index

January 17, 2022

1 Resampling Methods - Lab

1.1 Introduction

Now that you have some preliminary background on bootstrapping, jackknife, and permutation tests, its time to practice those skills by coding them into functions. You'll then apply these tests to a hypothesis test and compare the results to a parametric t-test.

1.2 Objectives

In this lab you will:

- Create functions that perform resampling techniques and use them on datasets

1.3 Bootstrap sampling

Bootstrap sampling works by combining two distinct samples into a universal set and generating random samples from this combined sample space in order to compare these random splits to the two original samples. The idea is to see if the difference between the two **original** samples is statistically significant. If similar differences can be observed through the random generation of samples, then the observed differences are not actually significant.

Write a function to perform bootstrap sampling. The function should take in two samples A and B. The two samples need not be the same size. From this, create a universal sample by combining A and B. Then, create a resampled universal sample of the same size using random sampling with replacement. Finally, split this randomly generated universal set into two samples which are the same size as the original samples, A and B. The function should return these resampled samples.

Example:

```
A = [1,2,3]
B = [2,2,5,6]

Universal_Set = [1,2,2,2,3,5,6]
Resampled_Universal_Set = [6, 2, 3, 2, 1, 1, 2] # Could be different (randomly generated with

Resampled_A = [6,2,3]
Resampled_B = [2,1,1,2]
```

[1]:
```
import numpy as np
```

[20]:
```
def bootstrap(A, B):
    C = list(A) + list(B)
    c_prime = np.random.choice(C,len(C), replace = True)
    A_new = c_prime[:len(A)]
    B_new = c_prime[len(A):]
#     A_new = np.random.choice(C,len(A), replace = True)
#     B_new = np.random.choice(C,len(B), replace = True)
    return A_new, B_new
    # Your code here


#### From GitHub Solution:

# def bootstrap(A, B):
#     universe = list(A) + list(B)
#     universe_shuffled = np.random.choice(universe, size=len(universe),␣
 ↪replace=True)
#     new_a = universe_shuffled[:len(A)]
#     new_b = universe_shuffled[len(A):]
#     return new_a, new_b
```

1.4 Jackknife

Write a function that creates additional samples by removing one element at a time. The function should do this for each of the `n` items in the original sample, returning `n` samples, each with `n-1` members.

[2]:
```
def jack1(A):
    """This function should take in a list of n observations and return n lists
    each with one member (presumably the nth) removed."""
    # Your code here
    A_new = []
    for i in range(len(A)):
        new_elements = A[:i] + A[i+1:]
        A_new.append(new_elements)
    return A_new
```

1.5 Permutation testing

Define a function that generates all possible, equally sized, two set splits of two sets A and B. Sets A and B need not be the same size, but all of the generated two set splits should be of equal size. For example, if we had a set with 5 members and a set with 7 members, the function would return all possible 5-7 ordered splits of the 12 items.

> Note that these are actually combinations! However, as noted previously, permutation tests really investigate possible regroupings of the data observations, so calculating combinations is a more efficient approach!

Here's a more in depth example:

```
A = [1, 2, 2]
B = [1, 3]
combT(A, B)
[([1,2,2], [1,3]),
 ([1,2,3], [1,2]),
 ([1,2,1], [2,3]),
 ([1,1,3], [2,2]),
 ([2,2,3], [1,1])]
```

These are all the possible 3-2 member splits of the 5 elements: 1, 1, 2, 2, 3.

[3]:
```
#### From GitHub Solution

from itertools import combinations

def combT(a, b):
    union = sorted(a + b)
    all_combs = []
    for x in set(combinations(union, len(a))):
        union_copy = union.copy()
        for y in x:
            union_copy.remove(y)
        all_combs.append((list(x), list(combinations(union_copy, len(union) -␣
 ↪len(a)))))
    return all_combs
```

[ ]:
```
def combT(a,b):
    # Your code here
```

1.6 Permutation testing in Practice

Let's further investigate the scenario proposed in the previous lesson. Below are two samples A and B. The samples are mock data for the blood pressure of sample patients. The research study is looking to validate whether there is a statistical difference in the blood pressure of these two groups using a 5% significance level. First, calculate the mean blood pressure of each of the two samples. Then, calculate the difference of these means. From there, use your `combT()` function, defined above, to generate all the possible combinations of the entire sample data into A-B splits of equivalent sizes as the original sets. For each of these combinations, calculate the mean blood pressure of the two groups and record the difference between these sample means. The full collection of the difference in means between these generated samples will serve as the denominator to calculate the p-value associated with the difference between the original sample means.

For example, in our small handwritten example above:

$\mu_a = \frac{1+2+2}{3} = \frac{5}{3}$
and
$\mu_b = \frac{1+3}{2} = \frac{4}{2} = 2$

Giving us

$\mu_a - \mu_b = \frac{5}{3} - 2 = \frac{1}{2}$

In comparison, for our various combinations we have:

([1,2,2], [1,3]): $\mu_a - \mu_b = \frac{5}{3} - 2 = \frac{1}{2}$
([1,2,3], [1,2]): $\mu_a - \mu_b = 2 - \frac{3}{2} = \frac{1}{2}$
([1,2,1], [2,3]): $\mu_a - \mu_b = \frac{4}{3} - \frac{5}{3} = -\frac{1}{2}$
([1,1,3], [2,2]): $\mu_a - \mu_b = \frac{5}{3} - 2 = \frac{1}{2}$
([2,2,3], [1,1]): $\mu_a - \mu_b = \frac{7}{3} - 1 = \frac{4}{3}$

A standard hypothesis test for this scenario might be:

$H_0 : \mu_a = \mu_b$
$H_1 : \mu_a < \mu_b$

Thus comparing our sample difference to the differences of our possible combinations, we look at the number of experiments from our combinations space that were the same or greater than our sample statistic, divided by the total number of combinations. In this case, 4 out of 5 of the combination cases produced the same or greater differences in the two sample means. This value .8 is a strong indication that we cannot refute the null hypothesis for this instance.

[5]:
```
a = [109.6927759 , 120.27296943, 103.54012038, 114.16555857,
       122.93336175, 110.9271756 , 114.77443758, 116.34159338,
       112.66413025, 118.30562665, 132.31196515, 117.99000948]
b = [123.98967482, 141.11969004, 117.00293412, 121.6419775 ,
       123.2703033 , 123.76944385, 105.95249634, 114.87114479,
       130.6878082 , 140.60768727, 121.95433026, 123.11996767,
       129.93260914, 121.01049611]
```

[ ]:
```
### From GitHub Solution

# # Your code here
# from scipy.special import comb
# n_items = len(a)+len(b)
# unique = len(set(a+b))
# sample_size = len(a)

# print(n_items, sample_size)
# print(comb(n_items, sample_size))



# # Your code here
# diff_mu_a_b = np.mean(a) - np.mean(b)
# combos = combT(a, b)
# print("There are {} possible sample variations.".format(len(combos)))
# num = 0 # Initialize numerator
# for ai, bi in combos:
#     diff_mu_ai_bi = np.mean(ai) - np.mean(bi)
#     if diff_mu_ai_bi >= diff_mu_a_b:
```

```
#         num +=1
# p_val = num / len(combos)
# print('P-value: {}'.format(p_val))
```

[6]:
```
# Your code here
#   Expect your code to take several minutes to run

diff_means_total = np.mean(a) - np.mean(b)
num = 0
for first, second in combT(a, b):
    diff_mean_combos = np.mean(first) - np.mean(second)
    if diff_mean_combos >= diff_means_total:
        num += 1
p_value = num / len(combT(a, b))
print(f"P-Value is {p_value}")
```

```
P-Value is 0.9890762811021258
```

1.7 T-test revisited

The parametric statistical test equivalent to our permutation test above would be a t-test of the two groups. Perform a t-test on the same data above in order to calculate the p-value. How does this compare to the above results?

[11]:
```
# Your code here
import scipy.stats as stats

results = stats.ttest_ind(a,b, equal_var=False)
print(results.pvalue)
```

```
0.020794009741792126
```

[14]:
```
### From GitHub Solution

num = np.mean(a) - np.mean(b)
s = np.var(a+b)
n = len(a+b)
denom = s/np.sqrt(n)
t = num / denom
pval = stats.t.sf(np.abs(t), n-1)*2
print(pval)
```

```
0.6196331755824978
```

1.8 Bootstrap applied

Use your code above to apply the bootstrap technique to this hypothesis testing scenario. Here's a pseudo-code outline for how to do this:

1. Compute the difference between the sample means of A and B
2. Initialize a counter for the number of times the difference of the means of resampled samples is greater then or equal to the difference of the means of the original samples
3. Repeat the following process 10,000 times:
    1. Use the bootstrap sampling function you used above to create new resampled versions of A and B
    2. Compute the difference between the means of these resampled samples
    3. If the difference between the means of the resampled samples is greater then or equal to the original difference, add 1 the counter you created in step 2
4. Compute the ratio between the counter and the number of simulations (10,000) that you performed > This ratio is the percentage of simulations in which the difference of sample means was greater than the original difference

[21]:
```
# Your code here
diff_means = np.mean(a) - np.mean(b)
num = 10**4
counter = 0
for i in range(num):
    a_new, b_new = bootstrap(a, b)
    diff_means_each = np.mean(a_new) - np.mean(b_new)
    if diff_means_each >= diff_means:
        counter += 1

p_value = counter / num

print(f"P-value is {p_value}")
```

```
P-value is 0.9887
```

1.9 Summary

Well done! In this lab, you practice coding modern statistical resampling techniques of the 20th century! You also started to compare these non-parametric methods to other parametric methods such as the t-test that we previously discussed.