A ~~hitchhiker's~~ software tester's guide to randomised testing - Part 1

29 April 2018 · automation, heuristics, investigation, tools

Mostly Harmless,

I've talked and written about randomisation as a technique in software testing several times over the last few years. It's great to see people's eyes light up when they grok the concept and its potential.

The idea that they can create random test data on the fly and pour this into the app step back and see what happens is exciting to people looking to find new blockers on their apps path to reliability.

But it's not long before a cloud appears in their sunny demeanour and they start to conceive of the possible pitfalls. Here are a few tips on how to avert the common apparent blockers. (Part 1)

Problem: I've created loads of random numbers as input data, but how will I know the answer the software returns, is correct? - Do I have to re-implement the whole app logic in my test code?

Do you remember going to the fun-fair as a kid? Or maybe you recall taking your kids now as an adult? If so then you no doubt are familiar with the height restriction - Do you meet the [e.g.] 3ft / 1m minimum height that lets you ride the ‘roller coaster of doom’?

The pimple-faced, minimum waged and minimally enthused teenager, standing guard at the entrance to the ride, was not daunted by the regulatory burden of the height restriction. He didn’t need to measure each child with a laser-ruler or tape measure. He didn’t need to remove each child’s shoes or shave each child’s head to ensure an accurate measurement, from their scalp to the ground. Nor did they feel the need to scour the relevant [and no doubt confusing] EU regulations on the subject of amusement-park attendee height regulation.

They just had a line painted on the wall next to the entrance. The line was also probably slightly higher than the stated 1metre (3ft), to give the company a slight safety margin on height. All our pimple faced youth needed to do was glance up from their is iPhone every few seconds to guesstimate whether the child is above the line. Their job was made more manageable by the fact that the ride-takers would obscure the measurement warning if they were tall enough. Almost zero thinking was required.

Using that approach in your testing is one of the simplest tricks to implement and use. If you are testing a complicated calculation of algorithm, find out the properties of ‘good’ or correct answers.

Is a good answer:

1) Positive?

2) <100?

3) Only an integer (whole number) or unlikely to be an integer?

4) Proportional to another value? (e.g. if X is big then Y is probably small)

5) Found after several seconds of processing time?

If the system behaves ‘good’, then we might be able to assume it handled those inputs OK, and look at the next set - without too much thought. We know that scenario was probably mostly harmless.

You can search for behaviours that don’t match the ‘good’ and use these as a starting place for your more in-depth testing.

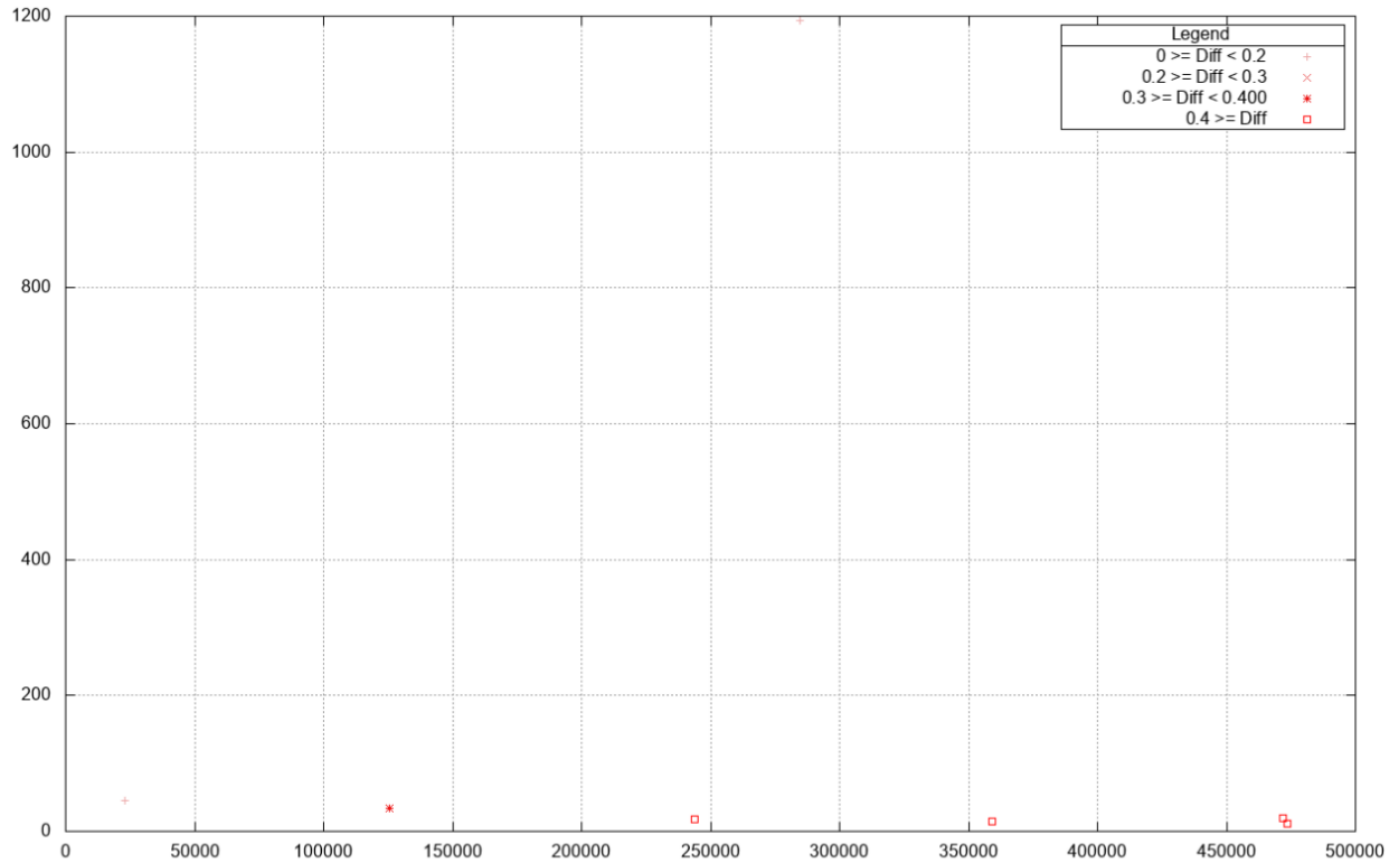
e.g. The system returned 3736.2. The percentage value in this field should be <=100 so something might be bust.

e.g. The answer was returned in 10ms. The answer usually takes 400ms to be calculated and returned. Maybe the app gave a cached response?. Did the app connect to the database [located in Dublin/Seattle/Berlin etc]? We should look into that.

This approach is easy to implement in an automated-check. You might also find presenting the results in other ways reveals more information about the bugs in the system. The binary pass/fail used for simple unit tests or BDD scenarios doesn't tell us much about the problems infesting our app.

For example, I tested a system with dozens of numerical calculations. Many of the calculations fed into other calculations and so on - and it was tricky and time-consuming to track down the cause of the errors. It was especially difficult as the issues appeared at first to be intermittent.

Rather than labour by hand, to try and pinpoint when there were miscalculations and why other inputs seemed to work fine - I graphed the data I had used, and just marked the results in a different colour/symbol.



Here is one of the graphs I produced, after I had adjusted it to show the aberrant results and ignore the 'good' results.

As you can see, the system strayed from the correct answer generally when the Y (vertical/left) axis numbers were lower [than the X values], while the X (horizontal/bottom) axis values could cover a wider domain and still see the bug.

To allow my testing to utilise both automated checks for sensible values and more exploratory testing; I structure my tools so that they are reusable and I can easily parameterise them with random test data.

To improve your testing tools try to avoid the low-fidelity example tables used by [development methodology oriented] tools like Cucumber. While people might use them as development aids, they don’t usually scale to this kind of exploratory investigation. The kind of investigation that finds bugs - and lets you dig down to see more.