Exercise 6

Due Friday June 25 2021.

Some files are provided that you need below: E6.zip. As usual, **you may not write any loops** in your code (except where noted below).

A/B Testing

A very common technique to evaluate changes in a user interface is A/B testing *(https://en.wikipedia.org/wiki/A/B_testing)* : show some users interface A, some interface B, and then look to see if one performs better than the other.

I started an A/B test on CourSys at some point. I wanted to use it for this question, and it was going to be awesome. Then I learned that there was no significance anywhere and the numbers were much too small. So I faked some data. That's life, I guess.

The provided `searches.json` (load with Pandas using `orient='records', lines=True`) has information about users' usage of the "search" feature, which is where the A/B test happened. Users **with an odd-numbered uid were shown a new-and-improved search box**. Others were shown the original design.

The question I was interested in: do people search more with the new design? I see a few ways to approach that problem:

- Did more users use the search feature? (More precisely: did a different fraction of users have search count > 0?)
- Did users search more often? (More precisely: is the number of searches per user different?)

The number of searches is far from being normal. Unless you're more clever than me, you won't be able to transform it to anything that looks even slightly normal. (Data is integers, mostly zero. No transform will have much to work with.) As a result, we're going to be using **nonparametric** tests.

Create a program `ab_analysis.py` that takes an input file on the command line, and analyses the provided data to get p-values for the above questions. The provided `ab_analysis_hint.py` gives a template for output.

Hint 1 *(https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2_contingency.html)* . Hint 2 *(https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html)* . In your contingency table, your categories will be even/odd uid (aka control/treatment), and "searched at least once"/"never searched".

When analysing the (real, not faked) data, I realized: I don't care so much about **all** users. Instructors are the ones who can get more useful information from the search feature, so perhaps non-instructors didn't touch the search feature because it was genuinely not relevant to them.

Maybe we should ask about what happened to just the instructors from our data set. **Repeat the above analysis looking only at instructors**.

Report all of the p-values in the output format specified in the hint. [And note the question at the bottom about your results.]

Benchmarking Sorting

For this question, you will be benchmarking some code to see how it performs, and analysing the results for statistical significance. Throughout this course, we have been doing what most people do: run the program a few times (maybe with `%timeit`) and looking at the averages. We now know that isn't quite enough to draw conclusions.

In the provided `implementations.py`, I have implemented several sorting algorithms in different ways. Each sorts NumPy arrays not-in-place (i.e. they do not modify their input array, but return a new array).

The question is simple: what can you say about which implementations are faster/slower?

Data Creation

We are concerned with the seven sorting implementations given in the `all_implementations` array: `qs1`, `qs2`, `qs3`, `qs4`, `qs5`, `merge1`, `partition_sort`. As you might guess, there are several QuickSort *(https://en.wikipedia.org/wiki/Quicksort)* implementations, one Merge Sort *(https://en.wikipedia.org/wiki/Merge_sort)* , and one QuickSort-like thing I have named "partition sort".

Write a program `create_data.py` that generates random arrays, and uses time.time *(https://docs.python.org/3/library/time.html)* to measure the wall-clock time *(https://en.wikipedia.org/wiki/Wall-clock_time)* each function takes to sort them. You **may** write loops in the `create_data.py` program.

We are concerned about (1) arrays with random integers, which are randomly sorted, (2) arrays that are as large as possible, and (3) being able to meaningfully analyse the results. [Note: see the restrictions below.]

You're going to want enough data points that you can assume normality for your analysis. It's likely that your results will be somewhat-normally-distributed, but not precisely. You'll need a large enough n to proceed with tests that assume normality anyway.

You can import `implementations.py` and iterate through the relevant sorting functions like this:

```
import time
from implementations import all_implementations
# ...
for sort in all_implementations:
    st = time.time()
    res = sort(random_array)
    en = time.time()
```

In this program, create a DataFrame in a format that makes sense, and save it as `data.csv`, something like this:

```
data.to_csv('data.csv', index=False)
```

Restriction #1: processor time

Your `create_data.py` must **run in at most 60 seconds on a CSIL workstation** (single-threaded). You should be able to SSH to one of them *(https://www.sfu.ca/computing/about/support/covid-19-response--working-remotely/csil-linux-remote-access.html)* and run your code there; the modules you need should be installed.

You can check the running time, or enforce a time limit (respectively) with commands like these:

```
time python3 create_data.py # just check the time
timeout -s SIGKILL 60 python3 create_data.py # kill after 60 seconds
```

This restriction is imposed to limit the number of samples you can collect. This is generally realistic: I have sized this question so you don't have to wait too long to get data, but often the code you're benchmarking will take so long that it's prohibitive to collect thousands of data points. My intention is to give you a realistic data-collection limit, without making you wait overnight for "good" results.

You may also experience a certain amount of noise in your samples due to noisy neighbours *(https://en.wikipedia.org/wiki/Cloud_computing_issues#Performance_interference_and_noisy_neighbors)* . It's probably unlikely if you choose a random workstation, but more likely on one of the CPU servers.

Restriction #2: run count

When creating your data, you must **run each sorting implementation an equal number of times**.

In this artificial situation, you could run the test a few times, decide which sorting functions are more or less critical to have data for, and bias your experiment to collect more of the nearly-equal values. That's not realistic in general, where it might not be feasible to run the experiment a dozen times. (Technically, something like this could be done as a pilot study *(https://en.wikipedia.org/wiki/Pilot_experiment)* , but I'm going to call it “cheating by knowing the answer before your experiment”.)

Data Analysis

Once you have some data, you'll need to do some analysis to decide which sorting implementations are faster/slower/indistinguishable. You will need a statistical test that can be used to determine if the means of multiple samples are different, and then decide which ones. Hmm…

In your analysis, you can almost certainly apply the lesson from the “It's Probably Okay” section in lecture: if you have more than about 40 data points and they are reasonably-normal-looking, then you can use a parametric test.

Create a program `analyse_data.py` that reads the `data.csv` that you produced above and does the relevant statistical analysis. It should `print` the information you used to answer question 3 below, but there is no specific format required.

Aside: Why Aren’t More Users More Happy With Our VMs? *(https://www.microsoft.com/en-us/research/video/arent-users-happy-vms/)* , a research talk about how hard benchmarking code can be, and how tricky the analysis can be (which contains nothing that will help you do this exercise, but it's interesting anyway).

Questions

Answer these questions in a file `answers.txt`.

1. In the A/B test analysis, do you feel like we're p-hacking? How comfortable are you coming to a conclusion at $p < 0.05$?
2. If we had done T-tests between each pair of sorting implementation results, how many tests would we run? If we looked for $p < 0.05$ in them, what would the probability be of having any false conclusions, just by chance? That's the effective p-value of the many-T-tests analysis. [We could have done a Bonferroni correction *(https://en.wikipedia.org/wiki/Bonferroni_correction)* when doing multiple T-tests, which is a fancy way of saying “for m tests, look for significance at α / m”.]
3. **Give a ranking** of the sorting implementations by speed, including which ones could not be distinguished. (i.e. which pairs could our experiment not conclude had different running times?)

Submitting

Submit your files through CourSys for Exercise 6.

Updated Wed April 07 2021, 09:16 by ggbaker.