Resampling (Bootstrapping & Permutation Testing)

Matthew E. Parker Aug 8, 2019 · 5 min read

Common machine learning resampling methods like bootstrapping and permutation testing attempt to describe how reliably a given sample represents the true population by taking multiple sub-samples of the original sample; or, more precisely, how much confidence can be placed in the sample statistics. At first blush, however, these techniques may sound like statistical sorcery; how could samples of samples tell anything us anything about the population and not just about the original sample? Let's dig a little deeper into these two approaches to see what's really going on.

Bootstrapping

Bootstrapping is the process of taking multiple sub-samples from the original sample, of equal size to the original and with replacement, and then calculating sampling distributions of certain sample statistics using this new synthetic pool of samples.

For example, you may have conducted a poll of 100 people for your local newspaper (what is a newspaper?) and now you wish to know how reliable the statistics (mean, standard deviation, etc.) you have calculated about this sample are. The deadline for your article is tomorrow, so there's no way to gather additional samples from the population and of course no way to poll the entire city's population. Though 100 people seemed like a large sample when you were conducting the poll, you now worry that this sample is too small to tell you anything reliable about the city's 1 million inhabitants. This is where bootstrapping comes in.

The most vital aspect of resampling is the fact that it assumes the original sample is a good one; that is, each sample is independent and was collected randomly. Without these two assumptions about sound experimental design, we cannot progress any further.

So, assuming independence and randomness of the samples, we can bootstrap some new samples by sub-sampling randomly (random) with replacement (independent) from the original sample. In Python, this can be achieved with a single line of code if you have imported the NumPy library:

```
numpy.random.choice(sample, size=len(sample), replace=True)
```

To be usable, you will need to do this dozens (or hundreds, or millions, …) of times. From this new pool of synthetic samples you can calculate sampling distributions of various statistics, like the mean, which ultimately produces a confidence interval for that statistic in reference to the entire population.

Remember, since your experiment was well-constructed with truly independent and random samples, your original sample of 100 people is naturally very likely to mimic the population at large, regardless of its distribution. At the same time, bootstrapping brings the Central Limit Theorem into play where the sampl*ing* distributions of the sampl*ing* statistics will be normal in shape because you made a large number of bootstrapped sub-samples.

Furthermore, the process of bootstrapping itself has the effect of reducing the impact of any outliers from the original sample. Because bootstrapping randomly samples *with* replacement, any outliers are less likely too appear in the sub-samples than their occurrence rate in the original sample itself. For example, in a sample set [1,1,1,2,2,2,37], there will likely be many subsets with only 1's and 2's and very few with more than a single 37. While outliers are still present in the end, we have reduced the skew and kurtosis of our sampling distribution (i.e. narrowing the confidence interval).

Permutation Testing

Permutation testing works a bit differently than bootstrapping. The goal of a permutation test is to determine whether or not two given (random) samples are from the same larger population. The test itself is performed by calculating a test statistic for the given samples, commonly the difference between the sample means. Then, the two samples are merged together and then iteratively redivided into every possible *permutation* of two groups of similar sizes to the original two samples (in practice, though, we generally use combinations since order does not matter). For each iteration, the test statistic is calculated; taken together, the proportion of test statistics that are greater than or equal to the original test statistic provides the one-sided p-value for the null hypothesis (that the two original samples come from the same population).

This may seem a bit arbitrary in the abstract, but a simple example can help illustrate the efficacy of this method.

Let's say we have two sets, A & B:

A: [2,3]

B: [35,36,37]

The difference of their means is 33.5. There are 10 possible re-combinations of the merged A+B set [2,3,35,36,37], but let's look at one of them: [2,35] & [3,36,37]. In this new iteration we have sample means of 18.5 and 25.333…, producing a mean difference of 6.8333… . Since this is lower than the original mean difference of 33.5, this iteration does not contribute to the p-value. In fact, in this example the only combination that has a mean difference greater than or equal to the original is the original itself! This gives us a p-value of 0.1, making it likely that we should reject the null hypothesis that the two samples are from the same population.

As a counter example, if samples A and B were composed of values all in the mid-30s, the sample mean differences would all be roughly the same, producing a much higher p-value.

As with bootstrapping, permutation testing relies completely on the assumption that the given random samples were collected properly (i.e. all observations are independent and randomly selected from the population). Given this assumption, two random samples (of sufficient size) from a population should have similar means and distributions, resulting in a very small difference of means. Likewise, if the two samples are from different populations, their means and distributions should be significantly different from one another.

Turning this into Python code is a bit lengthier than with bootstrapping, and relies on the **itertools** and **numpy** libraries:

```
union = a + b
new_as = list(itertools.combinations(union, len(a)))
new_bs = []for new_a in new_as:
    u_copy = union.copy()
    for a in new_a:
        u_copy.remove(a)
    new_bs.append(u_copy)
ab_combos = list(zip(new_as, new_bs))


diff_mu_a_b = np.mean(a) - np.mean(b)
print(f"There are {len(ab_combos)} sample variations.")
num = 0     #Initialize numerator
for ai, bi in ab_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(ab_combos)

print(f'P-value: {p_val}')
```

Though the example above pretty obviously has samples from different populations, in practice the sample sizes are typically either too large to tell visually, or the values are much closer together, or both. In these scenarios, permutation testing is the preferred method for determining whether two samples come from the same population.

14

Statistics Python

More from Matthew E. Parker

Follow

data scientist and historian

Aug 8, 2019

Visualizing Data with Hexbins in Python

In the course of learning the basics of producing data visualizations with Python, I happened to run across a fairly uncommon style of graph called **hexbins**. Hexbin plots take in lists of X and Y values (and an optional list of Z values) and returns what looks somewhat similar to a scatter plot, but where the entire graphing space has been divided into hexagons (like a honeycomb) and all points have been grouped into their respective hexagonal regions with a color gradient indicating the density (or mean Z values) of each hexagonal area.

While at first mention this might seem…

Read more · 6 min read

36

Aug 8, 2019

Why I decided to learn data science?

Why did I decide to learn data science?

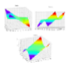
Several years ago, when I was just beginning coursework for my PhD in Medieval History at Saint Louis University, I had the privilege to take a course on Medieval Italy, taught by Prof. Thomas F. Madden. In this course, we covered a wide variety of subjects and cities, with each student being responsible for reading a different book each week on that week's topic, then reporting back a summary of the books' contents. …

Read more · 6 min read

14 1

More From Medium

3D visualization of a function of two variables (from ℝ² into ℝ) with Python -Matplotlib
Joséphine Picot in Analytics Vidhya



Deep Learning: Introduction to PyTorch
Pedro Borges in deeplearningbrasilia

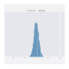
ETLs vs ELTs: Why are ELTs Disrupting the Data Market?
SeattleDataGuy in Coriers

The Data Is Ahead, What Will I Do Now?—2
Mehmet A.

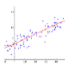
Quantum Computing
Alexander Barriga

Assumptions of Linear Regression
Utkarsh Kumar

Easily display animated media content inside your Jupyter Notebooks
Jérôme Buisine



Linear Regression From Scratch With Python
Sarvasv Kulpati in Sigmoid



Medium

About Help Legal