N_2), color='g', label=f'groundtruth mean = {mean_2
plt.legend()
plt.title('Synthetic data 1 vs 2')

t_statistic_gt, p_val = stats.ttest_ind(group1, group2)
print(f'T statistic: {t_statistic_gt:.2f}, p-value: {p_val:.5f}')
```

```
Data 1: mean : 1.53, std: 0.13. Expected mean and std: 1.5, 0.2
Data 2: mean : 2.01, std: 0.19. Expected mean and std: 2.0, 0.2
T statistic: -4.86, p-value: 0.00039
```



q1.b i)

In [3]:

```
# put group 1 and 2 into 1D array D
D = np.hstack((group1, group2))
```

q1.b ii)

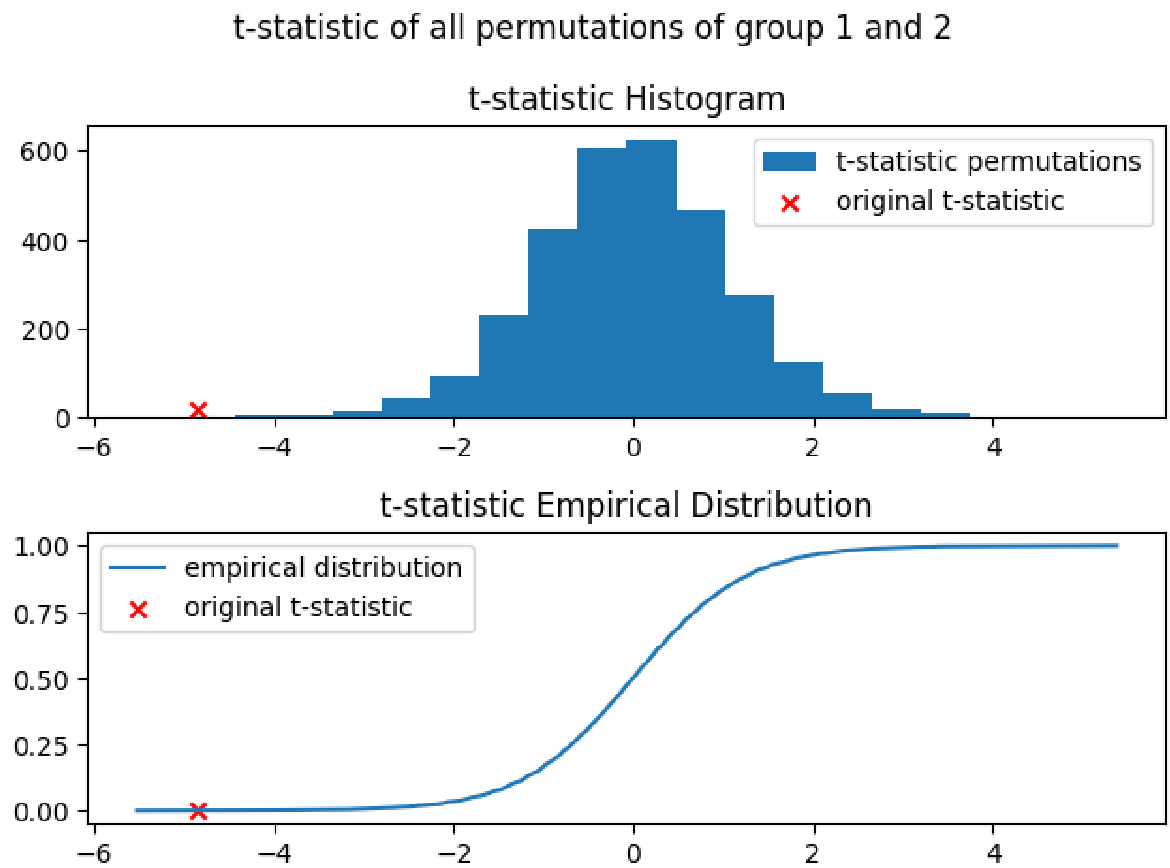
In [4]:

```
# Find all permutations of group 1 and group 2

N = int(special.comb(N_1 + N_2, N_1))
D_group_perms = np.zeros((N, N_1 + N_2))
for row, group1_perm in enumerate(combinations(D, N_1)):
    # for the group 1 permutation, store it as group 1 for this row in D_group_perm
    group1_perm = np.asarray(group1_perm)
    D_group_perms[row, :N_1] = group1_perm

    # for any item not in the group 1 permutation then add it to group 2
    group2_idx = N_1
    for item in D:
        if item not in group1_perm:
            D_group_perms[row, group2_idx] = item
            group2_idx += 1
print(f'total number of permutations is {N}')
print(f'sum of D_group_perm should be {N} * D.sum : {D.sum() * N - D_group_perms.su
```

```
total number of permutations is 3003
sum of D_group_perm should be 3003 * D.sum : 0.00
```

q1.b iii)

In [5]:
```
# compute the t-statistic for all group members
# Assuming group 1 and group 2 are independent samples

N = D_group_perms.shape[0]
t_statistic_perms = np.zeros(N)
for row in range(N):
    group1_perm = D_group_perms[row, :N_1]
    group2_perm = D_group_perms[row, N_1:]
    ttest_perm, _ = stats.ttest_ind(group1_perm, group2_perm)
    t_statistic_perms[row] = ttest_perm

fig, axs = plt.subplots(nrows=2, ncols=1)
fig.suptitle('t-statistic of all permutations of group 1 and 2')

axs[0].hist(t_statistic_perms, bins=20, label='t-statistic permutations')
axs[0].scatter(x=t_statistic_gt, y=20, marker='x', color='r', label='original t-sta
axs[0].set_title('t-statistic Histogram')
axs[0].legend()

t_ecdf = ECDF(t_statistic_perms)
axs[1].plot(np.sort(t_statistic_perms), t_ecdf(np.sort(t_statistic_perms)), label=
axs[1].scatter(t_statistic_gt, t_ecdf(t_statistic_gt), marker='x', color='r', label
axs[1].set_title('t-statistic Empirical Distribution')
axs[1].legend()

fig.tight_layout()
```



q1.b iv)

In [6]:
```
# to find p-val find original t-statistic and find number of t-statistics with equa
# Note we are doing a 2-tailed test, so due to the symmetry of the ecdf we multiply
```

```
t_stat_num_equal_or_greater = 2* (t_statistic_perms <= t_statistic_gt).sum()
p_val_perms = (t_stat_num_equal_or_greater / N)

print(f'original t-statistic = {t_statistic_gt:.3f}, empirical p-val = {p_val_perms
print(f'number of t-statistic empirical values equal or more extreme than original:
print(f'original calculated p-val: {p_val:.5f}')
```

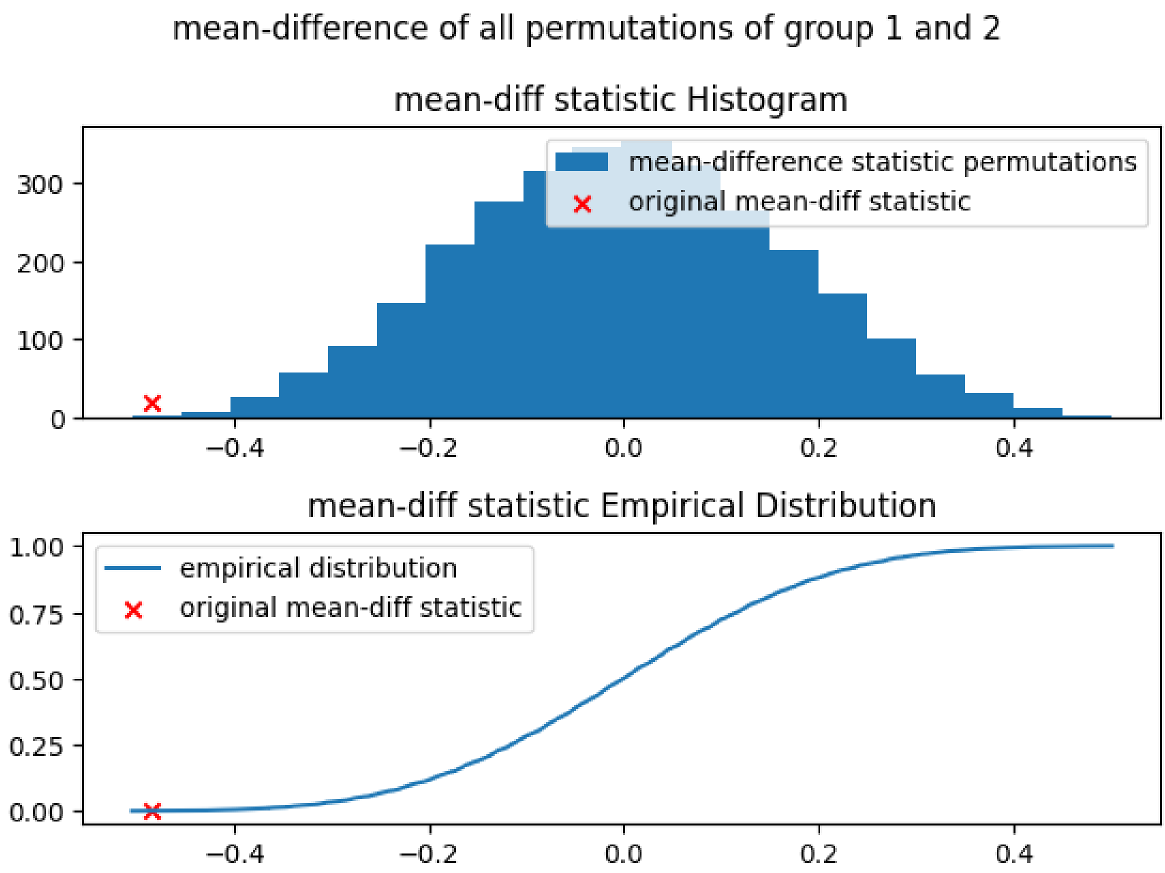
```
original t-statistic = -4.859, empirical p-val = 0.00133
number of t-statistic empirical values equal or more extreme than original: 4
original calculated p-val: 0.00039
```

q1.c)

In [7]:

```
# caclulate using the means as the statistic
mean_statistic_gt = group1.mean() - group2.mean()

N = D_group_perms.shape[0]
mean_statistic_perms = np.zeros(N)
for row in range(N):
    group1_perm = D_group_perms[row, :N_1]
    group2_perm = D_group_perms[row, N_1:]
    mean_statistic_perms[row] = group1_perm.mean() - group2_perm.mean()

# to find p-val find original t-statistic and find number of t-statistics with equa
# Note we are doing a 2-tailed test, so due to the symmetry of the ecdf we multiply
mean_stat_num_equal_or_greater = 2*(mean_statistic_perms <= mean_statistic_gt).sum(
p_val_mean_perms = mean_stat_num_equal_or_greater / N
print(f'original mean-diff statistic = {mean_statistic_gt:.2f}, empirical p-val = {
print(f'number of mean-diff statistic empirical values equal or more extreme than o

fig, axs = plt.subplots(nrows=2, ncols=1)
fig.suptitle('mean-difference of all permutations of group 1 and 2')

axs[0].hist(mean_statistic_perms, bins=20, label='mean-difference statistic permuta
axs[0].scatter(x=mean_statistic_gt, y=20, marker='x', color='r', label='original me
axs[0].set_title('mean-diff statistic Histogram')
axs[0].legend()

mean_ecdf = ECDF(mean_statistic_perms)
axs[1].plot(np.sort(mean_statistic_perms), mean_ecdf(np.sort(mean_statistic_perms))
axs[1].scatter(mean_statistic_gt, mean_ecdf(mean_statistic_gt), marker='x', color=
axs[1].set_title('mean-diff statistic Empirical Distribution')
axs[1].legend()

fig.tight_layout()
```

```
original mean-diff statistic = -0.48, empirical p-val = 0.00133
number of mean-diff statistic empirical values equal or more extreme than origina
l: 4
```



q1.d i)

In [8]:

```
# calculate 1000 random permutations of group1 and group2 selection
# Note the original group1 and group2 musrt be in the final sample

M = 1000
# we will create a set of all unique selections of group 1
# then we will create an ndarray of all samples including the missing group 2 items
group1_unique_perms = set()
# set is initialised with group 1
group1_unique_perms.add(tuple(group1))
D_tuple = tuple(D)

# use a while loop because sampled perms are not always going to be unique so will
# have to go through the loop probably more than M times
# Note: sets only allow unique items, so perm is only added if unique
while len(group1_unique_perms) < M:
    group1_perm = tuple(sorted(random.sample(D_tuple, N_1)))
    group1_unique_perms.add(group1_perm)

# now add group 2 to the group 1 selections
D_group_perms_1000 = np.zeros(shape=(M, N_1 + N_2))
for row, group1_tuple_perm in enumerate(group1_unique_perms):
    # store group 1 in ndarray
    D_group_perms_1000[row, :N_1] = np.asarray(group1_tuple_perm)
    # store the remaining items as group 2 in ndarray
    i = N_1
    for item in D:
        if item not in group1_tuple_perm:
            D_group_perms_1000[row, i] = item
            i += 1
```

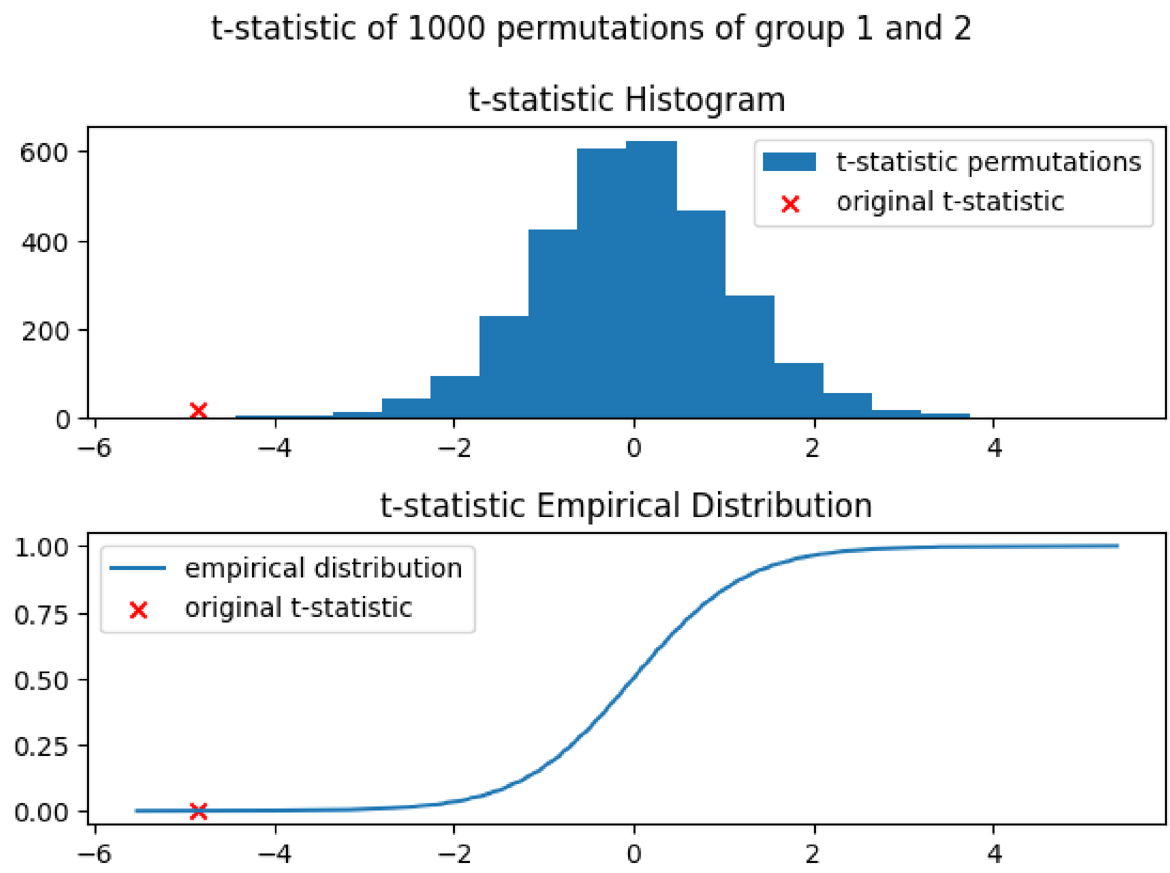
```
print(f'check permutation array sum is {M} * D.sum(), diff is : {M * D.sum() - D_gr
```

```
check permutation array sum is 1000 * D.sum(), diff is : 0.0
```

q1.d ii)

In [9]:
```
# compute the t-statistic for all group members using the 1000 sampled permutations
# Assuming group 1 and group 2 are independent samples

N = D_group_perms_1000.shape[0]
t_statistic_perms_1000 = np.zeros(N)
for row in range(N):
    group1_perm = D_group_perms_1000[row, :N_1]
    group2_perm = D_group_perms_1000[row, N_1:]
    ttest_perm, _ = stats.ttest_ind(group1_perm, group2_perm)
    t_statistic_perms_1000[row] = ttest_perm

fig, axs = plt.subplots(nrows=2, ncols=1)
fig.suptitle('t-statistic of 1000 permutations of group 1 and 2')

axs[0].hist(t_statistic_perms, bins=20, label='t-statistic permutations')
axs[0].scatter(x=t_statistic_gt, y=20, marker='x', color='r', label='original t-sta
axs[0].set_title('t-statistic Histogram')
axs[0].legend()

t_ecdf = ECDF(t_statistic_perms)
axs[1].plot(np.sort(t_statistic_perms), t_ecdf(np.sort(t_statistic_perms)), label=
axs[1].scatter(t_statistic_gt, t_ecdf(t_statistic_gt), marker='x', color='r', label
axs[1].set_title('t-statistic Empirical Distribution')
axs[1].legend()

fig.tight_layout()

# find p-val of 1000 perms vs original p-val and all perms p-val
t_stat_1000_num_equal_or_greater = 2 * (t_statistic_perms_1000 <= t_statistic_gt).s
p_val_perms_1000 = t_stat_1000_num_equal_or_greater / N
print(f'original t-statistic = {t_statistic_gt:.2f}, empirical p-val = {p_val_perms
print(f'number of t-statistic empirical values equal or more extreme than original:
print(f'\np-val comparison:\n{p_val:.5f} : original p-val\n{p_val_perms:.5f} : all
```

```
original t-statistic = -4.86, empirical p-val = 0.00400
number of t-statistic empirical values equal or more extreme than original: 4

p-val comparison:
0.00039 : original p-val
0.00133 : all permutations p-val
0.00400 : 1000 permutations p-val
```



q1.d iii)

In [10]:

```
# check there are no duplicate permutations
# To check this we will create an array of the same size as D_group_perms_1000, and
# sort all group 1 items and sort all group 2 items. This way any duplicates that h
# sort order will be flagged as duplicate.
# once group 1 and group 2 are sorted we then create a unique array of permutations
# to calculate the number of unique permutations

# create array of sorted group 1 and sorted group 2 permutations
D_group_perms_1000_sorted = np.zeros(shape=D_group_perms_1000.shape)
D_group_perms_1000_sorted[:,:N_1] = np.sort(D_group_perms_1000[:,:N_1], axis=1)
D_group_perms_1000_sorted[:,N_1:] = np.sort(D_group_perms_1000[:,N_1:], axis=1)

# only keep the unique permutations
D_group_perms_1000_sorted_unique = np.unique(D_group_perms_1000_sorted, axis=0)

# calculate the number of duplicates
print(f'number of duplicated permutations: {M - D_group_perms_1000_sorted_unique.sh
```

```
number of duplicated permutations: 0
```

Part 2 - Q2

q2.a)

In [1]:
```
import numpy as np
import scipy.stats as stats
import scipy.special as special
import matplotlib.pyplot as plt
from itertools import combinations
import random
import struct
from statsmodels.distributions.empirical_distribution import ECDF

rng = np.random.default_rng(seed=22197823)
random.seed(22197823)
```

In [2]:
```
def load_files(filenames):
    n = len(filenames)
    data = np.zeros(shape=(n, 40, 40, 40))
    for im_num, filename in enumerate(filenames):
        with open(filename, 'rb') as f:
            # Read the binary data in little-endian format
            file_data = f.read()
            # Convert the binary data to a list of little-endian floats
            floats = list(struct.unpack('<' + 'f' * (len(file_data) // 4), file_da
        im = np.array(floats).reshape((40,40,40))
        data[im_num] = im

    return data
```

In [3]:
```
# load all files
filename_CPA_nums = ['04', '05', '06', '07', '08', '09', '10', '11']
filename_PPA_nums = ['03', '06', '09', '10', '13', '14', '15', '16']
filenames_CPA = []
filenames_PPA = []
filenames_mask = ['data/wm_mask.img']
for CPA_num in filename_CPA_nums:
    filenames_CPA.append('data/CPA' + CPA_num + '_diffeo_fa.img')
for PPA_num in filename_PPA_nums:
    filenames_PPA.append('data/PPA' + PPA_num + '_diffeo_fa.img')

data_CPA = load_files(filenames_CPA)
data_PPA = load_files(filenames_PPA)
data_mask = load_files(filenames_mask)[0,:,:,:]
```

In [4]:
```
# change voxel data into shape nxm where n is number of samples, and m is number o

# flatten data mask to use as an index on CPA and PPA data. data_mask_flatten is a
n = data_CPA.shape[0]
data_mask_flatten = data_mask.reshape(-1)

# flatten CPA and PPA data. They are now of shape nxM where M is number of all vox
data_CPA_flatten = data_CPA.reshape((n, -1))
data_PPA_flatten = data_PPA.reshape((n, -1))

# only select the voxels of interest to get an array of shape nxm
data_CPA_roi = data_CPA_flatten[:, data_mask_flatten > 0]
```

```python
data_PPA_roi = data_PPA_flatten[:, data_mask_flatten > 0]

# check that we have selected the correct voxels. Check this by adding all relevant
# original data array and compare with summing all selected voxels
original_direct_sum = np.einsum('ijkl,jkl->i', data_CPA, data_mask).sum() + np.ein
converted_sum = data_CPA_roi.sum() + data_PPA_roi.sum()
print(f'check the correct roi voxels have been picked. Sum should be zero : {origi
```

```
check the correct roi voxels have been picked. Sum should be zero : 0.0
```

In [5]:

```python
def calculate_t_statistic(Y, X, lmbda, is_max_only=False, Px=None, M=None):
    """given a Y of many voxels, and X design matrix and contrast vector lmbda retu
    the t_statistic for all voxels

    Args:
        Y (ndarray - shape [nxv]): n is number of observations, v is number of voxe
        X (ndarray - shape [nx2]): design matrix
        lmbda (ndarray - shape [2x1]): contrast vector

    return:
        t_statistic (ndarray - shape [v]): t_statistic for each voxel
    """
    if Px is None:
        Px = X @ np.linalg.pinv(X.T @ X) @ X.T

    d = Px.shape[0]
    I = np.identity(d)
    Rx = (I - Px)
    error_hat = Rx @ Y

    if M is None:
        M = np.linalg.pinv(X.T @ X)

    beta = M @ X.T @ Y

    numerator = np.einsum('ji,ij->j', error_hat.T, error_hat)
    denominator = d - np.linalg.matrix_rank(X)
    var_hat = numerator / denominator

    S_beta = np.einsum('i,jk->jki', var_hat, M)

    numerator_all = (lmbda.T @ beta).flatten()
    L = np.einsum('ij,jkl->ikl', lmbda.T, S_beta)
    denominator_all = np.sqrt(np.einsum('ijk,jl->k', L, lmbda))

    t_statistic = (numerator_all / denominator_all)

    if is_max_only:
        return t_statistic.max()
    else:
        return t_statistic
```

In [6]:

```python
N_1 = 8
N_2 = 8
# put it in the form of GLM
Y = np.vstack((data_CPA_roi, data_PPA_roi))

# create design matrix. X[i] is design matrix for voxel i
X = np.zeros(shape=(Y.shape[0], 2))
X[:N_1, 0] = 1
X[N_1:, 1] = 1

print(f'rank of X is: {np.linalg.matrix_rank(X)}')
```

```
rank of X is: 2
```

In [7]:
```
# create contrast vector lmbda
lmbda = np.array([1,-1])
lmbda = lmbda.reshape((2,1))
lmbda

# calculate the t-statistic
t_statistic_roi = calculate_t_statistic(Y, X, lmbda)

print(f'Max t_statistic on all voxels: {t_statistic_roi.max():.2f}')
```

```
Max t_statistic on all voxels: 6.53
```

q2.b)

In [8]:
```
# Find all permutations of the indexs of group 1 and group 2, with each group being
def find_all_group_perm_idxs(N_1, N_2):
    N = int(special.comb(N_1 + N_2, N_1))
    group_idxs = np.arange(N_1 + N_2)
    group_perms_idxs = np.zeros((N, N_1 + N_2))

    for row, group1_perm in enumerate(combinations(group_idxs, N_1)):
        # for the group 1 permutation, store it as group 1 for this row in D_group_
        group1_perm = np.asarray(group1_perm)
        group_perms_idxs[row, :N_1] = group1_perm

        # for any item not in the group 1 permutation then add it to group 2
        group2_idx = N_1
        for item in group_idxs:
            if item not in group1_perm:
                group_perms_idxs[row, group2_idx] = item
                group2_idx += 1
    return np.int32(group_perms_idxs)
```

In [9]:
```
group_perm_idxs = find_all_group_perm_idxs(N_1, N_2)
group_perm_idxs.shape
```

Out[9]:
```
(12870, 16)
```

In [10]:
```
total_perms_num = group_perm_idxs.shape[0]
t_statistic_perms_max = np.zeros(total_perms_num)

# pre calculate the inverse matrixes which don't change with each permutation
M = np.linalg.pinv(X.T @ X)
Px = X @ M @ X.T

for row, perm in enumerate(group_perm_idxs):
    Y_perm = Y[perm, :]
    t_statistic_perms_max[row] = calculate_t_statistic(Y_perm, X, lmbda, is_max_onl
    if row % 1000 == 0:
        print(f'done {((row / total_perms_num)*100):.0f}%')
print('done 100%')
```

```
done 0%
done 8%
done 16%
done 23%
done 31%
done 39%
done 47%
done 54%
done 62%
done 70%
done 78%
done 85%
done 93%
done 100%
```

q2.c)

In [11]:
```
t_statistic_max_gt = t_statistic_roi.max()
num_t_stat_more_extreme = (t_statistic_perms_max >= t_statistic_max_gt).sum()
total_t_stat_num = t_statistic_perms_max.shape[0]

p_val = num_t_stat_more_extreme / total_t_stat_num

print(f'Total number of max t-statistics : {total_t_stat_num}')
print(f'Number of max t-stats equally or more extreme than original labeling : {nu
print(f'This gives a p-val of : {p_val:.4f}')
```

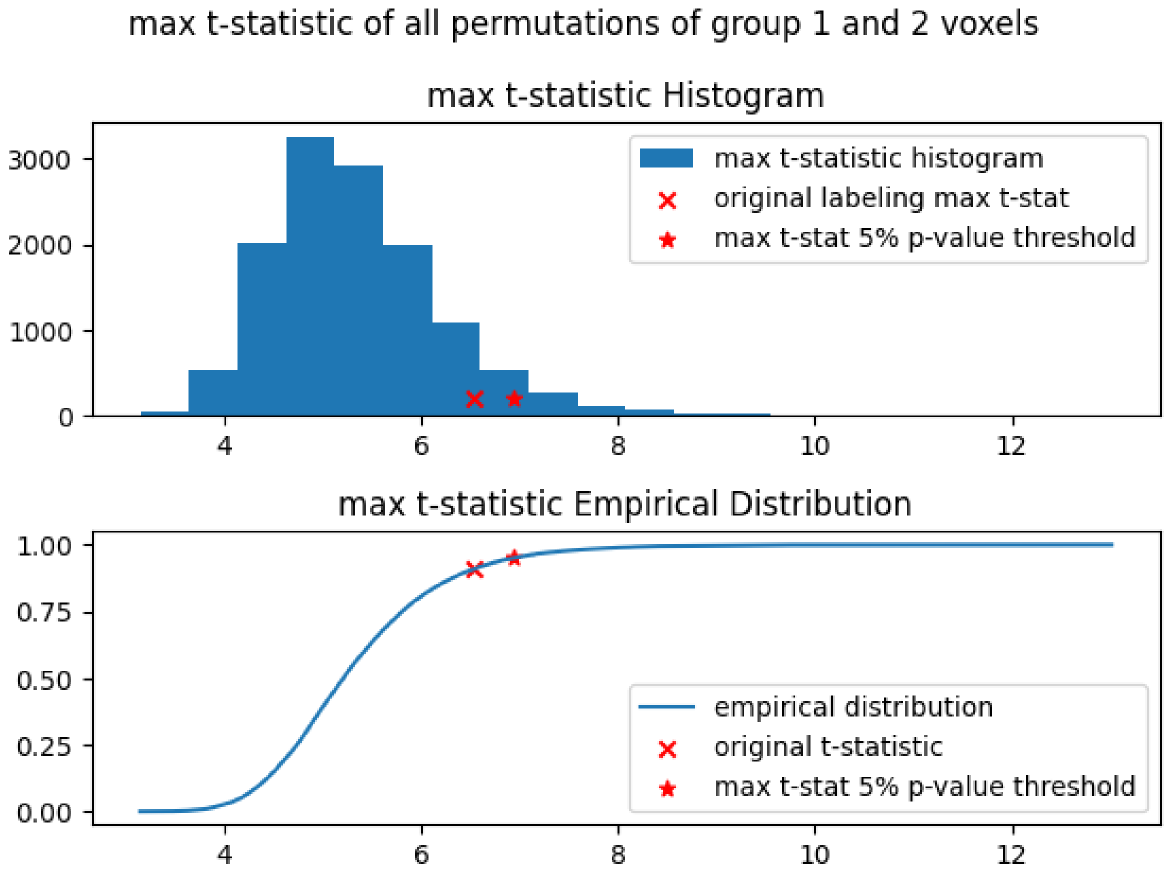
```
Total number of max t-statistics : 12870
Number of max t-stats equally or more extreme than original labeling : 1182
This gives a p-val of : 0.0918
```

q2.d)

In [12]:
```
# find the 95-th percentile max t-stat to find the threshold for
# 5% p-val
sorted_max_t_stat = np.sort(t_statistic_perms_max)
thresh_idx = int(0.95 * total_t_stat_num)
t_stat_thresh_5pct = sorted_max_t_stat[thresh_idx]
print(f'Threshold for maximum t-stat for a corresponding 5% p-value : {t_stat_thres
```

```
Threshold for maximum t-stat for a corresponding 5% p-value : 6.93826
```

In [13]:
```
fig, axs = plt.subplots(nrows=2, ncols=1)
fig.suptitle('max t-statistic of all permutations of group 1 and 2 voxels')

axs[0].hist(t_statistic_perms_max, bins=20, label='max t-statistic histogram')
axs[0].scatter(x=t_statistic_max_gt, y=200, color='r', marker='x', label='original
axs[0].scatter(x=t_stat_thresh_5pct, y=200, color='r', marker='*', label='max t-st
axs[0].set_title('max t-statistic Histogram')
axs[0].legend()

t_max_ecdf = ECDF(t_statistic_perms_max)
axs[1].plot(np.sort(t_statistic_perms_max), t_max_ecdf(np.sort(t_statistic_perms_m
axs[1].scatter(t_statistic_max_gt, t_max_ecdf(t_statistic_max_gt), marker='x', col
axs[1].scatter(x=t_stat_thresh_5pct, y=t_max_ecdf(t_stat_thresh_5pct), color='r', 
axs[1].set_title('max t-statistic Empirical Distribution')
axs[1].legend()

fig.tight_layout()
```


max t-statistic of all permutations of group 1 and 2 voxels
max t-statistic Histogram
max t-statistic histogram
original labeling max t-stat
max t-stat 5% p-value threshold
3000
2000
1000
0
4
6
8
10
12
max t-statistic Empirical Distribution
empirical distribution
original t-statistic
max t-stat 5% p-value threshold
1.00
0.75
0.50
0.25
0.00